PARALLEL SOLUTION OF SVD-RELATED PROBLEMS, WITH APPLICATIONS

A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy
in the Faculty of Science and Engineering

October 1993

By
Pythagoras Papadimitriou
Department of Mathematics

Copyright

Copyright in text of this thesis rests with the Author. Copies (by any process) either in full, or of extracts, may be made **only** in accordance with instructions given by the Author and lodged in the John Rylands University Library of Manchester. Details may be obtained from the Librarian. This page must form part of any such copies made. Further copies (by any process) of copies made in accordance with such instructions may not be made without the permission (in writing) of the Author.

The ownership of any intellectual property rights which may be described in this thesis is vested in the University of Manchester, subject to any prior agreement to the contrary, and may not be made available for use by third parties without the written permission of the University, which will prescribe the terms and conditions of any such agreement.

Further information on the conditions under which disclosures and exploitation may take place is available from the head of Department of Computer Science.

Contents

Copyright **2**

Abstract **9**

Declaration **11**

Acknowledgements **12**

1 Introduction **13**

2 The Kendall Square Research KSR1 **17**

2.1 Introducing the KSR1 17

2.2 Architectural Overview 18

2.3 KSR Fortran 22

2.3.1 Pthreads 23

2.3.2 Parallel Regions 24

2.3.3 Parallel Sections 26

2.3.4 Tile Families 27

2.3.5 Compiling, Running, and Timing a KSR Fortran Program 30

2.3.6 Language Extensions 34

2.3.7 Numerical Software 37

2.4 Experiences and Remarks 38

3 Two-sided Procrustes-type problems **40**

3.1 Introduction 40

3.2 The Most General Problem . . . 41

3.3 The Two-Sided Orthogonal Procrustes Problem . . . 43

3.4 The Two-sided Rotation Procrustes Problem . . . 46

3.5 Permutation Problems . . . 49

3.5.1 Maximizing the Trace of PA . . . 49

3.5.2 Solving the TSPOPP . . . 55

3.5.3 A Two-Sided Permutation Problem . . . 62

3.6 The Two-Sided Symmetric Procrustes Problem . . . 65

3.7 Concluding Remarks . . . 72

4 The Polar Decomposition 74

4.1 Introduction . . . 74

4.2 Properties of the Polar Decomposition . . . 75

4.3 Sequential Algorithms . . . 78

4.4 The Matrix Sign Function and the Polar Decomposition . . . 81

4.5 A Parallel Algorithm for the Polar Decomposition . . . 88

4.6 Implementation on the KSR1 . . . 96

4.7 Experimental Results . . . 100

4.8 Applications of the Polar Decomposition . . . 103

4.9 Conclusions . . . 108

5 A New Approach to Computing the SVD 109

5.1 Introduction . . . 109

5.2 The Classical Jacobi Method . . . 112

5.3 Cyclic Schemes . . . 117

5.4 Threshold Jacobi Methods . . . 119

5.5 Parallel Jacobi Methods . . . 122

5.6 A Reduced Cyclic Jacobi Method . . . 129

5.7 The Davies-Modi Method . . . 135

5.8 Block Jacobi Methods . . . 139

5.9 Implementing Jacobi methods on the KSR1 150
5.10 Numerical and Timing Results for the SEP 157
5.11 Numerical and Timing Results for the SVD 163
5.12 Applications of the SVD 166
5.12.1 The SVD in Regression Analysis 166
5.12.2 Some Less Well-known Applications of the SVD 171
5.13 Conclusions 174

6 Concluding Remarks and Future Work **176**

A Listings of KSR Fortran Routines **178**
A.1 The source code for Algorithm PJDM 178
A.2 The source code for Algorithm Method 1 186
A.3 The Source Code for Algorithm Method 2 196
A.4 Auxiliary Routines 205

Bibliography **215**

List of Tables

3.7.1 Two-sided Procrustes Problems and tools for their solution. 72

4.4.1 Similarities between the polar decomposition and the matrix sign decomposition 85
4.5.1 Behaviour on a 10×10 Vandermonde matrix. 94
4.6.1 Timing results for SGEMM (Mflops) for the default value of PS. 97
4.6.2 Speedups for SGEMM for the default value of PS. 97
4.6.3 Timing results for SGEMM (Mflops) with PS = 1. 98
4.6.4 Speedups for SGEMM with PS = 1. 98
4.6.5 Timing results for SGETRF/SGETRI (Mflops) 99
4.7.1 Algorithm Parallel Polar with 8 processors, $n = 1024$. 101
4.7.2 Algorithm Parallel Polar with 16 processors, $n = 1024$. 102
4.7.3 Newton iteration with 16 processors, $n = 1024$. 102

5.4.1 Comparison of five threshold strategies. 121
5.6.1 Convergence for Algorithms Parallel Jacobi and X. 134
5.8.1 Numerical results for the partial Jacobi method. 148
5.9.1 Timing results for four scalar parallel Jacobi algorithms. 151
5.9.2 Some numerical and timing results for Method 1 152
5.10.1 Timing results for SSEYV. 158
5.10.2 Timing results for Method 2. 158
5.10.3 Numerical and timing results for Method 2. 159
5.10.4 Numerical and timing results for Variant 1. 159
5.10.5 Numerical and timing results for Variant 2. 160

5.10.6 Numerical results with matrices with close eigenvalues. 161
5.10.7 Numerical and timing results for Variant 3. 162
5.10.8 Numerical and timing results for Variant 4. 163
5.10.9 Timing results for Method 2 and its variants. 163

List of Figures

2.2.1 KSR1 hierachy of rings. 19

2.2.2 The page allocation procedure. 21

2.2.3 The KSR1 processor. 22

2.3.1 Pthread execution pattern. 24

2.3.2 Iteration space and data space. 28

2.4.1 The 256 kilobytes data cache inside a subcache. 39

3.5.1 A bipartite graph. 54

3.5.2 A maximal matching. 55

3.5.3 A possible form of the TSPOPP. 61

4.6.1 SGETRF/SGETRI performance for the KSR1. 99

5.5.1 A merry-go-round scheme for players 2 through 8. 126

5.7.1 Performance of two methods for determining δ. 138

5.8.1 The first alternative. 143

5.8.2 The second alternative. 143

5.11.1 Computing the SVD via the Polar Decomposition and the SEP. 165

5.12.1 The first two principal components. 170

Abstract

This thesis is motivated by a class of least squares approximation problems known as two-sided Procrustes-type problems. For their solution these problems require three decompositions that play a central role in numerical linear algebra: the polar decomposition, the SVD, and the spectral decomposition of a symmetric matrix.

We begin by discussing the Kendal Square Reserach KSR1, the parallel computer on which we implemented our algorithms. We focus our attention on issues that concern a numerical analyst. However, a description of the memory system of the KSR1 is also given, since the main requirement in designing efficient algorithms for the KSR1 is a clear understanding of its memory system.

We survey two-sided Procrustes-type problems, deriving analytic solutions where possible and otherwise developing numerical methods for their solution.

We discus existing algorithms for computing the polar decomposition and then present a new parallel algorithm for this decomposition. The implementation of this algorithm on the KSR1 is discussed, and timing results show that our implementation is faster than existing methods for computing the polar decomposition of a large matrix on the KSR1.

An observation on the relation between the polar decomposition and the singular value decomposition reveals a new approach to computing the SVD. This approach requires the computation of the sprectral decomposition of a symmetric matrix. We discuss and investigate on the KSR1 several schemes based on the Jacobi method for the solution of the symmetric eigenvalue problem. Two of these methods, both parallel block Jacobi methods, are the fastest way we know for computing the spectral decomposition of a large dense symmetric matrix on the KSR1. We use a combination of our parallel algorithm

for computing the polar decomposition and one of these methods for the computation of the SVD, and find that this approach is significantly faster than the LAPACK SVD routine on the KSR1.

Applications are also discussed in this thesis. Applications of the two-sided Procrustes-type problems are mentioned in the corresponding chapter. Applications of the polar decomposition and of the SVD, the decompositions for which we derived new parallel algorithms, are discussed in detail.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institution of learning.

Acknowledgements

It is pleasure for me to acknowledge the excellent opportunities I have received for research and study in the Department of Mathematics at the University of Manchester. I would need many more pages to thank in full appreciation all the people that they have led, directly or indirectly, to the submission of this thesis. I would like to apologize to all of them that I will not mention their names, promising that I will never forget their contribution. I thank the Science and Engineering Research Council for paying the tuition fees of my postgratude studies at the Manchester University. I owe a debt of gratitude to Professor Christopher Baker for his support to my applications to attend two international conferences abroad, Dr. Len Freeman for the time he spent for considering my questions, Dr. George Hall for his efforts in order to be awarded with the Harvey Goodwill grant, and Mr. Graham Riley for his advice on the use of the KSR1. I extend my warmest thanks to my colleagues and the members of staff in the Department of Mathematics for their cooperation during this project. It is hard to find words, even in Greek, to express my gratitude to my supervisor Dr. Nicholas Higham, not only for the influential supervision of this work, but primarily for giving me the valuable and unique experience of working with him. Without Dr. Nicholas Higham and another two people this work would have never be done. These are my parents, Dimitris and Aspa Papadimitriou, the people to whom I owe everything.

Chapter 1

Introduction

This thesis is motivated by a class of constrained least squares approximation problems which will be referred to as *two-sided Procrustes-type problems.* These problems can be described as follows: given A, $B \in \mathbf{C}^{m\times n}$ $(m \geq n)$, find matrices $X \in \mathbf{C}^{m\times m}$ and $Y \in \mathbf{C}^{n\times n}$ with a given property to minimize

$$\|XAY - B\|_F,$$

where $\|\cdot\|_F$ denotes the Frobenius norm ($\|A\|_F = (\sum_{i,j} |a_{ij}|^2)^{1/2}$). Problems with $Y \equiv I_n$ will be referred to as *one-sided Procrustes-type problems.* An example of this class is the orthogonal Procrustes problem [106], where the unknown matrix X is a unitary matrix. The general two-sided problem has not received the same amount of attention as the one-sided one, and there are still many open questions associated with it, some of which we consider in this thesis.

The investigation of a class of two-sided Procrustes-type problems revealed the main tools required for their solution. Among them are the polar decomposition, the spectral decomposition of a symmetric matrix, and the singular value decomposition (SVD). These three decompositions play an important role in numerical linear algebra and they are also important tools in numerous applications. The two-sided Procrustes-type problems discussed in this thesis occupy just a narrow band in the broad spectrum of their applications.

The above-mentioned decompositions have received a great deal of attention from

the numerical linear algebra community and they have been studied in depth for many years. Furthermore, highly sophisticated software for their traditional approaches has been developed for conventional (sequential) computers.

The advent of parallel computers some twenty five years ago both provided an opportunity and prompted a need to develop parallel algorithms. The spectral decomposition of a symmetric matrix and the singular value decomposition have attracted research interest from the early days of parallel computing. The polar decomposition, probably due to the fact that it can be obtained via the SVD, has not received the same amount of attention. We are not aware of any parallel algorithm developed especially for the polar decomposition. However, the development of parallel algorithms for the symmetric eigenvalue problem and the SVD did not saturate the research interest in this field, since the efficiency of a parallel algorithm is strongly connected with the underlying architecture of the parallel machine. Thus, the introduction of a new parallel architecture may cause reconsideration of the existing parallel algorithms. An example of an advanced parallel architecture that has been introduced recently (1991), is the Kendall Square Research KSR1.

The KSR1 is a virtual shared memory MIMD computer system. The bulk of our computational work took place on a 32-processor configuration KSR1, installed at the Centre for Novel Computing at the University of Manchester. Having been familiar with the KSR1, our interested was focused on developing parallel algorithms for the above-mentioned decompositions and implementing them on this parallel machine.

We started with the polar decomposition. It is a well-known result that any matrix $A \in \mathbf{C}^{m\times n}$ $(m \geq n)$ has a polar decomposition $A = UH$, where $U \in \mathbf{C}^{m\times n}$ has orthonormal columns and $H \in \mathbf{C}^{n\times n}$ is Hermitian and positive semidefinite. If A has full rank then H is nonsingular and U is unique. Having developed, analyzed, and implemented a parallel algorithm for the polar decomposition we made the following observation: if $H = VDV^*$ is the spectral decomposition of the Hermitian positive semidefinite polar factor H of A, where V is unitary and $D = \mathrm{diag}(d_i)$ with $d_1 \geq d_2 \geq \cdots \geq d_n \geq 0$, then $A = P\Sigma Q^*$ is a SVD of A, where $P = UV$, $\Sigma = D$ and $Q = V$. The above observation

suggested a new approach to computing the SVD, and led our research to investigating parallel methods for the symmetric eigenvalue problem.

Unlike the polar decomposition, the symmetric eigenvalue problem has received much attention from the parallel computing community. There is a rich literature on parallel methods for the symmetric eigenvalue problem. Most of these methods are based on the Jacobi method for computing the eigenvalues and the eigenvectors of a symmetric matrix. We experimented with several versions of the Jacobi method on the KSR1 and we found that block versions are more suitable for this parallel machine. According to our experimental results, two of the block Jacobi methods that we implemented on the KSR1 were found to be the fastest methods known to us for computing the spectral decomposition of large dense symmetric matrices on this parallel system. Moreover, the combination of our parallel algorithm for computing the polar decomposition and of these block Jacobi methods for the computation of the spectral decomposition of a symmetric matrix, was found to be the fastest stable method we know for computing the SVD of large dense matrices. Our parallel methods have been compared with parallelized versions of the corresponding LAPACK routines, and parallelized versions of corresponding existing sequential algorithms.

This thesis is also concerned with applications. There are two separate sections devoted to applications of the polar decomposition and the SVD, the decompositions for which we derive new parallel algorithms. Applications of the two-sided Procrustes problems discussed in this thesis, most of them applications of the SVD and the polar decomposition, are not discussed in detail. However, their known to us applications of these two-sided Procrustes-type problems are mentioned during their discussion.

This thesis is structured in the following way. Chapter 2 introduces the KSR1, and can be viewed as a numerical analysis perspective of the KSR1. Here, having described the KSR1 we put emphasis on issues that concern a numerical analyst. These include an introduction to the parallel constructs of the KSR1, KSR Fortran, and numerical software. In the final section of this chapter we report our experiences as a pioneer user of the KSR1. Chapter 3 concentrates on a class of two-sided Procrustes-type problems, suggesting

methods for their solution. Chapter 4 is the core of this thesis. In this chapter, having discussed the existing sequential algorithms for the polar decomposition, we present a parallel algorithm for its computation and discuss its implementation on the KSR1. In the same chapter we state numerical and timing results concerning the performance of our parallel algorithm and other existing techniques on the KSR1. Applications of the polar decomposition are also discussed in this chapter. The bulk of Chapter 5 is concerned with Jacobi methods for the solution of the symmetric eigenvalue problem. In the first sections we discuss sequential and parallel scalar Jacobi methods. We also discuss and examine two schemes designed to improve the performance of a parallel Jacobi method. Then, we present two parallel block Jacobi methods and we discuss their implementation on the KSR1. The implementation of parallel scalar Jacobi algorithms on the KSR1 is also discussed. In the same chapter we state numerical and timing results concerning the performance of our algorithms for the symmetric eigenvalue problem and the SVD, and of the correspondind standard and parallelized LAPACK routines on the KSR1. There is also a separate section in this chapter devoted to applications of the SVD. Finally in Chapter 6 we state our conclusions and discuss our future research plans. Listings of our KSR Fortran routines for the polar decomposition of a general matrix and the spectral decomposition of a symmetric matrix are given in Appendix. The source codes for the most important algorithms mentioned in this work can be found inside the back cover of this thesis.

Chapter 2

The Kendall Square Research KSR1

2.1 Introducing the KSR1

In this chapter we introduce the KSR1, the parallel machine that we used for our research. This chapter may be viewed primarily as a numerical analysis perspective of the KSR1. As numerical analysts, rather than computer scientists, we focus our attention on issues concerning numerically intensive processing, and it is beyond the scope of this chapter to describe technical details in depth. However, a description of the KSR1's memory system is inevitable. The main requirement in designing efficient algorithms for the KSR1 is a clear understanding of its memory system, which differentiates the KSR1 from most advanced computer architectures. An architectural overview of the KSR1 is given in Section 2.

Kendall Square Reseach released the KSR1 in 1991. The KSR1 is a virtual shared memory MIMD computer system designed to run a broad range of mainstream applications, ranging from numerically intensive computation, to on-line transaction processing and database management and inquiry [76]. The KSR1 claims to be the first MIMD shared memory machine which has the scalability[1] of highly parallel message passing systems[2]. These systems are scalable to large number of processors, but their use requires the management of the complex details of work distribution, data placement,

[1] *Scalability* is a term that is used to signify whether a given parallel architecture shows improving performance for a parallel algorithm as the number of processors increases.

[2] A typical example of this class the Intel iPSC/1.

message generation, and scheduling. On the other hand, true shared memory multiprocessors[3] offer the simple programming model of a single address space, but they are less scalable due to the cost of the processor–memory switch. The KSR1 has been designed to combine the advantages of both categories, providing a user friendly environment and scalability. A study on the scalability of the KSR1 can be found in [99]. At present, a KSR1 system contains from 8 to 1088 processors. The KSR1 installed in the Centre for Novel Computing at the University of Manchester is a 32-processors configuration with 1 gigabyte of memory and 10 gigabytes disk capacity. Its peak computational performance is 1.28 gigaflops. Each processor is a superscalar 64-bit unit able to issue up to two instructions every 50 nanoseconds, giving a performance rating of 40 MIPS and a peak floating point performance of 40 megaflops.

The KSR operating system is an extension of the OSF/1 version of Unix [74]. There is also an extensive set of programming languages and compilers, including Fortran, C, C++, Cobol, and assembly language. We consider only the Fortran programming environment which is discussed in Section 3. The Kendall Square Research also supplies the KSRlib/BLAS library [77] and the KSRlib/LAPACK library [78]. Both libraries have been used extensively for our research, and they are also discussed in Section 3. For applications that require database management capability, the KSR1 supports the relational database management system ORACLE7.

2.2 Architectural Overview

Unlike typical high-performance computers, which have large pools of main memory and small caches, the KSR1 has main memory consisting of large, communicating local caches, each capable of storing 32 megabytes. This feature characterizes the KSR1 as a Cache-Only Memory Architecture (COMA) system [53, 70]. The programmer perceives the KSR1 memory system as a collection of processors connected to a shared memory. The shared memory is addressed by a 64-bit System Virtual Address (SVA). The contents of SVA locations are physically stored in a distributed fashion, in a collection of local caches.

[3] A typical example of this class is the CRAY C90.

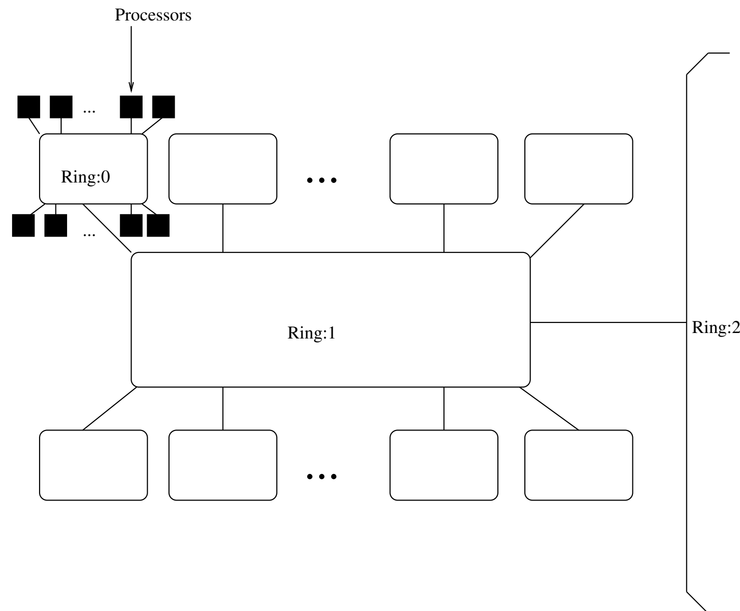

Figure 2.2.1: KSR1 hierachy of rings.

The KSR architecture supports an SVA space of 2^{64} bytes; the KSR1 implementation supports an SVA space of 2^{40} bytes (1 terabyte). There is one local cache for each processor in the system. Up to 32 processors are connected to a slotted, pipelined ring, called a Ring:0. The general KSR architecture is a multiprocessor system composed of a hierachy of rings. The lowest level, Ring:0, consists of a 34 message slots connecting 32 processing cells (processors) and two cells responsible for routing to the next higher layer ring, Ring:1. A fully populated Ring:1 is composed of the interconnecting cells from 34 Ring:0 rings, providing up to 1088 processors. A fully configured KSR1 is composed of only two layers. The general KSR architecture provides for a third layer which connects Ring:1 rings into a Ring:2 layer. Figure 2.2.1 shows the hierarchical ring structure of the KSR multiprocessor. In what follows we only consider a KSR1 with a single Ring:0 installed. The *search engine* interconnects the local caches and provides routing and directory services for the collection of local caches. As a result, all the local caches behave logically as a single, shared address space. The combination of the local caches and the search engine is referred to as the ALLCACHE memory system.

As we mentioned earlier, a Ring:0 ring has 34 message slots, where 32 are constructed

for the 32 processors and the remaining two slots are used for the directory cell connecting to the Ring:1 ring. Each slot can be loaded with a packet, made up of a 16 byte header and a 128 byte of data. This is the basic data unit in the KSR1, called a *subpage*. A processor in a ring ready to transmit a message waits until an empty slot is available, which rotates through a ring interface of the processor. A single bit in the header of the slot identifies an empty slot. The interface between the processor and the ring is provided by a *Cell Interconnect* (CI). The role of a CI can be described as follows. If a local cache cannot satisfy a request from its local processor, a request message is inserted into the Ring:0 ring when an empty slot is available to its CI. As the request message passes each CI of the ring, that CI checks to see if the requested data is present in its local cache. If the data is available in that local cache in the appropriate state, the CI will extract the request and then will insert a response into the ring. The response will be rotated back to the requested processor.

ALLCACHE stores data in units of *pages* and *subpages*. Pages contain 16 kilobytes (2^{14} bytes), divided into 128 subpages of 128 (2^7) bytes. The unit of allocation in local caches is a page, and each page of SVA space is either entirely represented in the system or not represented at all. As we mentioned earlier, the unit of transfer and sharing between local caches is a subpage. The page allocation procedure may be described as follows. When a processor references an address not found in its local cache, ALLCACHE memory creates a space for it there by allocating a page. Each local cache can accommodate 2048 (2^{11}) pages, or equivalently its capacity is 32 megabytes. The contents of the newly allocated page are filled as needed, one subspace at a time.

The page allocation procedure is illustrated in Figure 2.2.2, where we suppose that processor B executes a load instruction for address X and that the corresponding page is not found in local cache B. In this case, ALLCACHE memory allocates a complete page in B's local cache, but only the subpage containing X is copied into local cache B. All other subpages in that page are marked invalid in local cache B. However, a later reference by processor B to address Y will not require a new page allocation. In this case, ALLCACHE memory will find that the page is already allocated, and the search engine

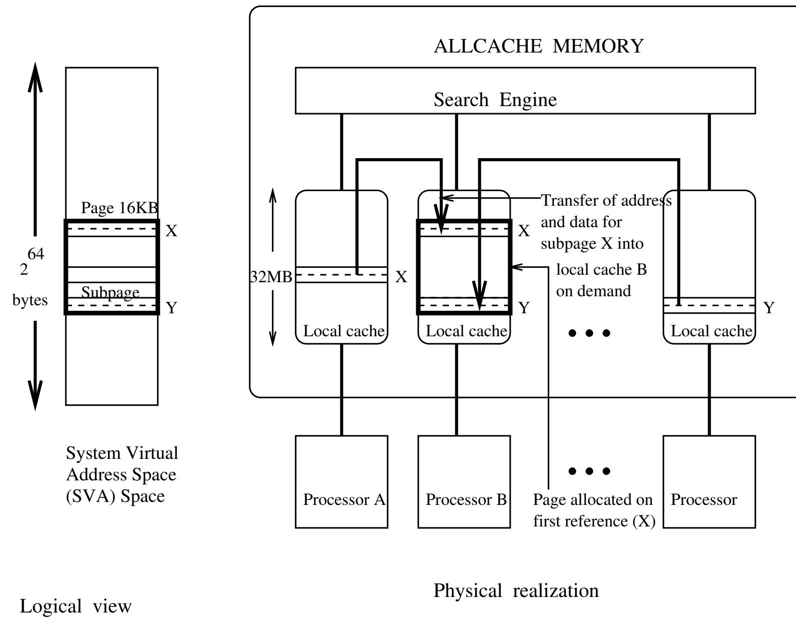

Figure 2.2.2: The page allocation procedure.

will be instructed to find the required subpage in another cache and copy the address and data into the already allocated page on local cache B.

The KSR1 processor consists of 12 custom CMOS chips:

- The Co-Execution Unit (CEU) fetches all instructions, controls data fetch and store, controls instruction flow, and does the arithmetic required for address calculations.
- The Integer Processing Unit (IPU) executes integer arithmetic and logical instructions.
- The Floating Point Unit (FPU) executes floating point instructions.
- The eXternal Input/output Unit (XIU) performs data management administration and programmed I/O.
- Four Cache Control Units (CCU) are the interface between the 0.5 megabytes subcache and the 32 megabytes local cache. (The subcache is discussed in the following paragraph.)
- Four Cell Interconnect Units (CIU) are the interface between a processor and the Ring:0 ring.

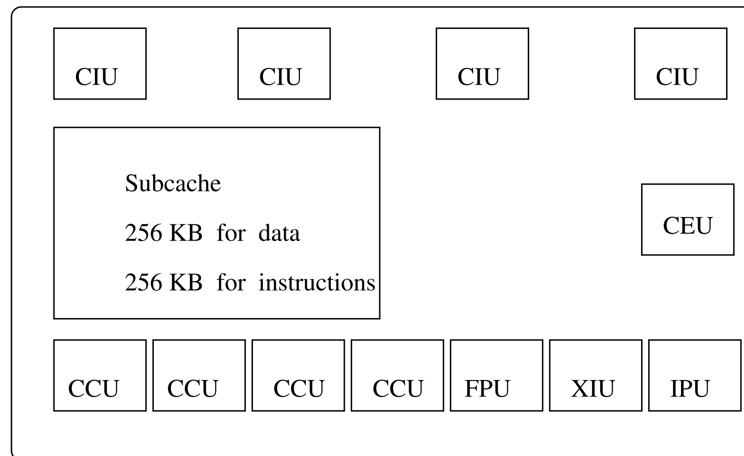

Figure 2.2.3: The KSR1 processor.

Each processor also contains a 256 kilobytes data cache and a 256 kilobytes instruction cache, which comprise a first-level cache called the *subcache*. A clear understanding of how the subcache works may improve significantly the performance of a code on the KSR1. This issue is discussed in Section 4. The KSR1 processor is depicted in Figure 2.2.3.

2.3 KSR Fortran

The KSR Fortran [73] is an extended version of Fortran 77, the standard established by the American National Standards Institute (ANSI). Among the extensions beyond Fortran 77 are additional data types, allocatable arrays, array syntax, and additional control statements. KSR Fortran also supports an extended set of intrinsic and external procedures, in addition to the basic set of Fortran 77 intrinsic functions. However, although KSR Fortran includes some elements of Fortran 90, for example allocatable arrays and array syntax, it is far from fulfilling the requirements of Fortran 90 or the High Performance Fortran (HPF) standard [15]. The above-mentioned language extensions are discussed later in this section.

The KSR Fortran compiler provides a powerful set of parallel processing capabilities.

The major parallel constructs of KSR Fortran are *pthreads*, *parallel regions*, *parallel sections*, and *tile families*. Low level parallelism is expressed with pthreads, while high level parallelism is expressed using the other three parallel constructs. The parallel constructs are expressed as compiler directives that are processed by the KSR Fortran compiler but ignored by other Fortran compilers. (As we will see later in this section, other Fortran compilers consider the directives as comments.) This approach supports program portability. However, correct programs that use the parallel region constract may need minor code changes to run correctly on other machines[4].

2.3.1 Pthreads

A pthread is a sequential flow of control within a process that cooperates with other pthreads to solve a problem. A program may be divided into multiple chunks of work which can be run in parallel. Since the assigned work of a pthread is usually executed on the same processor, cache misses are reduced by controlling what work is assigned to a pthread. The assignment of work to pthreads can be controlled by *teams* of pthreads. A team is a group of pthreads with a team identification number. The execution of a parallel KSR Fortran program consists of the execution of one or more team of pthreads. The execution of the program begins with a single pthread, called the *program master*. When a pthread encounters a begin-parallel directive, that pthread assembles a team. The pthread that assembles the team is called team *leader* and the other pthreads are team *members*. Each parallel directive can take an identification number as a parameter. If so, the specified team will execute the parallel domain. Otherwise, the KSR Fortran run-time system identifies a team to execute the parallel domain. If needed, the KSR Fortran run-time system creates a new team. When all the pthreads in this team reach the end-parallel directive, the team is usually disbanded, and the team members return to the idle pool. The team leader continues the execution from the next statement after the end-parallel directive.

Each pthread in a team has a sequence number between 0 (the team leader's number)

[4] Changes will be needed if the program depends on the unique identifier of pthreads.

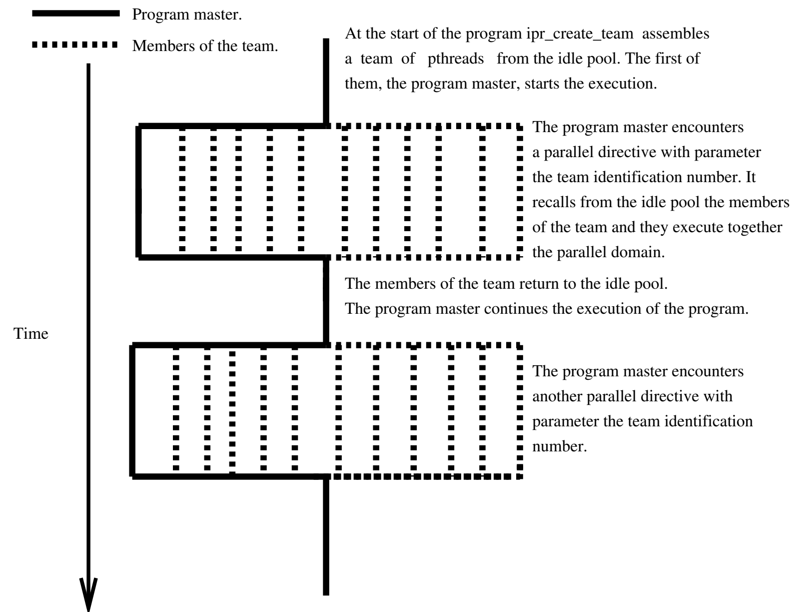

Figure 2.3.1: Pthread execution pattern.

and $n - 1$, where n is the number of pthreads in a team. The sequence number of a pthread may be used to control the assignment of work to particular pthreads. A team of pthreads can also be created explicity by a call to the library routine `IPR_CREATE_TEAM`. This routine takes as an input argument the desired number of team members, assembles the requested number of team members, minus one (the team leader), from the idle pool, and returns the team identification number. Creating explicitly the teams of pthreads and stating the team identification number in every parallel directive, the user takes from the run-time system the control of the teams of pthreads. In our codes we used only one team of pthreads, created explicitly by `IPR_CREATE_TEAM`. The team identification number returned by `IPR_CREATE_TEAM` was used as a parameter in each parallel directive, to order the same team of pthreads to execute the parallel domain. Figure 2.3.1 describes a typical pthread execution pattern that we used in our implementation.

2.3.2 Parallel Regions

The parallel region construct enables the user to execute in parallel multiple instantiations of a single code segment. Parallel regions are declared by the programmer, by

enclosing the desired code within PARALLEL REGION and END PARALLEL REGION directives. The role of a parallel region in a KSR Fortran program may be illustrated by the following example. Suppose that we use one team of eight pthreads and we wish to compute the Cholesky factorizations of 16 $n \times n$ real symmetric positive definite matrices A_k, $k = 1, \ldots, 16$. Since we use a team of eight pthreads, we can compute simultaneously the Cholesky factorizations of the first eight matrices, and then simultaneously the factorizations of the rest matrices. This can be implemented as follows, using the LAPACK routines SPOTRF and SPOTRI for the Cholesky factorization [1].

```
C*KSR* PARALLEL REGION (TEAMID = TEAM, PRIVATE = (K,MYNUM))
        MYNUM = IPR_MID()
        DO K=1,16
             IF (MOD(K,NPROCS) .EQ. MYNUM) THEN
              CALL SPOTRF('L', N, A(1,1,K), LDA, INFO(K))
              CALL SPOTRI('L', N, A(1,1,K), LDA, INFO(K))
             END IF
        END DO
C*KSR* END PARALLEL REGION
```

In the above sample code, each pthread calls the library routine IPR_MID which assigns a sequence number between 0 and 7 to the parameter MYNUM. The parameters K and MYNUM have been declared as PRIVATE since their values must not be shared within the parallel region[5]. The parameter TEAM is the team identification number, which has been determined earlier by IPR_CREATE_TEAM. The variable NPROCS represents the number of processors, and it is assumed to be equal to the number of pthreads. The IF statement in the body of the DO loop actually determines which pthread will compute the Cholesky factorization of the matrix A_k. Each pthread computes two Cholesky factorizations.

The call to IPR_MID returns the sequence number of the executing pthread in the team. This number can be used in order to assign work to particular pthreads. If the

[5]When a program unit encounters a parallel directive, the program unit's local variables are shared by default within the parallel routine. This default can be overriden with the PRIVATE parameter of the parallel directive.

same team is used to execute several parallel regions sequentially, as it happens in our codes, a particular pthread can choose work in one parallel region which is consistent with the choice made in a previous parallel region. In this way, the locality of reference is maximized, thus reducing cache misses.

2.3.3 Parallel Sections

A parallel section is a group of different code segments that may be run in parallel. The code segments within a parallel section are called *section blocks*. If we do not assign a team of pthreads for the execution of a parallel section the run-time system creates one pthread for each section block, and in general, each section block is executed on a different processor. The pthreads that execute the section blocks within a parallel section run simultaneously.

The code of a parallel section is included within `PARALLEL SECTIONS` and `END PARALLEL SECTIONS` directives. All code within a section must belong to a section block, and the section blocks may not overlap. In the following code extract a team of 3 pthreads is being used to compute the inverse of an $n \times n$ real matrix A, the singular value decomposition of a $m \times n$ real matrix B, and the spectral decomposition of an $n \times n$ symmetric matrix C simultaneously. The LAPACK routines `SGETRF` and `SGETRI` have been for the matrix inversion, `SGESVD` for the singular value decomposition, and `SSEYV` for the spectral decomposition [1].

```
C*KSR* PARALLEL SECTIONS ( TEAMID = TEAM )
C*KSR* SECTION
         CALL SGETRF ( N, N, A, LDA, IPIV, INFO)
         CALL SGETRI ( N, A, IPIV, WORKA, LWORKA, INFOA)
C*KSR* SECTION
         CALL SGESVD ( JOBU, JOBVT, M, N, B, LDB, S, U,
     &                 LDU, VT, LDVT, WORKB, LWORKB, INFOB )
C*KSR* SECTION
         CALL SSEYV  ( JOBZ, UPLO, N, C, LDC, W, WORKC,
```

```
     &                 LWORKC, INFOC)
C*KSR* END PARALLEL SECTIONS
```

As in the above example, the parallel sections construct should be used for the parallel execution of a fixed number of independent tasks. It is noteworthy that the above code extract would be understood by any other Fortran 77 compiler, since the KSR directives would be considered as comments.

2.3.4 Tile Families

Loop parallelization in KSR Fortran is achieved by *tiling*, in which the iteration space defined by a Fortran `DO` loop is decomposed into groups of iterations. These groups are called *tiles*. The group of tiles that make up a loop nest is called a *tile family*. The loop indices over which tiling is to occur are specified in tile directives. These indices define an space, called an *iteration space*. Tiles in iteration space are mapped onto data in *data space*, which is defined by array bounds.

The role of tiling in a KSR Fortran program may be illustrated by a matrix-matrix multiplication. Consider the following code extract which multiplies a matrix $A \in \mathbf{R}^{n\times p}$ with a matrix $B \in \mathbf{R}^{p\times m}$ using one team of pthreads.

```
C*KSR* PTILE (I,J, TEAMID = TEAM)
       DO I = 1,N
         DO J = 1,M
           DO K = 1,P
               C(I,J) = C(I,J) + A(I,K)*B(K,J)
           END DO
         END DO
       END DO
```

Figure 2.3.2 shows both the iteration space where the tiles are constructed, and the data space where data is stored. In the above code extract, tiling has been done only on the i and j indices and the tiles have been stretched in the k direction (see Figure 2.3.2).

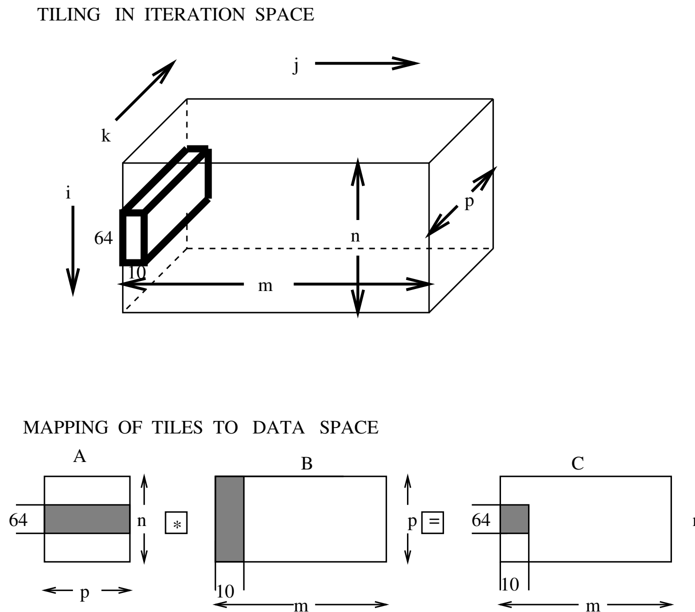

Figure 2.3.2: Iteration space and data space.

This means that within a given tile, execution proceeds sequentially in the k direction. In Figure 2.3.2, the KSR Fortran compilation system uses a tile size of $(64, 10)$.[6] The tile directive `PTILE` causes the iteration space to be partitioned into rectilinear regions (tiles) each of which contains enough loop iterations to create a reasonable amount of work for one processor. Each tile is executed by a pthread, and usually each pthread executes on a different processor[7].

Apart from the tile indices and the team identification number, the user may specify and other parameters in a tile directive. These parameters fall into two categories: The *efficiency related* parameters and the *correctness related* parameters. The former parameters are `TILESIZE` and (tiling) `STRATEGY`. A `TILESIZE` parameter defines a number of loop iterations in each dimension of the tile. There are four distinct tiling strategies: The `SLICE` strategy, `MOD` strategy, `WAVE` strategy, and `GRAB` strategy. The correctness related tiling parameters (`ORDER`, `PRIVATE`, `REDUCTION`, and `LASTVALUE`), are generally specified by the compilation system. The above-mentioned parameters are discussed in detail in

[6] This tile size ensures that a given data item in array C will be used by only one tile, minimizing in this way data contention.

[7] If the number of the pthreads is greater than the number of processors, then the pthreads will not all execute on different processors.

[73].

Tiling is accomplished through the use of a Fortran pre-processor (KSR KAP[8]), the Fortran compiler (f77), and the parallel run-time system (KSR Presto). The user invokes the compiler, which in turn invokes KSR KAP and KSR Presto. KSR KAP inserts tile indices and correctness related tiling parameters where needed. KSR Presto inserts specifications for efficiency related parameters, and the number of pthreads if the team identification number has not been specified. There are three types of tiling:

- Fully automatic tiling.

  Fully automatic tiling allows a user without any knowledge of how to parallelize a program to receive the benefits of automatic parallelization. In fully automatic tiling, KSR KAP tiles the loop nest if possible, inserting a `TILE` directive immediately before the `DO` statement of the outermost loop, and an `END TIlE` statement immediately after the terminal statement of that loop. KSP KAP also inserts the tile indices, and the correctness related parameters where needed. The efficient related parameters are inserted by KSR Presto.

- Semi-automatic tiling.

  Semi-automatic tiling allows the user to influence the efficiency of the KSR Fortran compilation system's tiling decisions, while keeping the same assurance and correctness as with fully automatic tiling. Semi-automatic tiling is recommended for most applications since it offers the user the maximum amount of control over tiling, while still assuring correctness [75]. In semi-automatic tiling the user inserts a `PTILE` directive immediately before or in the loop nest. The placement of `PTILE` affects how KSP KAP tiles the loop and there are certain rules for this placement. The user may specify tile indices and/or efficiency related parameters; otherwise the KSR Fortran compilation system will fill these specifications. KSR KAP analyzes the loop nest and if possible tiles it, replacing the `PTILE` directive with a `TILE` directive. KSP KAP also determines the correctness related parameters.

[8]Kuck Automatic Parallelizer.

- Manual tiling.

  Manual tiling is used when a user does not wish any help from KSR KAP. In manual tiling, the user inserts a `USER`[9] `TILE` directive immediately before the outermost `DO` statement, whose `DO` variable is in the index list, and a `END TILE` directive immediately after the terminal statement of that loop. KSR KAP ignores the loop nest and the directive itself. It does not verify or augment the specified parameters; hence, specification of incorrect parametes causes incorrect execution of the loop nest. The use of manual tiling requires a clear understanding of the tiling procedure.

Our experiences with the these different types of tiling are discussed in Section 4.

2.3.5 Compiling, Running, and Timing a KSR Fortran Program

Suppose that one has a correct Fortran 77 program and wishes to run it on the KSR1, without changing the source code and using more than one processor. If the name of the program is `program.f`, this can be done by issuing the command

```
f77 -kap -para -r8 -O2 program.f -lpresto
```

In the above command, `f77` invokes the Fortran compiler, and the compiler options `-kap`, `-para`, `-r8`, `-O2`, and `-lpresto` perform the following tasks:

- `-kap` invokes KSR KAP as a preprocessing pass.
- `-para` links the program with the libraries necessary to support pthreads and asynchronous Input/Output.
- `-O2` optimizes the object code. `-O2` does global optimization plus automatic loop unrolling.
- `-r8` Makes the default size for both `REAL` and `DOUBLE PRECISION` values be 8 bytes. Otherwise, `REAL` is 8 bytes and `DOUBLE PRECISION` 16 bytes.
- `-lpresto` links the program with the KSR Presto run-time library.

[9] The statement of the keyword USER is optional in manual tiling and is used to indicate that the tile family is user-defined.

If the complilation is successful, KSR KAP produces the following two files:

- `program.cmp`

  This is a transformed source file which contains any directives and restructuring done by KSR KAP. This file is submitted to the compiler.

- `program.out`

  This is a file which reports what KSR could and could not do for this program and also explains why.

The results are obtained by running the executable file `a.out`.

The above described procedure is the simplest way to run any Fortran 77 program on the KSR1. However, it suffers from the following drawbacks:

- KSR KAP does not recognize parallel regions and parallel sections. Therefore, these parallel constructs are not included in the `.cmp` file which is submitted to the compiler.

- KSR KAP parallelizes every loop, including simple small loops for which the startup cost may outweight the gain.

- The teams of pthreads are created dynamically during the execution of the program. This dynamic creation has its own cost, which influences the performance of the code.

- The user is unaware of the number of processors being used, and therefore performance measurements cannot be done.

The tiling of simple, small loops can be avoided if instead of `-kap`, one uses the compiler option `-kap="-noautotile"` and semi-automatic tiling (`PTILE`). In this case, KSR KAP tries to tile only the loops that correspond to a `PTILE` directive, ignoring the rest. The dynamic creation of the teams of pthreads can be avoided using the KSR Presto routine `IPR_CREATE_TEAM` discussed in Subsection 2.3.1. This is recommended,

especially when each parallel directive uses the same numbers of pthreads, as happens in our implementation.

Suppose now that we want to run a program on 16 processors using one team of 16 pthreads[10]. We assume that the following function

```
FUNCTION LIB_NUMBER_OF_PROCESSORS()
CHARACTER*20 ENV

CALL GETENV('PL_NUM_THREADS',ENV)
READ(ENV,*) NUMTHREADS
LIB_NUMBER_OF_PROCESSORS = NUMTHREADS

END
```

is an external function of the program. We also assume that before running the program the following command

```
setenv PL_NUM_THREADS 16
```

has been typed on the command line, and that before the parallel parts of the program there are the the following statements

```
NPROCS = LIB_NUMBER_OF_PROCESSORS()
CALL IPR_CREATE_TEAM(NPROCS,TEAM)
```

When the execution reaches the first statement, the variable `NPROCS` (the number of processors) is being set equal to the environmental variable `PL_NUM_THREADS` (the number of pthreads) which is known to be equal to 16. Then, `IPR_CREATE_TEAM` assembles from the idle pool the team members and determines the identification number of the team, `TEAM`. Specifying this idendification number in every directive of the parallel constructs, all the parallel constructs are executed on 16 processors by a team of 16 pthreads. Finally, it is recommended to use the following command

[10]Typically the number of pthreads is equal to the number of processors, and each pthread corresponds to one processor.

```
allocate_cells -A 16 a.out
```

for running this parallel program instead of simply `a.out`. `allocate_cells` partitions off the requested number of cells (16 for this example) for a parallel program, avoiding certain loaded cells (`-A`), and deallocates them after the execution.

A KSR Fortran program may be timed from inside[11] using facilities provided in the `libtimer.a` library. This can be done by linking the program with the `libtimer.a` library, and setting up a timer calling the routine `EQTIMER` from inside the program. The specific part of the code which is to be timed, is included between a call of the routine `ETON` and a call of the routine `ETOFF`. The routine `PRINTET` prints the timer's information. The following example demonstrates how to measure the time needed for a matrix-matrix multiplication.

```
      CALL EQTIMER(1,' Matrix-matrix multiplication ')
      CALL ETON(1)

C*KSR* PTILE (I,J, TEAMID = TEAM)
       DO  I = 1,N
         DO  J = 1,M
           DO  K = 1,P
             C(I,J) = C(I,J) + A(I,K)*B(K,J)
           END DO
          END DO
       END DO

      CALL ETOFF(1)
      CALL PRINTET()
```

There can be more than one timers inside a KSR Fortran program, each of them with a unique identification number. This number is the argument of `ETON` and `ETOFF`. Multiple

[11] As in any Unix system we can use the command `time` to time a program from the command line.

calls over the same timer accumulate the timings. We are not aware of any way to save the elapsed time into a variable.

2.3.6 Language Extensions

KSR Fortran is an extended version of Fortran 77. Among the most interesting language extensions, from the numerical analysis point of view, are the following:

- Additional data types.

  KSR Fortran supports a set of additional data types, as for example `INTEGER*1`, `LOGICAL*8`, `REAL*8`, and `COMPLEX*16`. The positive integer after the data type indicates the number of bytes needed for its representation.

- Allocatable arrays.

  An *allocatable array* is an array whose space can be allocated and deallocated dynamically during program execution. Allocatable arrays are a standard feature of Fortran 90 [90]. However, the syntax of statements concerning allocatable arrays in KSR Fortran is not the standard Fortran 90 syntax. A program can allocate an allocatable array to the program heap or to the program stack. An allocatable array is declared as an ordinary Fortran 77 array, but without specifying its bounds and using the colon notation. (For example `REAL A(:,:)`.) When a program begins execution, its allocatable arrays are considered not definable and no space is allocated for them. To allocate space for an allocatable array, the program executes an `ALLOCATE` statement, which names the array to be allocated and specifies its bounds. For example, a 100-by-100 array A is allocated to the program heap by the following statement

  ```
  ALLOCATE (A(100,100), STAT = IERR_ALLOC)
  ```

  where `IERR_ALLOC` is the statement's return status[12]. The bounds in an `ALLOCATE`

[12] If the statement is successful, `IERR_ALLOC` is set to zero. If there is inadequate memory to meet the request, `IERR_ALLOC` is set to a positive integer.

statement may also be expressions (for example `2*N,2*N` instead of `100,100`). After an `ALLOCATE` statement, the named arrays are considered currently allocated. A currently allocated array becomes not currently allocated when the program executes a `DEALLOCATE` statement for that array. For example the following statement

```
DEALLOCATE (A, STAT = IERR_DEALLOC)
```

where `IERR_DEALLOC` is the statement's return status, deallocates the above two dimensional array A. However, A can subsequently be reallocated with an `ALLOCATE` statement.

If a currently allocated array is not deallocated, it becomes undefined when the subprogram in which it was allocated returns with a `RETURN` or `END` statement. An undefined allocatable array should not be referenced, defined, reallocated, or deallocated. It is recommended to deallocate an allocatable array before the `RETURN` or `END` statements [73].

For faster execution, a program can allocate an allocatable array to the current program unit's stack. The corresponding statement for the above mentioned array A may be

```
ALLOCATE STACK(A(100,100), STAT = IERR_ALLOC)
```

Arrays allocated to the stack are unconditionally deallocated when the subprogram returns. The advantages of using allocatable arrays in KSR Fortan programs are discussed in Section 4.

- Array Syntax.

  A KSR Fortran program can reference multiple elements of an array in a single assignment statement. The reference can be to all elements of the array or to a subset of elements. In the former case, we have a reference to a *full array*, and in the latter to an *array section*. KSR Fortran supports an array syntax through the KSR KAP preprocessor, which converts the array syntax to equivalent tiled `DO`

loops. For example the statement `A = B`, which copies a 100-by-100 real matrix B into A, will be converted by KSR KAP into

```
C*KSR* TILE ( II2,II1 )
       DO 2 II2=1,100
         DO 2 II1=1,100
           A(II1,II2) = B(II1,II2)
    2      CONTINUE
C*KSR* END TILE
```

A program references a full array by naming the array without subscripts. For example, the statement `A = 5.*B` will multiply all entries of `B` by 5. The notation for array sections is similar to the MATLAB and Fortran 90 notation. For example, if A is a 10-by-10 matrix, the statement `B = A(:,1:3)` will copy the first three columns of A into the 10-by-3 matrix B.

The use of array syntax saves programmer's time from writing trivial loops, improves the readability of the source code, and it is a very helpful facility in writing codes that deal with matrix blocks.

- Additional control statements.

  The most interesting additional control statement provided in KSR Fortran is the `DO WHILE` statement. `DO WHILE` executes a set of statements while a logical expression is true. The syntax of the `DO WHILE` statement is

  `DO` *label* `WHILE` *expression*

  The parameter *label* in the above syntax identifies the line number of the terminal statement of the `DO WHILE` loop. Another additional control statement is the `END DO` statement, which terminates a `DO`, or `DO WHILE LOOP`, and is an alternative to using labels.

- Additional intrinsic and external procedures.

KSR Fortran supports an extended set of intrinsic and external procedures, in addition to the basic set of Fortran 77 intrinsic functions. More information on these procedures can be found in [73].

2.3.7 Numerical Software

Kendall Square Research supplies the KSRlib/BLAS library [77], which contains the highly optimized for the KSR1 level 3 BLAS routines `SGEMM` and `CGEMM`. These routines can be called either with the standard calling interface or with a slightly different interface for higher performance. In a non-standard call of `SGEMM` the user specifies four additional parameters. These include the number of processors and the team identification number. A correct specification of the additional parameters may improve the performance of `SGEMM` significantly, giving an almost linear speedup. `SGEMM` is the fastest matrix multiply on the KSR1 known to us. If these additional parameters are not supplied then there is no parallelism within `SGEMM`. At the time of writing, the other routines in KSRlib/BLAS library do not appear to be highly optimized.

The Kendall Square Research also supplies the KSRlib/LAPACK library [78]. When this work was begun this library was not available to us, so we used the standard LAPACK distribution. A few months later, when we gained access to the KSRlib/LAPACK library, we reran our programs, linking them with the the KSRlib/LAPACK library. We observed that the timing results were almost identical. However, the performace of the LAPACK routines may be improved significantly if we modify their source code. We experimented with the block size in some LAPACK routines (the block size is set in the environmental routine `ILAENV`) and we found that a block size of 16 gives the best all round performance on the KSR1. We also changed all the calls to `SGEMM`, specifying the four additional parameters. In Chapter 4 and Chapter 5, we discuss the modifications that we did in particular LAPACK routines in detail.

The NAG Fortran Library Mark 15 [95] has also been implemented on the KSR1 at the Centre of Novel Computing at the University of Manchester. All the parallelism in this implementation is within the KSRlib/BLAS library.

2.4 Experiences and Remarks

We worked on the KSR1 from the September of 1992 until the September of 1993. We started by porting codes written in standard Fortran 77 and asking the KSR1 to parallelize them automatically. The results were disappointing. Looking at the `.out` files, it was clear that the main reason for this was that KSR KAP tiled every loop possible, including the simple and small loops for which the startup cost outweighs the gain. We also noticed that some other loops had not been tiled because of the way they were written. And of course, parts of the codes that could have been written as parallel regions and parallel sections had been ignored by KSR KAP. The result was really not very surprising. It is known that there is a long way to go before one can give a program written in standard Fortran 77 to a parallel computer, and get it to run with a high degree of parallelism [114]. Thus, we always modified the source codes when we we ported existing Fortran 77 codes to the KSR1.

The use of parallel regions and parallel sections requires only an understanding of the purpose of each of these parallel constracts. Correct implementation of parallel regions and parallel sections can give almost linear speedup.

In our first codes that we wrote from scratch we used semi-automatic tiling. Later we experimented using manual tiling and in some instances we achieved slightly better timing results.

In Section 2 we mentioned that each processor contains a 256 kilobytes data cache and a 256 kilobytes instruction cache. The 256 kilobytes data cache may be perceived as in Figure 2.4.1. Suppose now that we want to add two 128-by-128 real matrices A and B. If one declares the matrices A and B as $A(128, 128)$ and $B(128, 128)$, then the length of these arrays will be 128 kilobytes. This will cause contention and cache misses since both arrays will compete to map to the same set. But, if one declares the arrays A and B as $A(128, 130)$ and $B(128, 130)$, then a pad of 2 kilobytes in the SVA will be created between the two arrays, and the above mentioned problem will be avoided. For the same reason, when we deal with large matrices whose dimension n is a power of 2, we declare them as n-by-$n + 2$. We found that this improves the timing results significantly.

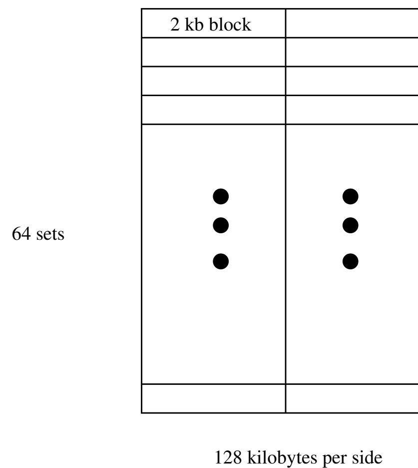

Figure 2.4.1: The 256 kilobytes data cache inside a subcache.

A drawback of the KSR1 observed during this project was the slowness of compilation. The compilation of a KSR Fortran program takes a significant amount of time. In some instances, the compilation of the same code needed an order of magnitude more time on the KSR1 than it needed on a SUN SPARC workstation. It is therefore a good practice to create libraries with the object codes of the external subroutines and functions of a program, and to link them with the program. In many instances during our research, we had to measure the performance of a code for matrices of various order. Before "discovering" allocatable arrays, we did a separate compilation for each dimension. Using allocatable arrays, the recompilation of the same code may be avoided. The size of the matrices may be given from the command line[13], or during the execution using a `READ` statement. Finally, we mention that on several occasions the system suddenly broke down and it needed rebooting to get working again. However, this is not necessarily due to a hardware error but it could just be incomponent programming.

[13]As in C programming language.

Chapter 3

Two-sided Procrustes-type problems

3.1 Introduction

In 1962, Hurley and Cattell [69] coined the term *Procrustes*[1] *problem* to denote a least squares problem where the independent variable is a matrix. In particular, they called the problem of approximating a given matrix A by rotating another given matrix B, that is

$$\min_{V^T V = I_n} \|A - BV\|_F, \quad A, B \in \mathbf{R}^{m \times n},$$

where $\|\cdot\|_F$ denotes the Frobenius norm, an *orthogonal Procrustes problem.* The problem arises frequently in applications[2], and a solution is $V = U$ where U is the orthogonal factor in the polar decomposition $B^T A = UH$ [61]. This result holds for the Frobenius norm but it does not hold for all unitarily invariant norms. In [88], Mathias shows by a counter-example that the orthogonal Procrustes problem is not solved by the polar decomposition for the trace norm, which is equal to the sum of the singular values of a matrix.

In 1968, Schönemann [107] generalized this problem to its two-sided form: given

[1] *Procrustes* was a robber in Hellenic (Greek) mythology. He was in the habit of putting his victims in a bed, whether they fitted in this bed or not. If they were too long for it, he performed radical surgery on their legs to fit them exactly on it. If there were too short, he streched their limbs so that they became the right length. The Athenian hero *Theseus* fitted *Procrustes* to his own bed as *Procrustes* had fitted others, freeing the road to Athens from South Greece.

[2] Applications that require the solution of an orthogonal Procrustes problem are discussed in Section 8 in Chapter 4.

two square matrices A and B, find two orthogonal matrices U and V such that UBV approximates A in a least squares sense. Starting with the singular value decompositions

$$A = U_A \Sigma_A V_A^T, \quad B = U_B \Sigma_B V_B^T, \tag{3.1.1}$$

Schönemann proved that the least squares fit to A is $U_A \Sigma_B V_A^T$, corresponding to $V = V_B V_A^T$ and $U = U_A U_B^T$.

In this chapter, we consider some possible forms of the above two-sided orthogonal Procrustes problem that arise when A and B are not necessarily square matrices, and U and V are required to have other special properties. For example, U and V may be arbitrary, symmetric, orthogonal, orthogonal with determinant $+1$, permutation matrices or some combination of them. Unless otherwise stated, all matrices are assumed to be real.

3.2 The Most General Problem

The most general problem, which will be referred to as the *general two-sided Procrustes problem* can be stated as follows:

THE GENERAL TWO-SIDED PROCRUSTES PROBLEM (GTSPP)

Given A and $B \in \mathbf{R}^{m\times n}$ $(m \geq n)$, find two arbitrary matrices $X \in \mathbf{R}^{m\times m}$ and $Y \in \mathbf{R}^{n\times n}$ to minimize

$$\|A - XBY\|_F. \tag{3.2.1}$$

The problem can be solved as follows. If $A = U_A \Sigma_A V_A^T$ and $B = U_B \Sigma_B V_B^T$ are the singular value decompositions of A and B, then

$$\begin{aligned} \|A - XBY\|_F &= \|U_A \Sigma_A V_A^T - X U_B \Sigma_B V_B^T Y\|_F \\ &= \|\Sigma_A - \underbrace{U_A^T X U_B}_{\hat{X}} \Sigma_B \underbrace{V_B^T Y V_A}_{\hat{Y}}\|_F \end{aligned}$$

$$
\begin{aligned}
&= \|\Sigma_A - \underbrace{\hat{X}\Sigma_B\hat{Y}}_{\Psi}\|_F \\
&= \|\Sigma_A - \Psi\|_F,
\end{aligned} \tag{3.2.2}
$$

and the problem is equivalent to finding the m-by-n matrix Ψ that minimizes (3.2.2). It is apparent that if $\text{rank}(A) \leq \text{rank}(B)$, (3.2.2) is equal to zero when $\Psi = \Sigma_A$. If $\text{rank}(A) > \text{rank}(B) = r$, then the minimizing Ψ is

$$\Psi = \text{diag}(\sigma_1(A), \ldots, \sigma_r(A), 0, \ldots, 0).$$

This follows from the following well-known rank r approximation result: If $W \in \mathbf{R}^{m\times n}$, $m \geq n$, has the singular value decomposition $W = U_W \Sigma_W V_W^T$, then

$$\min_{\text{rank}(\Omega)=r} \|W - \Omega\| = \left[\sum_{i=r+1}^{n} \sigma_i \, [W]^2\right]^{1/2}, \tag{3.2.3}$$

and the minimum in (3.2.3) is achieved for the matrix

$$W' = U_W \text{diag}(\sigma_1(A), \ldots, \sigma_r(A), 0, \ldots, 0) V_W^T.$$

(In fact, W' is the minimizer for any unitarily invariant norm, as shown in [67, pages 449-450]).

To summarise, writing $r_B = \text{rank}(B)$,

$$\min_{X,Y \text{arbitrary}} \|A - XBY\|_F = \begin{cases} 0 & \text{if } \text{rank}(A) \leq r_B \\ \sum_{i=r_B+1}^{n} (\sigma_i(A)^2)^{1/2} & \text{otherwise.} \end{cases} \tag{3.2.4}$$

Since we did not impose any restriction on X and Y, (3.2.4) indicates the lowest value of any two-sided Procrustes problem for given A and B.

If we set $\hat{Y} = I_n$ and $\hat{X} = \Psi\Sigma_B^+$, where Σ_B^+ is the pseudo-inverse of Σ_B, then $X = U_A \hat{X} U_B^T$ and $Y = V_B V_A^T$ is a solution of the GTSPP. Since Y is an orthogonal matrix, the solution of the following two-sided Procrustes problem

$$\min_{X, V^T V = I_n} \|A - XBV\|_F \tag{3.2.5}$$

has been also obtained, but the problem (3.2.5) can be solved independently as follows.

Assuming that the orthogonal matrix V is known, the matrix X that minimizes $\|A - XBV\|_F$ is given by

$$X = A(BV)^+ = AV^T B^+.$$

Writing the singular value decompositions of A and B^+B as $U_A\Sigma_A V_A^T$ and $W\Lambda W^T$ respectively, where $\Lambda = \mathrm{diag}(\lambda_i)$ and $\lambda_i = 0$ or 1, we have

$$\begin{aligned}\|A - XBV\|_F &= \|U_A\Sigma_A V_A^T - U_A\Sigma_A V_A^T V^T B^+ BV\|_F \\ &= \|\Sigma_A - \Sigma_A \underbrace{(V_A^T V^T W)}_{Z} \Lambda \underbrace{(W^T V V_A)}_{Z^T} \|_F.\end{aligned}$$

The matrix Z is an orthogonal matrix and setting $Z = I_n$ we obtain the solution of (3.2.5), that is $V = WV_A^T$ and $X = A(WV_A^T)^T B^+$. For the above selection of X and V,

$$\|A - XBV\|_F = \begin{cases} 0, & \text{if } \mathrm{rank}(A) \le r_B \\ \sum_{i=r_B+1}^{n} (\sigma_i(A)^2)^{1/2}, & \text{otherwise,} \end{cases}$$

where r_B is the rank of B.

3.3 The Two-Sided Orthogonal Procrustes Problem

The *two-sided orthogonal Procrustes problem* is a generalization of Schönemann's problem mentioned in the introduction of this chapter. Here, the matrices A and B are rectangular instead of square, and the problem can be described as follows.

The Two-Sided Orthogonal Procrustes Problem (TSOPP)

Given A and $B \in \mathbf{R}^{m\times n}$, find two orthogonal matrices $U \in \mathbf{R}^{m\times m}$ and $V \in \mathbf{R}^{n\times n}$ to minimize

$$\|A - UBV\|_F. \tag{3.3.1}$$

The analysis of this problem will be based on the following two theorems, both taken from [67, pages 432–433][3]

Theorem 3.3.1 *Let $A \in \mathbf{R}^{n\times n}$ have the singular value decomposition $A = V\Sigma W^T$. Then (a) the problem*

[3] The theorems in [67] deal with complex matrices.

$$\max\{\text{tr } AU : U \in \mathbf{R}^{n\times n} \textit{ is orthogonal}\}$$

has the solution $U = WV^T$*, and the value of maximum is* $\sigma_1(A) + \cdots + \sigma_n(A)$*, where* $\{\sigma_i(A)\}$ *is the set of singular values of* A*. (b) There exists an orthogonal matrix* $U \in \mathbf{R}^{n\times n}$ *such that* $AU \in \mathbf{R}^{n\times n}$ *is a symmetric positive semidefinite matrix. An orthogonal matrix* U *is a maximizing matrix for the problem in (a) if and only if* AU *is positive semidefinite; a maximizing* U *is uniquely determined if* A *is nonsingular. The eigenvalues of* AU *are the singular values of* A*.*

Theorem 3.3.2 *Let* $A \in \mathbf{R}^{m\times n}$*,* $B \in \mathbf{R}^{n\times m}$*, and* $q = \min\{m, n\}$*. Let* $\sigma_1(A), \ldots, \sigma_q(A)$*, and* $\sigma_1(B), \ldots, \sigma_q(B)$*, be the singular values of* A *and* B *respectively, arranged in nonincreasing order. If both* $AB \in \mathbf{R}^{m\times m}$ *and* $BA \in \mathbf{R}^{n\times n}$ *are positive semidefinite, then there exists a permutation* τ *of the integers* $1, 2, \ldots, q$ *such that*

$$\text{tr}\, AB = \text{tr}\, BA = \sum_{i=1}^{q} \sigma_i(A)\sigma_{\tau(i)}(B). \tag{3.3.2}$$

The functional $(\cdot,\cdot) : \mathbf{R}^{m\times n} \times \mathbf{R}^{m\times n} \to \mathbf{R}$, where

$$(M, N) \equiv \text{tr}\, MN^T,$$

defines an inner product on the vector space $\mathbf{R}^{m\times n} \times \mathbf{R}^{m\times n}$ and we can write

$$\|A - UBV\|_F^2 = (A - UBV, A - UBV) = \|A\|_F^2 - 2(A, UBV) + \|B\|_F^2,$$

since $\|A\|_F^2 = \text{tr}\, AA^T = (A, A)$. Thus to minimize (3.3.1) we must find orthogonal matrices $U \in \mathbf{R}^{m\times m}$ and $V \in \mathbf{R}^{n\times n}$ that maximize

$$(A, UBV) = \text{tr}\, AV^T B^T U^T.$$

Since the sets of orthogonal matrices in $\mathbf{R}^{m\times m}$ and $\mathbf{R}^{n\times n}$ are compact, their Cartesian product is compact, and so there exist orthogonal matrices U_0, V_0 that maximize $\text{tr}\, AV^T B^T U^T$. These maximizing matrices have the property that

$$\text{tr}(AV_0^T B^T)U_0^T \geq \text{tr}(AV_0^T B^T)U$$

for any orthogonal matrix $U \in \mathbf{R}^{m\times m}$, and hence by Theorem 3.3 we know that $AV_0^TB^TU_0^T$ is positive semidefinite. Similarly,

$$\operatorname{tr}(AV_0^TB^T)U_0^T = \operatorname{tr}(B^TU_0^TA)V_0^T \geq \operatorname{tr} B^TU_0^TAV$$

for any orthogonal matrix $V \in \mathbf{R}^{n\times n}$, and so $B^TU_0^TAV_0^T$ is positive semidefinite. Thus the two matrices AV_0^T and $B^TU_0^T$ satisfy the hypotheses of Theorem 3.3 and if $q = \min\{m, n\}$ then

$$\begin{aligned}\max_{U^TU=I,V^TV=I} \operatorname{tr} AV^TB^TU^T &= \operatorname{tr} AV_0^TB^TU_0^T \\ &= \sum_{i=1}^{q} \sigma_i(AV_0^T)\sigma_{\tau(i)}(B^TU_0^T) \\ &= \sum_{i=1}^{q} \sigma_i(A)\sigma_{\tau(i)}(B)\end{aligned}$$

for some permutation τ of the integers $1, \ldots, q$, since the singular values are orthogonally invariant. Horn and Johnson show in [67, page 436] that the maximum value of the sum is achieved for the identity permutation τ, and we can conclude that

$$\max_{U^TU=I,V^TV=I} \operatorname{tr} AV^TB^TU^T = \sum_{i=1}^{q} \sigma_i(A)\sigma_i(B),$$

where the singular values of A and B are both arranged in decreasing order.

Using this result for the TSOPP, we find for $A, B \in \mathbf{R}^{m\times n}$ and $q = \min\{m, n\}$ that

$$\begin{aligned}\min_{U^TU=I,V^TV=I} \|A - UBV\|_F &= \left[\|A\|_F^2 - 2\sum_{i=1}^{q} \sigma_i(A)\sigma_i(B) + \|B\|_F^2\right]^{1/2} \\ &= \left[\sum_{i=1}^{q} \sigma_i^2(A) - 2\sum_{i=1}^{q} \sigma_i(A)\sigma_i(B) + \sum_{i=1}^{q} \sigma_i^2(B)\right]^{1/2} \\ &= \left[\sum_{i=1}^{q} [\sigma_i(A) - \sigma_i(B)]^2\right]^{1/2}. \end{aligned} \tag{3.3.3}$$

We consider now the singular value decompositions of A and B

$$A = U_A\Sigma_AV_A^T \quad \text{and} \quad B = U_B\Sigma_BV_B^T.$$

Since the Frobenius norm is orthogonally invariant and

$$\|\Sigma_A - \Sigma_B\|_F = \left[\sum_{i=1}^{q} [\sigma_i(A) - \sigma_i(B)]^2\right]^{1/2},$$

we can write

$$\begin{aligned}\|\Sigma_A - \Sigma_B\|_F &= \|U_A(\Sigma_A - \Sigma_B)V_A^T\|_F \\ &= \|U_A\Sigma_A V_A^T - U_A(U_B^T U_B)\Sigma_B(V_B^T V_B)V_A^T\|_F \\ &= \|A - (U_A U_B^T)B(V_B V_A^T)\|_F, \end{aligned} \tag{3.3.4}$$

and thus from (3.3.3) minimizing matrices U and V are $U_A U_B^T$ and $V_B V_A^T$, respectively.

Provided parallel changes are made in the orthogonal matrices, the singular value decompositions are invariant. Thus the minimizing orthogonal matrices $U = U_A U_B^T$ and $V = V_B V_A^T$ are certainly not unique.

3.4 The Two-sided Rotation Procrustes Problem

A rotation matrix is an orthogonal matrix with determinant +1. The *two-sided rotation Procrustes problem* can be stated as follows:

> THE TWO-SIDED ROTATION PROCRUSTES PROBLEM (TSRPP)
> Given A and $B \in \mathbf{R}^{m\times n}$ $(m \geq n)$, find two rotation matrices U and V, m-by-m and n-by-n respectively, to minimize
>
> $$\|A - UBV\|_F. \tag{3.4.1}$$

The problem of finding only the left rotation matrix in (3.4.1), that is

$$\min_{U^T U = I_m, \det(U)=+1} \|A - UB\|_F, \tag{3.4.2}$$

arises in the estimation of the attitude of a satellite [56, 121]. This application is discussed in Section 8 in Chapter 4.

Assuming that the minimizing rotation matrix V in (3.4.1) is fixed, and setting $M = BV$, the minimizing rotation matrix U can be found as follows. Since

$$\|A - UM\|_F^2 = \operatorname{tr} A^T A - 2\operatorname{tr} MA^T U + \operatorname{tr} M^T M,$$

the problem is equivalent to finding the rotation matrix U that maximizes $\operatorname{tr} MA^TU$. If the polar decomposition of MA^T is $MA^T = ZP$, where Z is orthogonal and P is symmetric positive semidefinite (Z is uniquely defined and P is positive definite if MA^T is nonsingular), and if NDN^T is the spectral decomposition of P with the diagonal elements of D arranged in decreasing order, then

$$\begin{aligned}
\operatorname{tr}(MA^TU) &= \operatorname{tr}(ZPU) \\
&= \operatorname{tr}(ZNDN^TU) \\
&= \operatorname{tr}(\underbrace{N^TUZN}_{X}D) \\
&= \operatorname{tr}(XD) \\
&= \sum_{i=1}^{m} d_i x_{ii}. \qquad (3.4.3)
\end{aligned}$$

The above sum is a linear function of the nonnegative numbers $d_1, \ldots, d_m$, and its maximum is attained when the diagonal elements of X attain their maximum values. Because X is an orthogonal matrix, all entries of X are between -1 and 1, so with no restrictions on $\det(U)$ (3.4.3) is maximized when $x_{ii} = 1$, $x_{ij} = 0$, $i \neq j$. Because $\det(U)$ is required to be $+1$,

$$\det(X) = \det(N^TUZN) = \det(N)^2 \det(U) \det(Z) = \det(Z).$$

If $\det(Z) = -1$, then $\det(X)$ must be -1 as well, and since $d_1 \geq \ldots \geq d_m$,

$$X_0 = \begin{bmatrix} I_{m-1} & 0 \\ 0 & -1 \end{bmatrix}$$

is a solution. Obviously $X_0 = I_m$ is a solution when $\det(Z) = +1$. Thus the rotation matrix U that minimize (3.4.1) is

$$U = NX_0N^TZ^T.$$

Similarly, if we assume that U is fixed and set $M' = UB$, the minimizing rotation matrix V is given by

$$V = N'X_0'N'^TZ'^T,$$

where $Z'P'$ is the polar decomposition of $A^T M'$, $P' = N'D'N'^T$ is the spectral decomposition of P', and

$$X'_0 = \begin{cases} \begin{bmatrix} I_{n-1} & 0 \\ 0 & -1 \end{bmatrix}, & \text{if } \det(Z') = -1, \\ \\ I_n, & \text{otherwise.} \end{cases}$$

The above solutions of the left and right one-sided rotation Procrustes problems suggest the following iterative procedure for the solution of the TSRPP. First set $V = I_n$ and determine U as the solution of

$$\min_{U^T U = I_m, \det(U) = +1} \|A - U(BV)\|_F,$$

then fix U and determine V as the solution of

$$\min_{V^T V = I_n, \det(V) = +1} \|A - (UB)V\|_F,$$

repeating the process until it converges. Each step of the iteration will reduce (or not change) the value of the norm and so convergence, not necessarily to the global minimum, is assured.

This approach of repeatedly fixing one matrix and minimizing over the other has been referred to as a *flip-flop* algorithm by Van Loan. This approach has the advantage that it can utilize the known solutions of the one-sided Procrustes problems for the solution of the corresponding two-sided ones. On the other hand, a flip-flop algorithm does not guarantee convergence to a global minimum. However, in our numerical experiments this approach almost always yielded global minimizing matrices to the GTSPP and the TSOPP, where to global minimum is known. This observation encourages us to use flip-flop algorithms for the investigation of two-sided Procrustes problems where an analytical expression of the minimizing matrices cannot be found.

3.5 Permutation Problems

Another two-sided Procrustes problem of interest is the following:

THE TWO-SIDED PERMUTATION-ORTHOGONAL

PROCRUSTES PROBLEM (TSPOPP)

Given A and $B \in \mathbf{R}^{m\times n}$, $m \geq n$, find an m-by-m permutation matrix P and an orthogonal matrix $V \in \mathbf{R}^{n\times n}$ to minimize

$$\|A - PBV\|_F. \tag{3.5.1}$$

When $V \equiv I_n$, the above problem takes a simple form that occurs in the context of multidimensional scaling [48]. Here, A and B may refer to the same (or similar) objects but in different orders. By minimizing $\|A - PB\|_F$, subject to P being a permutation matrix, one can test whether there is some similarity between the samples. If the samples consist of the same objects in different orders, $\|A - PB\|_F$ will be zero modulo roundoff for the optimal permutation matrix P.

Since

$$\|A - PB\|_F^2 = \operatorname{tr} A^T A + \operatorname{tr} B^T B - 2 \operatorname{tr} PBA^T,$$

minimizing $\|A - PB\|_F$ is equivalent to maximizing $\operatorname{tr} PBA^T$. The scalar $\operatorname{tr} PBA^T$ is a linear function of the elements of P. Since maximizing this function is the key for further investigation of Procrustes-type permutation problems, we discuss this independently.

3.5.1 Maximizing the Trace of PA

Given $A \in \mathbf{R}^{n\times n}$, we wish to find the n-by-n permutation matrix P that maximizes the trace of PA. The problem can be posed as a linear programming problem, as we now explain. If instead of a permutation matrix, P was required to be a doubly-stochastic matrix, that is matrix of non-negative elements whose rows and columns sum to unity,

the problem could be stated as

$$\text{maximize} \sum_{i=1}^{n} \sum_{j=1}^{n} p_{ij} a_{ji} \tag{3.5.2}$$

subject to

$$\sum_{j=1}^{n} p_{ij} = 1, \quad i = 1, \ldots, n,$$
$$\sum_{i=1}^{n} p_{ij} = 1, \quad j = 1, \ldots, n,$$
$$p_{ij} \geq 0, \quad i, j = 1, \ldots, n.$$

The maximum must occur at a vertex of the feasible region bounded by the constraints and this vertex corresponds to a permutation matrix, a special case of a doubly stochastic matrix. This fact is explained by the theory of *total unimodularity* in [97].

The problem can be solved by the *simplex method*, but first it must be written as a linear programming problem in standard form, that is

$$\begin{array}{ll} \text{minimize} & c^T x \\ \text{subject to} & Mx = b, \; x \geq 0. \end{array}$$

Setting $c = \text{vec}(A)$ and $x = \text{vec}(P^T)$, where $\text{vec}(A) \in \mathbf{R}^{n^2}$ denotes the columns of A strung out into one long vector, we observe that the objective function (3.5.2) can be written as $c^T x$ and the problem is equivalent to minimizing $-c^T x$ subject to $Mx = b$,

where M is the $2n$-by-n^2 matrix

$$M = \begin{bmatrix} \underbrace{1,\ldots,1}_{n} & 0,\ldots,0 & \ldots & 0,\ldots,0 \\ 0,\ldots,0 & \underbrace{1,\ldots,1}_{n} & \ldots & 0,\ldots,0 \\ \ldots & \ldots & \ldots & \ldots \\ 0,\ldots,0 & 0,\ldots,0 & \ldots & \underbrace{1,\ldots,1}_{n} \\ & & & \\ I_n & I_n & \ldots & I_n \\ & & & \end{bmatrix},$$

and b is the $2n$-vector $(1,\ldots,1)^T$.

The solution of the problem, and therefore the permutation matrix P, can now be easily obtained as a straightforward application of the simplex method. The following MATLAB function solves the problem of maximizing the trace of PA, where A is any square real matrix and P is the required permutation matrix of the same dimensions. The function uses the MATLAB function `simplex.m` written by Philip Gill of the University of California at San Diego.

```
function P = maxtrace(A)


% This function evaluates the permutation
% matrix P that maximizes trace(PA).
% A is a given square real matrix.


[m,n] = size(A);


if m ~= n
 error( ' Matrix A must be a square matrix ' )
end
```

```
% Construction of the matrix M.
M = zeros(2*n,n*n);

for i=1:n
  for j = 1 + (i-1)*n:i*n
      M(i,j) = 1;
  end
end

for i=0:n:(n-1)*n
    M(n+1:2*n,i+1:i+n) = eye(n);
end

% Construction of the vector b.
b = ones(2*n,1);

% Construction of the vector c.
c = -A(:);

% Solution of the linear programming problem.
x = simplex(M,b,c);

% Construction of the required permutation matrix P
% from the vector x given by the simplex method.
P = zeros(n);
P(:) = x;
P = P';
```

Observing the special structure of M, one may ask if there is a simpler way to solve

this problem than by applying the simplex method, which does not exploit the structure of M. The key to the answer is the similarity of the permutation problem with the *assignement problem* which is a special case of the *transportation problem.* Both problems arise frequently in operational research [24, 115].

Informally, the *assignment problem* can be described as follows. Assuming that the numerical scores are available for the performance of each of n persons on each of n jobs, the *assignment problem* is the quest for an assignment of persons to jobs so that the sum of n scores so obtained is as large as possible. The close relation between the problem of finding the permutation matrix P which maximizes the trace of PA, and the *assignment problem* is clarified by the following example.

Suppose that 3 individuals are available for 3 jobs. A rating matrix $R = (r_{ij})$ is a square matrix which indicates the performance of the ith individual on the jth job in some units. Let R be

$$R = \begin{bmatrix} 2 & 5 & 1 \\ 3 & 1 & 8 \\ 4 & 1 & 2 \end{bmatrix}.$$

The permutation matrix P that maximizes the trace of PR is

$$P = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix},$$

which gives also the solution to the *assignment problem.* According to P, reading P columnwise, the best total performance of this group would be achieved if the first individual did the second job, the second individual the third, and the third individual the first.

In 1955, Kuhn [82] introduced a method of solving the *assignment prolem* based on the ideas of two Hungarian mathematicians, J. Egerváry [38] and König [81], and termed his method *the Hungarian method for the assignment problem.* In his analysis, the rating matrix consists of positive integers. Since then, several other techniques for solving the problem have been developed. Among the people who have made significant contributions are Dwyer [36], Flood [41], Votaw and Orden [120]. In 1982, Papadimitriou

Figure 3.5.1: A bipartite graph.

and Steiglitz [97], exploiting advantages given by graph theory, presented a new version of the Hungarian algorithm. Their work is based on Kuhn's ideas but but they follow a different approach to the solution of the *assignment problem* using exclusively terms and tools from graph theory.

Papadimitriou and Steiglitz consider two sets of n vertices, $V = \{V_1, \ldots, V_n\}$, and $U = \{U_1, \ldots, U_n\}$. The V_i vertex stands for the ith individual and the U_j vertex for the jth job. They also consider the set of the edges $E = \{E_{ij},\ i = 1, \ldots, n,\ j = 1, \ldots, n\}$, where E_{ij} is the numerical score of the ith individual on jth job. They write the *assignment problem* as a *bipartite graph*. A bipartite graph is a graph $B = (W, E)$ that has the following property. The set of vertices W can be partitioned into two sets, V and U, and each edge in E has one vertex in V and one in U. Thus the above example can be written as in Figure 3.5.1.

A *matching* M of a graph $G = (W, E)$ is a subset of the edges with the property that no two edges of M share the same vertex. In the above example, $M_1 = \{E_{13}, E_{22}\}$ and $M_2 = \{E_{12}, E_{23}, E_{31}\}$ are matchings, and M_2 is a *maximum* matching since a matching of G obviously can never have more than $6/2 = 3$ edges. Given a graph, the *matching problem* is to find a maximal matching. In our problem, we look for the maximal matching that gives us the maximum sum of the lengths of its edges. This problem can be solved by the Hungarian method, which solves the assignment problem in its bipartite graph form with $2n$ nodes, in $O(n^3)$ arithmetic operations. For a detailed description and analysis of the Hungarian method for the assignment problem in its bipartite graph form, see [97].

Figure 3.5.2: A maximal matching.

Having applied the Hungarian algorithm for the above example, we obtain the maximal matching depicted in Figure 3.5.2. From this graph, we can constuct the optimal permutation matrix P, setting $P_{ij} = 1$ if V_j is joined to U_i. Thus, the required permutation matrix is

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

A Fortran 77 routine for the Hungarian algorithm, written by Carpaneto and Toth, is given in [20]. This routine can be also obtained from the netlib distribution system (file TOMS/548).

3.5.2 Solving the TSPOPP

The link between the TSPOPP

$$\min_{V^TV=I_n, P \text{ permutation}} \|A - PBV\|_F, \tag{3.5.3}$$

and the TSOPP

$$\min_{U^TU=I_m, V^TV=I_n} \|A - UBV\|_F, \tag{3.5.4}$$

where A, B are given m-by-n real matrices, is clear by examining the above forms (3.5.3) and (3.5.4). The only difference is that U is a general orthogonal matrix while P is a permutation matrix, a special case of an orthogonal matrix. Gower [48] suggests the following approximate noniterative solution to minimizing (3.5.3) based on a suggestion first made by Schönemann in 1968 [107].

1. Find U and V for the TSOPP.
2. Estimate P as the nearest permutation matrix to U.

Step 2 is equivalent to minimizing $\|U - P\|_F$, where all the matrices are m-by-m. Since

$$\begin{aligned}\|U-P\|_F &= \operatorname{tr}(U-P)^T(U-P)\\ &= \operatorname{tr}(U^T-P^T)(U-P)\\ &= \operatorname{tr}(U^TU-U^TP-P^TU+P^TP)\\ &= \operatorname{tr}(I-U^TP-P^TU+I)\\ &= 2m-2\operatorname{tr}P^TU,\end{aligned}$$

the problem is equivalent to maximizing $\operatorname{tr}P^TU$ for which we can use the methods of the previous subsection. Schönemann [107], who first suggests this method, says: *It is hoped that eventually superior alternatives will emerge.* Investigating this noniterative method, we found many examples where the method fails to solve the TSPOPP (3.5.3). A typical one is the following.

Suppose the 4-by-3 matrices A and B

$$A=\begin{bmatrix}2&9&0\\1&4&1\\7&5&5\\7&8&7\end{bmatrix},\quad B=\begin{bmatrix}1&4&1\\2&9&0\\7&8&7\\7&5&5\end{bmatrix}.$$

Matrix B has the same entries with A but its rows are in a different order. Solving the problem as a TSOPP by the SVD approach described in Section 3 gives the orthogonal matrices

$$U=\begin{bmatrix}0.6277&0.7293&0.1453&-0.2304\\-0.7625&0.6277&0.1566&0.0082\\0.0082&-0.2304&0.9683&-0.0962\\0.1566&0.1453&0.1294&0.9683\end{bmatrix},\quad V=\begin{bmatrix}-0.0175&0.1695&0.9854\\0.1695&0.9718&-0.1642\\0.9854&-0.1642&0.0457\end{bmatrix},$$

and $\|A-UBV\|_F$ is equal to zero modulo roundoff errors as expected. For the singular value decompositions, we used the subroutine DGESVD from LAPACK [1]. The permutation matrix P which maximizes $\operatorname{tr}P^TU$ is the identity matrix I_4 and $\|A-I_4UV\|_F \approx 8.53$.

But if $V = I_3$ and

$$P = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix},$$

$\|A - PBV\|_F = 0$ and the need for another approach to the solution of the problem is now apparent.

The problem of minimizing $\|A - PBV\|_F$ is equivalent to maximizing $\operatorname{tr} PBVA^T$. Gower [48] suggests the following iterative procedure for its solution, which is an example of a flip-flop algorithm as mentioned in Section 3.4. First fix P and determine V as the solution of the orthogonal Procrustes problem

$$\min_{V^TV=I} \|A - (PB)V\|_F; \tag{3.5.5}$$

then fix V and determine P by the method discussed in the previous section, repeating the process until it converges.

Since the convergence of an iterative method to the global optimum depends on its initial value, we thought of modifying Gower's suggestion by beginning with $V = I_n$, and determining P at the initial stage of the method. Then, determine V and repeat the process until it converges. The reason is, that in many applications B has the same entries with A, or slightly perturbed, and its rows are in a different order [22]. In that case, this iterative method would first determine the permutation matrix which *matches* the same or slightly different rows and then it would determine the orthogonal matrix V. If the entries of A and B are the same, the solution to the orthogonal Procrustes problem will be the identity matrix and the iterative procedure will stop. If the rows are slightly different, the method will determine an orthogonal matrix V, and then finding again the same permutation matrix P, it will stop. In that case Gower's suggestion (for the iterative method) does not always work since it converges to local optima different than the global ones. The following algorithm, describes our iterative approach to the solution of TSPOPP.

Algorithm 1

Given A and $B \in \mathbf{R}^{m\times n}$ $(m \geq n)$ and a tolerance *epsilon* > 0, this algorithm computes an m-by-m permutation matrix P and an orthogonal matrix $V \in \mathbf{R}^{n\times n}$ that minimize $\|A - PBV\|_F$.

$k = 0;$

$V_k \equiv I_n;$

Find the permutation matrix P_k that maximizes $\operatorname{tr} P_k B A^T$;

$\rho_k = \|A - P_k B\|_F;$

while $\rho_k > \epsilon$

  $k = k + 1;$

  Find the permutation matrix P_k that maximizes $\operatorname{tr} P_k B V_{k-1} A^T$;

  $\rho_k = \|A - P_k B V_{k-1}\|_F;$

  if $(\rho_{k-1} - \rho_k) \leq \epsilon$, quit, **end**;

  Compute the polar decomposition of $(P_k B)^T A$;

  Set V_k equal to the orthogonal factor of the polar decomposition;

  $\rho_k = \|A - P_k B V_k\|_F;$

end

As we mentioned in Section 3.4, the convergence to the global optimum is not always assured for Algorithm 1. Nevertheless, a large number of numerical experiments made us feel certain that Algorithm 1 is a reliable way to compute P and V, especially when A and B have the same or slightly different entries and their rows are in a different order. The following numerical examples are representatives of a large number of numerical experiments on the TSPOPP.

We first consider the 4-by-3 matrices A and B which have the same entries but their rows are in a different order.

$$A = \begin{bmatrix} 2.0000 & 9.0000 & 0.0000 \\ 1.0000 & 4.0000 & 1.0000 \\ 7.0000 & 5.0000 & 5.0000 \\ 7.0000 & 8.0000 & 7.0000 \end{bmatrix}, \quad B = \begin{bmatrix} 7.0000 & 8.0000 & 7.0000 \\ 7.0000 & 5.0000 & 5.0000 \\ 1.0000 & 4.0000 & 1.0000 \\ 2.0000 & 9.0000 & 0.0000 \end{bmatrix}.$$

One solution to the TSOPP is

$$U = \begin{bmatrix} 0.7518 & 0.3160 & -0.2720 & -0.5109 \\ -0.2424 & 0.6824 & 0.6338 & -0.2720 \\ 0.5874 & -0.2992 & 0.6824 & 0.3160 \\ 0.1764 & 0.5874 & -0.2424 & 0.7518 \end{bmatrix}, \quad V = \begin{bmatrix} -0.4157 & 0.7791 & 0.4693 \\ 0.7791 & 0.0388 & 0.6257 \\ 0.4693 & 0.6257 & -0.6231 \end{bmatrix}.$$

for which $\|A - UBV\|_F$ is equal to zero modulo roundoff error. The nearest permutation matrix P to U is the identity matrix I_4 and $\|A - PBV\|_F \approx 11.96$. Algorithm 1 with error tolerance $\epsilon = 10^{-14}$, gives

$$P = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}, \quad V = \begin{bmatrix} 1.0000 & 0.0000 & 0.0000 \\ 0.0000 & 1.0000 & 0.0000 \\ 0.0000 & 0.0000 & 1.0000 \end{bmatrix},$$

and $\|A - PBV\|_F$ is obviously zero. It is worth noting that for this numerical example, Algorithm 1 does not enter its while loop.

If we perturb now some entries of B, such that

$$A = \begin{bmatrix} 2.0000 & 9.0000 & 0.0000 \\ 1.0000 & 4.0000 & 1.0000 \\ 7.0000 & 5.0000 & 5.0000 \\ 7.0000 & 8.0000 & 7.0000 \end{bmatrix}, \quad B = \begin{bmatrix} 7.3000 & 7.7000 & 6.6000 \\ 7.0000 & 4.8000 & 5.1000 \\ 1.0000 & 4.2000 & 1.0000 \\ 1.6000 & 9.0000 & 0.5000 \end{bmatrix},$$

then the matrices U and V for the TSOPP are

$$U = \begin{bmatrix} 0.7444 & 0.3359 & -0.2861 & -0.5012 \\ -0.2605 & 0.6981 & 0.6108 & -0.2677 \\ 0.5883 & -0.2871 & 0.7020 & 0.2806 \\ 0.1787 & 0.5633 & -0.2286 & 0.7736 \end{bmatrix}, \quad V = \begin{bmatrix} -0.4044 & 0.8243 & 0.3962 \\ 0.7316 & 0.0315 & 0.6810 \\ 0.5489 & 0.5653 & -0.6158 \end{bmatrix},$$

and $\|A - UBV\|_F \approx 0.55$. The matrices P and V given by Algorithm 1 with the same error tolerance in 3 iterations are

$$P = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}, \quad V = \begin{bmatrix} 0.9972 & -0.0281 & 0.0696 \\ 0.0321 & 0.9979 & -0.0570 \\ -0.0678 & 0.0591 & 0.9959 \end{bmatrix},$$

and $\|A - PBV\|_F \approx 0.64$. As expected, the permutation matrix P is the same as in previous numerical example. And here Schönemann-Gower's suggestion fails to provide a better solution than Algorithm 1, since it yields the matrices

$$P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad V = \begin{bmatrix} -0.4031 & 0.7980 & 0.4480 \\ 0.7625 & 0.0222 & 0.6466 \\ 0.5061 & 0.6022 & -0.6175 \end{bmatrix},$$

and $\|A - PBV\|_F \approx 11.85$.

In the last numerical example for the TSPOPP, A and B are the following random matrices

$$A = \begin{bmatrix} 2.0000 & 9.0000 & 0.0000 \\ 1.0000 & 4.0000 & 1.0000 \\ 7.0000 & 5.0000 & 5.0000 \\ 7.0000 & 8.0000 & 7.0000 \end{bmatrix}, \quad B = \begin{bmatrix} 10.0000 & 6.0000 & 5.0000 \\ 2.0000 & 9.0000 & 1.0000 \\ 8.0000 & 2.0000 & 3.0000 \\ 4.0000 & 1.0000 & 1.0000 \end{bmatrix}.$$

Here the matrices U and V for the TSOPP are

$$U = \begin{bmatrix} -0.0315 & 0.9886 & 0.1347 & -0.0590 \\ 0.5734 & 0.1382 & -0.7221 & 0.3614 \\ 0.1227 & -0.0161 & 0.5179 & 0.8464 \\ 0.8094 & -0.0570 & 0.4383 & -0.3866 \end{bmatrix}, \quad V = \begin{bmatrix} 0.9166 & 0.1544 & 0.3688 \\ -0.0496 & 0.9592 & -0.2783 \\ -0.3967 & 0.2368 & 0.8869 \end{bmatrix},$$

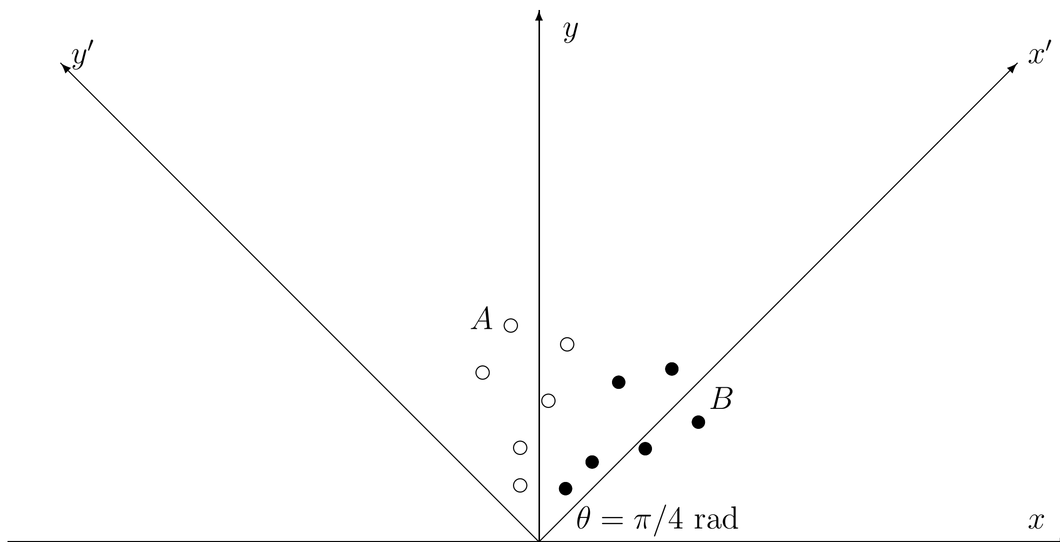

Figure 3.5.3: A possible form of the TSPOPP.

and $\|A - UBV\|_F \approx 1.35$. The nearest permutation matrix P to U is

$$P = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix},$$

for which $\|A - PBV\|_F \approx 11.88$. Again, Schönemann-Gower's suggestion fails to give a better solution than Algorithm 1 which yields

$$P = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}, \quad V = \begin{bmatrix} 0.9211 & 0.1588 & 0.3555 \\ -0.0690 & 0.9652 & -0.2523 \\ -0.3832 & 0.2079 & 0.9000 \end{bmatrix},$$

and $\|A - PBV\|_F \approx 3.92$ for the same error tolerance in 3 iterations.

Finally, to illustrate a possible application of Algorithm 1 for the TSPOPP consider the following problem, where the data are artificial and they have been constructed so that an exact solution exists. Similar problems are encountered in multivariate analysis [22] when one wishes to test whether two measurements correspond to the same sample. Let the 6-by2 matrices

$$A = \begin{bmatrix} 3.5355 & 53.0330 \\ -7.0711 & 35.3553 \\ -10.6066 & 81.3173 \\ 10.6066 & 74.2462 \\ -7.0711 & 21.2132 \\ -21.2132 & 63.6396 \end{bmatrix}, \quad B = \begin{bmatrix} 10.0000 & 20.0000 \\ 20.0000 & 30.0000 \\ 30.0000 & 60.0000 \\ 40.0000 & 3.0000 \\ 50.0000 & 6.0000 \\ 60.0000 & 4.0000 \end{bmatrix},$$

where A and B may represent two measured samples. Solving the TSPOPP for A and B using Algorithm 1 with error tolerance $\epsilon = 10^{-14}$, we find the following results

$$P = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}, \quad V = \begin{bmatrix} 0.7071 & 0.7071 \\ -0.7071 & 0.7071 \end{bmatrix},$$

and $\|A - PBV\|_F \approx 0$ in 2 iterations. One can notice that the orthogonal matrix V is the Givens rotation

$$\begin{bmatrix} \cos(\pi/4) & \sin(\pi/4) \\ -\sin(\pi/4) & \cos(\pi/4) \end{bmatrix}.$$

In the the two-dimensional space the above problem may be illustrated by Figure 3.5.3 where $\circ$'s stand for the entries of A (with coordinates in the $x'y'$ plane), and $\bullet$'s for the entries of B (with coordinates in the xy plane).

The TSPOPP is certainly a nontrivial problem and investigation of it stopped with Schönemann's suggestion in 1968. Unfortunately we cannot claim that Algorithm 1 is the ultimate solution since there is no way to guarrantee global convergence. Nevertheless, it is certainly a step which might help researchers in multivariate analysis to find acceptable solutions to this *notorious* problem.

3.5.3 A Two-Sided Permutation Problem

In this subsection we consider the following problem:

THE TWO-SIDED PERMUTATION PROCRUSTES PROBLEM (TSPPP)
Given $A, B \in \mathbf{R}^{m\times n}$, $m \geq n$, find two permutation matrices $P \in \mathbf{R}^{m\times m}$ and $Q \in \mathbf{R}^{n\times n}$ to minimize

$$\|A - PBQ\|_F. \tag{3.5.6}$$

The TSPPP can be viewed as the problem of revealing a presumed pattern by permuting the rows and columns of an observed data matrix. Such problems arise in a number of situations, as for example in Guttman's radex theory [49], in certain areas of scaling [25, 51], and in cluster analysis problems [21, 118].

Since

$$\|A - PBQ\|_F = \operatorname{tr} A^T A - 2 \operatorname{tr} Q^T B^T P^T A + \operatorname{tr} B^T B,$$

the problem is equivalent to maximizing $\operatorname{tr} Q^T B^T P^T A$ and the problem may be treated as a straight-forward application of the following flip-flop algorithm.

Algorithm 2
Given A and $B \in \mathbf{R}^{m\times n}$ $(m \geq n)$ and a tolerance *epsilon* > 0, this algorithm computes an m-by-m permutation matrix P and an n-by-n permutation matrix Q that minimize $\|A - PBQ\|_F$.

$k = 0$;

$P_k \equiv I_m$ (or $Q_k \equiv I_n$);

Find the permutation matrix Q_k that maximizes $\operatorname{tr} Q_k^T B^T A$

(or Find the permutation matrix P that maximizes $\operatorname{tr} P_k B A^T$);

$\rho_k = \|A - P_k B Q_k\|_F$;

while $\rho_k > \epsilon$

  $k = k + 1$;

  Find the permutation matrix P_k that maximizes $\operatorname{tr} P_k B Q_{k-1} A^T$

  (or Find the permutation matrix Q_k that maximizes $\operatorname{tr} Q_k^T B^T P_{k-1}^T A$);

$\rho_k = \|A - P_k B Q_{k-1}\|_F$ (or $\rho = \|A - P_{k-1} B Q_k\|_F$);

if $(\rho_{k-1} - \rho_k) \leq \epsilon$, quit, **end**;

Find the permutation matrix Q_k that maximizes $\operatorname{tr} Q_k^T B^T P_k^T A$

(or Find the permutation matrix P_k that maximizes $\operatorname{tr} P_k B Q_k A^T$);

$\rho_k = \|A - P_k B Q_k\|_F$;

end

Since a flip-flop algorithm does not guarantee global convergence and the choice of the initially fixed matrix may affect the efficiency of the algorithm, we suggest that one may apply Algorithm 2 twice, that is for both P and Q initially fixed. The following numerical example confirms the usefulness of this idea. Assume the following 5-by-5 matrices

$$A = \begin{bmatrix} 32 & 14 & 3 & 63 & 50 \\ 24 & 22 & 1 & 56 & 4 \\ 94 & 16 & 28 & 75 & 81 \\ 19 & 72 & 42 & 90 & 54 \\ 71 & 85 & 10 & 96 & 58 \end{bmatrix}, \quad B = \begin{bmatrix} 58 & 96 & 85 & 10 & 71 \\ 81 & 75 & 16 & 28 & 94 \\ 4 & 56 & 22 & 1 & 24 \\ 54 & 90 & 72 & 42 & 19 \\ 50 & 63 & 14 & 3 & 32 \end{bmatrix}.$$

The matrices A and B have the same entries but their rows and columns are in different orders.

If we first fix $P \equiv I_m$, and apply Algorithm 2 with error tolerance $\epsilon = 10^{-14}$ for the solution of the TSPPP, the algorithm terminates in 3 iterations finding the permutation matrices

$$P = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix}, \quad Q = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix},$$

and $\|A - PBQ\|_F \approx 93.7977$. But if we initially fix $Q \equiv I_n$ then Algorithm 2 with the same error tolerance terminates in six iterations finding the exact solution to this TSPPP, that is

$$P = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix}, \quad Q = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix},$$

and $\|A - PBQ\|_F = 0$.

3.6 The Two-Sided Symmetric Procrustes Problem

The one-sided symmetric Procrustes problem (OSSPP)

$$\min_{X=X^T} \|AX - B\|_F, \quad A, B \in \mathbf{R}^{m\times n} \tag{3.6.1}$$

arises in the determination of the strain matrix of an elastic structure. The problem has been solved by Higham [63], and we will describe his solution later in this section. The problem can be extended to its two-sided form.

> THE TWO-SIDED SYMMETRIC PROCRUSTES PROBLEM (TSSPP)
> Given $A, B \in \mathbf{R}^{m\times n}$, $m \geq n$, find two symmetric matrices $X \in \mathbf{R}^{m\times m}$ and $Y \in \mathbf{R}^{n\times n}$ to minimize
>
> $$\|A - XBY\|_F. \tag{3.6.2}$$

It is a well-known result that any matrix $A \in \mathbf{R}^{n\times n}$ can be expressed in the form

$$A = S_1 S_2, \tag{3.6.3}$$

where S_1, $S_2 \in \mathbf{R}^{n\times n}$ are symmetric matrices [54, 116]. However, it is an open question if, for given $A, B \in \mathbf{R}^{m\times n}$, there exist symmetric matrices $X \in \mathbf{R}^{m\times m}$ and $Y \in \mathbf{R}^{n\times n}$ so that $A = XBY$. Clearly $\text{rank}(A) \leq \text{rank}(B)$ is necessary and if $A \in \mathbf{R}^{n\times n}$ and $B = I_n$, the answer is given trivially by (3.6.3).

We consider first the full-rank case, that is $\mathrm{rank}(A) = \mathrm{rank}(B) = n$. The system $A = XBY$, where X, Y are symmetric matrices, has mn equations in $\frac{1}{2}m(m+1) + \frac{1}{2}n(n+1)$ unknowns and thus there should be infinitely many solutions since it has $\frac{1}{2}((m-n)^2 + (m+n))$ degrees of freedom. If we assume that the matrix Y is nonsingular and set $Z = Y^{-1}$, then the system $A = XBY$ may be written as

$$AZ - XB = 0. \tag{3.6.4}$$

Equation (3.6.4) is a generalized Sylvester equation where the unknown matrices are symmetric. Equation (3.6.4) may be written as a homogeneous system

$$Cx = 0, \tag{3.6.5}$$

where the coefficient matrix C is (mn)-by-$(\frac{1}{2}m(m+1) + \frac{1}{2}n(n+1))$. For example, if $m = n = 2$,

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} z_{11} & z_{12} \\ z_{12} & z_{22} \end{bmatrix} - \begin{bmatrix} x_{11} & x_{12} \\ x_{12} & x_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = 0$$

may be written as

$$\begin{aligned} a_{11}z_{11} + a_{12}z_{12} - b_{11}x_{11} - b_{21}x_{12} &= 0, \\ a_{11}z_{12} + a_{12}z_{22} - b_{12}x_{11} - b_{22}x_{12} &= 0, \\ a_{21}z_{11} + a_{22}z_{12} - b_{11}x_{12} - b_{21}x_{22} &= 0, \\ a_{21}z_{12} + a_{22}z_{22} - b_{12}x_{12} - b_{22}x_{22} &= 0, \end{aligned}$$

or equivalently

$$\underbrace{\begin{bmatrix} a_{11} & a_{12} & 0 & -b_{11} & -b_{21} & 0 \\ 0 & a_{11} & a_{12} & -b_{12} & -b_{22} & 0 \\ a_{21} & a_{22} & 0 & 0 & -b_{11} & -b_{21} \\ 0 & a_{12} & a_{22} & 0 & -b_{12} & -b_{22} \end{bmatrix}}_{C} \underbrace{\begin{bmatrix} z_{11} \\ z_{12} \\ z_{22} \\ x_{11} \\ x_{12} \\ x_{22} \end{bmatrix}}_{x} = 0.$$

Since every homogeneous system with more unknowns than equations has a nontrivial solution, there exists a vector $x \in \text{null}(C)$, $x \neq 0$, so that $Cx = 0$. The vector x is not unique since every vector in the nullspace of C satisfies (3.6.5). The same approach may be followed when $\text{rank}(A) \leq \text{rank}(B)$, assuming again that the matrix Y is invertible. It is worth noting that if $B = I_n$ we can use the above method to decompose a given $A \in \mathbf{R}^{n\times n}$ into a product of two symmetric matrices without using the companion matrix as in [54, 116]. Although, the above method is based on an assumption, that is the matrix Y is invertible, it always yielded in our numerical experiments symmetric matrices X and Y so that $A = XBY$. (The arbitrary entries of the matrix Z have been selected so that the matrix Y be nonsingular.)

But if $\text{rank}(A) > \text{rank}(B)$ there are no symmetric matrices X and Y so that $A = XBY$, since $\text{rank}(A) > \text{rank}(XBY)$ for any $X \in \mathbf{R}^{m\times m}$ and $Y \in \mathbf{R}^{n\times n}$. In this case the solution of the homogeneous system (3.6.5) corresponds to a singular Z and hence we cannot obtain X by its inversion. At this point it is necessary to describe Higham's solution to the OSSPP [63].

Higham considers the OSSPP in the form (3.6.1). If the singular value decomposion of A is

$$A = P \begin{bmatrix} \Sigma \\ 0 \end{bmatrix} Q^T,$$

then

$$\begin{aligned} \|AX - B\|_F^2 &= \|P \begin{bmatrix} \Sigma \\ 0 \end{bmatrix} Q^T X - B\|_F^2 \\ &= \| \begin{bmatrix} \Sigma \\ 0 \end{bmatrix} Q^T X Q - P^T B Q\|_F^2 \\ &= \| \begin{bmatrix} \Sigma \\ 0 \end{bmatrix} Y - C\|_F^2 \\ &= \|\Sigma Y - C_1\|_F^2 + \|C_2\|_F^2, \end{aligned}$$

where $Y = Q^T XQ$ and

$$C = \begin{bmatrix} C_1 \\ C_2 \end{bmatrix} = P^T BQ, \quad C_1 \in \mathbf{R}^{n \times n}.$$

Thus, the OSSPP reduces to minimizing the quantity

$$\|\Sigma Y - C_1\|_F^2 = \sum_{i=1}^{n} (\sigma_i y_{ii} - c_{ii})^2 + \sum_{j>i} ((\sigma_i y_{ij} - c_{ji})^2 + (\sigma_j y_{ij} - c_{ji})^2), \tag{3.6.6}$$

where the required symmetry of Y has been assumed. The variables y_{ij}, $j \geq i$, in (3.6.6) are uncoupled, so it suffices to minimize independently each of the terms $(\sigma_i y_{ii} - c_{ii})^2$ and $(\sigma_i y_{ij} - c_{ji})^2 + (\sigma_j y_{ij} - c_{ji})^2)$, $j > i$. The general solution is given by

$$y_{ij} = \begin{cases} \dfrac{\sigma_i c_{ij} + \sigma_j c_{ji}}{\sigma_i^2 + \sigma_j^2}, & \sigma_i^2 + \sigma_j^2 \neq 0, \\ \text{arbitrary}, & \text{otherwise}, \end{cases}$$

and the required solution is $X = QYQ^T$.

If the OSSPP is given in the form

$$\min_{X = X^T} \|XA - B\|_F, \quad A, B \in \mathbf{R}^{m \times n}, \quad m \geq n,$$

then since $\|XA - B\|_F = \|A^T X - B^T\|_F$, the problem is equivalent to finding the symmetric matrix X that minimizes $\|A^T X - B^T\|_F$, where $A^T, B^T \in \mathbf{R}^{n \times m}$, $n \leq m$. If

$$A^T = P \begin{bmatrix} \Sigma & 0 \end{bmatrix} Q^T$$

is the singular value decomposition of A^T, then using the invariance of the Frobenius norm under orthogonal transformations we have

$$\begin{aligned} \|A^T X - B^T\|_F^2 &= \|P \begin{bmatrix} \Sigma & 0 \end{bmatrix} Q^T X - B^T\|_F^2 \\ &= \| \begin{bmatrix} \Sigma & 0 \end{bmatrix} Q^T X - P^T B^T\|_F^2 \\ &= \| \begin{bmatrix} \Sigma & 0 \end{bmatrix} Q^T X Q - P^T B^T Q\|_F^2 \\ &\equiv \| \begin{bmatrix} \Sigma & 0 \end{bmatrix} \begin{bmatrix} Y_{11} & Y_{12} \\ Y_{12}^T & Y_{22} \end{bmatrix} - \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} \|_F^2 \\ &= \|\Sigma Y_{11} - C_{11}\|_F^2 + \|\Sigma Y_{12} - C_{12}\|_F^2. \end{aligned}$$

Here, $Y_{11} \in \mathbf{R}^{n\times n}$ is symmetric and $Y_{12} \in \mathbf{R}^{n\times(m-n)}$ is arbitrary, so the minimum is achieved when the entries of Y_{11} are given by

$$y_{ij} = \begin{cases} \dfrac{\sigma_i c_{ij} + \sigma_j c_{ji}}{\sigma_i^2 + \sigma_j^2}, & \sigma_i^2 + \sigma_j^2 \neq 0, \\ \text{arbitrary}, & \text{otherwise}, \quad 1 \leq i \leq j \leq n, \end{cases} \tag{3.6.7}$$

and the entries of Y_{12} are given by

$$y_{ij} = \begin{cases} \dfrac{c_{ij}}{\sigma_i}, & \sigma_i \neq 0, \\ \text{arbitrary}, & \text{otherwise}, \quad 1 \leq i \leq n, \quad n+1 \leq j \leq m. \end{cases}$$

The submatrix Y_{22} is arbitrary. The required symmetric matrix X is given by $X = QYQ^T$.

Since we can find the left and the right symmetric matrices that minimize the OSSPP, we can apply the following flip-flop algorithm to attempt to solve the TSSPP when $\text{rank}(A) > \text{rank}(B)$.

Algorithm 3
Given A and $B \in \mathbf{R}^{m\times n}$ $(m \geq n)$ and a tolerance *epsilon* > 0, this algorithm computes a symmetric matrix $X \in \mathbf{R}^{m\times m}$ and a symmetric matrix $Y \in \mathbf{R}^{n\times n}$ that minimize $\|A - XBY\|_F$.

$k = 0$;

$X_k \equiv I_m$ (or $Y_k \equiv I_n$);

Find the symmetric matrix Y_k that minimizes $\|A - BY_k\|_F$

(or Find the symmetric matrix X that minimizes $\|A - X_kB\|_F$;

$\rho_k = \|A - X_kBY_k\|_F$;

while $\rho_k > \epsilon$

  $k = k + 1$;

  Find the symmetric matrix X_k that minimizes $\|A - X_kBY_{k-1}\|_F$

  (or Find the symmetric matrix Y_k that minimizes $\|A - X_{k-1}BY_k\|_F$);

  $\rho_k = \|A - X_kBY_{k-1}\|_F$ (or $\rho = \|A - X_{k-1}BY_k\|_F$);

  if $(\rho_{k-1} - \rho_k) \leq \epsilon$, quit, **end**;

  Find the symmetric matrix Y_k that minimizes $\|A - X_kBY_k\|_F$

(or Find the permutation matrix X_k that minimizes $\|A - X_k B Y_k\|_F$);

$\rho_k = \|A - X_k B Y_k\|_F$;

end

For the same reasons explained in Subsection 3.5.3 it may be recommended to apply Algorithm 3 twice, that is for both X and Y initially fixed. During our numerical experiments, for $\text{rank}(A) \leq \text{rank}(B)$, Algorithm 3 always converged to zero, no matter which of X and Y had been fixed initially. (In this the solution may be obtained by solving the Sylvester equation (3.6.4).) But there is usually a difference in the number of iterations. A typical numerical example is the following. Consider the 3-by-2 matrices

$$A = \begin{bmatrix} 87 & 3 \\ 93 & 57 \\ 41 & 23 \end{bmatrix}, \quad B = \begin{bmatrix} 7 & 42 \\ 52 & 9 \\ 70 & 94 \end{bmatrix},$$

where $\text{rank}(A) = \text{rank}(B) = 2$. If we apply Algorithm 3 for the solution of the TSSPP fixing initially $X \equiv I_m$, with error tolerance $\epsilon = 10^{-14}$, we obtain the following results in 76 iterations (rounded here to four decimal digits)

$$X = \begin{bmatrix} -1.0406 & 1.1796 & 0.3390 \\ 1.1796 & 0.8845 & 0.3564 \\ 0.3390 & 0.3564 & 0.2024 \end{bmatrix}, \quad Y = \begin{bmatrix} 1.1193 & 0.0478 \\ 0.0478 & 0.5847 \end{bmatrix},$$

and $\|A - XBY\|_F = 8.9 \times 10^{-15}$. But if we first fix $Y \equiv I_n$ and apply Algorithm 3 with the same error tolerance, we obtain the following results, this time in 125 iterations,

$$X = \begin{bmatrix} 2.1286 & -1.0367 & 0.7733 \\ -1.0367 & 1.0413 & 0.9631 \\ 0.7733 & 0.9631 & -0.1284 \end{bmatrix}, \quad Y = \begin{bmatrix} 0.5601 & 0.5141 \\ 0.5141 & -0.0313 \end{bmatrix},$$

and $\|A - XBY\|_F = 9.2 \times 10^{-15}$. The number of iterations is certainly a drawback of Algorithm 3. For larger matrices the number of iterations increases enormously. For example, in a numerical experiment for $A, B \in \mathbf{R}^{5\times 5}$ Algorithm 3 required 43712 iterations. Therefore, one may think that it would be better to solve the homogeneous system

(3.6.5), constructing the coefficient matrix C from A and B. It is remarkable that in our numerical experiments for $\text{rank}(A) \leq \text{rank}(B)$, Algorithm 3 always converged but with a very disappointing linear convergence. But this fact encouraged us to apply Algorithm 3 for the TSSPP when $\text{rank}(A) > \text{rank}(B)$. In this this case, the matrix Z given by the solution of the homogeneous system (3.6.5), was always found to be singular (or very close to singular due to rounding errors), and therefore we cannot solve the TSSPP using the solution of (3.6.5).

It is remarkable that during our numerical experiments we noticed that Algorithm 3 needed a much smaller number of iterations when $\text{rank}(A) > \text{rank}(B)$ and always converged to the same limit, no matter which of X and Y had been fixed initially. The following numerical example demonstrates this observation. Let

$$A = \begin{bmatrix} 10 & 83 \\ 52 & 58 \\ 58 & 44 \end{bmatrix}, \quad B = \begin{bmatrix} 16 & 16 \\ 65 & 65 \\ 14 & 14 \end{bmatrix}.$$

Here, $\text{rank}(A) = 2$ and $\text{rank}(B) = 1$. Fixing X first, Algorithm 3 yields the following results with error tolerance $\epsilon = 10^{-14}$ in 4 iterations.

$$X = \begin{bmatrix} 0.2409 & 0.5530 & 0.2436 \\ 0.5530 & 0.5173 & 0.6171 \\ 0.2436 & 0.6171 & 0.2418 \end{bmatrix}, \quad Y = \begin{bmatrix} 0.5381 & 0.5381 \\ 0.5381 & 0.5381 \end{bmatrix},$$

and $\|A - XBY\|_F \approx 52.7304$. Fixing Y first, Algorithm 3 gives the following results with the same error tolerance in 2 iterations.

$$X = \begin{bmatrix} 0.2593 & 0.5951 & 0.2621 \\ 0.5951 & 0.5566 & 0.6640 \\ 0.2621 & 0.6640 & 0.2602 \end{bmatrix}, \quad Y = \begin{bmatrix} 0.5000 & 0.5000 \\ 0.5000 & 0.5000 \end{bmatrix},$$

and $\|A - XBY\|_F \approx 52.7304$. It is worth noting that in both cases Algorithm 3 yielded the same minimum in different number of iterations and the same behaviour of Algorithm 3 has been noticed in a large number of numerical experiments with larger matrices, no matter which of X or Y has been initially fixed.

Problem	**Tools for solution**
GTSPP	Singular value decomposition.
TSOPP	Singular value decomposition.
TSRPR	Polar decomposition and spectral decomposition of a symmetric matrix.
TSPOPP	Polar Decomposition and Hungarian algorithm.
TSPPP	Hungarian algorithm.
TSSPP	Singular value decomposition.

Table 3.7.1: Two-sided Procrustes Problems and tools for their solution.

3.7 Concluding Remarks

In this chapter we discussed some possible forms of the two-sided Procrustes problems. From these problems only the TSGPP and the TSOPP can be always solved analytically. For the rest of them, iterative flip-flop algorithms seem to be efficient methods in most cases. When $\text{rank}(A) \leq \text{rank}(B)$, a solution to the TSSPP may be found by solving a homogeneous system.

Table 7.1 summarizes the tools needed for our methods, in order to solve the two-sided Procrustes problems discussed in this chapter. (Table 7.1 does not include tools for the solution of the homogeneous system in the TSSPP.) Three of them, that is the singular value decomposition, the polar decomposition, and the spectral decomposition of a symmetric matrix, are among the most important decompositions in numerical linear algebra. Apart from tools for the solution of the two-sided Procrustes problems discussed in this chapter, they are important tools in many applications that arise in various scientific fields. The discussed two-sided Procrustes problems occupy a small part in the spectrum of their applications. Their importance motivated us to focus our attention on parallel algorithms for these decompositions. In the following chapters we discuss the development of new parallel algorithms for these decompositions and their implementation on the KSR1. These parallel algorithms may be used for the solution of these two-sided Procrustes, especially when the matrices A and B are large dense matrices.

The Hungarian algorithm has been developed for the solution of the assignment problem, a problem which arises mainly in operational research [24, 115]. The Hungarian algorithm, as developed by Kuhn [82] and studied and improved by other researchers (see for example [97]), is not suitable for parallel computation. An algorithm with a parallel potential for the assignment problem is the *auction algorithm*, developed by Bertsekas [10]. Parallel implementations of the auction algorithm on a shared memory machine are discussed in [11]. We did not implement the auction algorithm on the KSR1.

In the following chapters we discuss parallel methods for the polar decomposition, the spectral decomposition of a symmetric matrix, and the singular value decomposition. The discussion commences with the polar decomposition, a decomposition that turned out to play a key role in our research.

Chapter 4

The Polar Decomposition

4.1 Introduction

Every nonzero complex number z has a unique polar representation $z = pu$, where p is a positive real number and u is a complex number of modulus 1. If $z = 0$, then z still has a polar representation with $p = 0$, but u is no longer uniquely determined. This representation can be generalized to any matrix $A \in \mathbf{C}^{m\times n}$. Assuming that $m \geq n$, any matrix $A \in \mathbf{C}^{m\times n}$ may be written as

$$A = UH, \tag{4.1.1}$$

where $U \in \mathbf{C}^{m\times n}$ has orthonormal columns ($U^*U = I_n$), and $H \in \mathbf{C}^{n\times n}$ is Hermitian positive semidefinite. The matrix H has the same rank as A and is uniquely determined as $H = (A^*A)^{1/2}$, where $G^{1/2}$ denotes the unique Hermitian positive semidefinite square root[1] of the Hermitian positive semidefinite matrix G. If A has full rank, then U is uniquely determined and H is positive definite. If A is real, then both U and H may be taken to be real. Analogously, if $m < n$, then A may be written as $A = HU$, where $UU^* = I_m$ and $H = (AA^*)^{1/2}$. The factorization (4.1.1) is known as the *polar form* or *polar decomposition* of the matrix A.

The polar decomposition was introduced by Autonne in 1902 [4]. Autonne showed that every nonsingular $A \in \mathbf{C}^{n\times n}$ can be written as $A = UH$, where $U \in \mathbf{C}^{n\times n}$ is unitary

[1]Given a matrix A, a matrix X for which $X^2 = A$ is called a square root of A.

and $H \in \mathbf{C}^{n\times n}$ is positive definite. Autonne extended his ideas to the singular value decomposition in 1915 [5]. However, his pioneering work on these two closely associated decompositions seems to have been overlooked by his contemporary researchers. Winter and Murnaghan [125], claiming to be unaware of Autonne's prior work, rediscovered the polar decomposition in 1931. They additionally observed that one may always write $A = GU = UH$ (the same unitary U) if and only if A is normal. Finally, in 1935, Williamson [124] published a complete version of the polar decomposition for a rectangular complex matrix. Williamson acknowledged the fundamental contributions of both Autonne and Winter–Murnaghan to the evolution of the polar decomposition.

This chapter is devoted entirely to the polar decomposition and is structured in the following way. Section 2 introduces the properties of the polar decomposition. Section 3 is a survey of the existing sequential algorithms. Section 4 concetrates on the relationship between the polar decompostion and the matrix sign function. In Section 5 we present a new parallel algorithm for computing the polar decomposition. In Section 6 we discuss its implementation on the KSR1 and in Section 7 we present our experimental results. In Section 8 we present some applications from various scientific fields, where the polar decomposition plays an important role. Finally in Section 9 we summarize our conclusions.

4.2 Properties of the Polar Decomposition

There is a close relationship between the polar decomposition and the singular value decomposition. Let $A \in \mathbf{C}^{m\times n}$, $m \geq n$, have the singular value decomposition

$$A = P\Sigma Q^*, \tag{4.2.1}$$

where $P \in \mathbf{C}^{m\times m}$ and $Q \in \mathbf{C}^{n\times n}$ are unitary and

$$\Sigma = \mathrm{diag}(\sigma_1, \ldots, \sigma_n), \quad \sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_n \geq 0.$$

Inserting $Q^*Q = I_n$ before Σ,

$$A = P(Q^*Q)\Sigma Q^* = \underbrace{PQ^*}_{U}\underbrace{Q\Sigma Q^*}_{H},$$

we obtain the polar factors

$$U = PQ^* \quad \text{and} \quad H = Q\Sigma Q^*. \tag{4.2.2}$$

If $A \in \mathbf{C}^{n\times n}$, and $A = UH$ is its given polar decomposition, one may construct the singular value decomposition $A = (UQ)\Sigma Q^*$ using the spectral decomposition $H = Q\Sigma Q^*$. This is a worthwhile observation because it suggests a parallel method for computing the singular value decomposition. Presupposing the existence of an efficient parallel algorithm for the polar decomposition, one may obtain an effective parallel algorithm for the singular value decomposition using a parallel algorithm for solving the symmetric eigenvalue problem. This observation deserves attention, since the singular value decomposition is one of the most important tools in numerical linear algebra. This suggestion is being investigated in the next chapter of this thesis.

The following Lemma (4.2.1), taken from [61], summarises some elementary properties of the polar decomposition. $\lambda(A)$ and $\sigma(A)$ denote, respectively, the set of the eigenvalues and the set of the singular values of $A \in \mathbf{C}^{n\times n}$. $\kappa_2 = \|A\|_2\|A^{-1}\|_2 = \sigma_1/\sigma_n$ is the 2-norm condition number. Recall that $A \in \mathbf{C}^{n\times n}$ is said to be *normal* if A commutes with its Hermitian adjoint, that is $A^*A = AA^*$.

Lemma 4.2.1 *Let $A \in \mathbf{C}^{n\times n}$ have the polar decomposition $A = UH$. Then*

(i) $\lambda(H) = \sigma(A) = \sigma(H)$.

(ii) $\kappa_2(A) = \kappa_2(H)$.

(iii) *A is normal if and only if $UH = HU$.*

Proof. (i) Let $H = Q\Sigma Q^*$ be the spectral decomposition of the Hermitian polar factor H. As we mentioned earlier, the singular value decomposition of A can be written as

$$A = (UQ)\Sigma Q^*,$$

and therefore $\lambda(H) = \sigma(A) = \sigma(H)$.

(ii) It is immediate from (i) since $\sigma(A) = \sigma(H)$.

(iii) If A is normal, then $H^2 = UH^2U^*$. Now H^2 and UH^2U^* are both positive semidefinite matrices and H and UHU^* are their respective positive semidefinite square roots. It is known [67, Theorem 7.2.6, page 405] that the square root of a positive semidefinite matrix is unique, and therefore $H = UHU^*$, that is $UH = HU$. If U and H commute, then $AA^* = (UH)(UH)^* = (HU)(HU)^* = (HU)(U^*H) = H^2$, $A^*A = (UH)^*(UH) = HU^*UH = H^2$, and hence A is normal. □

The unitary polar factor possesses two significant approximation properties. The possession of these properties explains the widespread use of the polar decomposition in various applications. The discussion of these applications is deferred until Section 8. The first approximation property is that the unitary polar factor is the closest unitary matrix to a given matrix in both the Frobenius and the 2-norm. To be more specific, let $A \in \mathbf{C}^{m\times n}$ $(m \geq n)$, and let $A = UH$ be its polar decomposition. Then

$$\|A - U\| = \min\{\, \|A - Q\| : Q^*Q = I,\ Q \in \mathbf{C}^{m\times n}\,\}$$

for both the Frobenius and the 2-norm; if $m = n$ then the result is true for any unitarily invariant norm [39].

The second approximation property of the unitary polar factor is that it solves the *orthogonal Procrustes problem*,

$$\min\{\, \|A - BQ\|_F : Q^*Q = I,\ Q \in \mathbf{C}^{m\times n}\,\}, \quad A, B \in \mathbf{C}^{m\times n},$$

which arises in numerous applications[2]. The solution to this problem is the unitary polar factor of the matrix B^*A.

The Hermitian polar factor also possesses noteworthy properties. These properties are summarised in Lemma (4.2.2) taken from [61]. In this work, Higham defines for any Hermitian matrix B,

$$\delta(B) = \min\{\|E\|_2 : B + E \text{ is Hermitian positive semidefinite}\}.$$

We denote by λ_n the smallest eigenvalue of a $n \times n$ matrix with only real eigenvalues.

Lemma 4.2.2 *Let $A \in \mathbf{C}^{n\times n}$ be Hermitian with polar decomposition $A = UH$. Then*

[2] See Section 8.

(i) $\delta(A) = \max\{0, -\lambda_n(A)\} = \frac{1}{2}\|A - H\|_2$.

(ii) $\frac{1}{2}(A + H)$ *is a best Hermitian positive semidefinite approximation to* A *in the* 2-*norm.*

(iii) *For any Hermitian positive (semi-)definite* $X \in \mathbf{C}^{n\times n}$,

$$\|A - H\|_2 \leq 2\|A - X\|_2.$$

(iv) H *and* A *have a common set of eigenvectors.*

In Section 8 we discuss an application of the polar decomposition in optimization, where the Hessian matrix in Newton's method for the minimization of $F(x)$, $F : \mathbf{R}^n \to \mathbf{R}$, may be replaced by its polar factor.

4.3 Sequential Algorithms

As we mentioned in the previous section, the polar decomposition can be computed using the singular value decomposition. This approach provides a numerically stable way to compute the polar factors, providing that the singular value decomposition has been computed in a stable way. The standard approach to the computation of the singular value decomposition is the Golub–Reinsch algorithm [46], as implemented in LAPACK [1]. If $A \in \mathbf{C}^{m\times n}$, $(m \geq n)$, the computation proceeds in the following stages:

1. The matrix A is reduced to bidiagonal form: $A = P_1 B Q_1^*$, where P_1 and Q_1 are unitary, and B is real and upper bidiagonal, so that B is nonzero only on the main diagonal and the first superdiagonal.
2. The singular value decomposition of the bidiagonal matrix B is computed: $B = P_2 \Sigma Q_2^*$, where P_2 and Q_2 are orthogonal and Σ is the diagonal matrix of the singular values of A. The singular vectors of A are then $P = P_1 P_2$ and $Q = Q_1 Q_2$, that is the singular value decomposition of A is $A = P\Sigma Q^*$.

If $m \gg n$, it is more efficient to first perform a QR factorization of A, and then to compute the singular value decomposition of the n-by-n matrix R; if $A = QR$ and $R = U\Sigma V^*$, then the singular value decomposition of A is given by $A = (QU)\Sigma V^*$ [47, Sec. 5.4.5]. Once the singular value decomposition has been computed, the polar factors of A can be obtained as

$$U = P \begin{bmatrix} I_n \\ 0 \end{bmatrix} Q^*, \quad H = Q\Sigma Q^*. \tag{4.3.1}$$

The Golub–Reinsch SVD algorithm is not well-suited for parallel computation. Another basic drawback of this approach to compute the polar decomposition is that the algorithm requires a fixed number of floating point operations (somewhat more than $22n^3$ flops), regardless of the proximity of A to its unitary polar factor. In certain applications, especially in aerospace computations [8, 13], the columns of A are nearly orthogonal and there is a need for an algorithm that takes this fact into account. (These applications are discussed in Section 8). In 1971, Björck and Bowie [13], realizing the importance of devising an algorithm oriented to this problem, showed that the iteration

$$A_{k+1} = A_k \left(I + \frac{1}{2}T_k + \frac{3}{8}T_k^2 + \cdots + (-1)^p \begin{pmatrix} -1/2 \\ p \end{pmatrix} T_k^p \right), \quad A_0 = A, \tag{4.3.2}$$

where $T_k = I - A_k^* A_k$, converges locally to U with order $p+1$. (For $p = 1$, (4.3.2) is an inner Schulz iteration). The algorithm works for rectangular matrices but the sequence converges only if $\|I - A^*A\| < c_p$ for certain constants c_p.

Higham [61] considers the iteration

$$\begin{aligned} X_0 &= A \in \mathbf{C}^{n \times n}, \quad \text{nonsingular}, \\ X_{k+1} &= \frac{1}{2}\left(X_k + X_k^{-*}\right), \quad k = 0, 1, \ldots, \end{aligned} \tag{4.3.3}$$

where X_k^{-*} denotes $(X_k^{-1})^*$, and shows that the sequence converges quadratically to the unitary polar factor of A. The algorithm is based on the Newton iteration to compute the square root of a scalar. Although (4.3.3) is quadratically convergent, convergence can be slow initially, for example when A is an ill–conditioned matrix. The speed of convergence can be improved by scaling the iterates $X_k \leftarrow \gamma_k X_k$, so that iteration (4.3.3)

can be written as

$$X_{k+1} = \frac{1}{2}\left(\gamma_k X_k + \frac{1}{\gamma_k} X_k^{-*}\right). \tag{4.3.4}$$

From this point on, iteration (4.3.4) will be referred to as *Higham's method.* Higham finds very fitting that the optimum scaling factor γ_k is

$$\gamma_k = \left(\sigma_{\max}\left(X_k\right)\sigma_{\min}\left(X_k\right)\right)^{-1/2},$$

where $\sigma_{\min}$ and $\sigma_{\max}$ are the smallest and the largest singular values of X_k respectively. In practice, the exact computation of γ_k is too expensive, and Higham suggests as a good approximation the scaling factor

$$\widehat{\gamma}_k = \left(\frac{\|X_k^{-1}\|_1 \|X_k^{-1}\|_\infty}{\|X_k\|_1 \|X_k\|_\infty}\right)^{1/4}.$$

A thorough analysis of scaling the Newton iteration is given in [80].

In [66], Higham and Schreiber propose another algorithm to compute the polar decomposition of an arbitrary matrix. The idea is to use Higham's method for square matrices and to apply it to the triangular matrix obtained from a complete orthogonal decomposition of the original matrix. This decomposition requires rank decision. They also suggest switching from Higham's method to Schulz iteration once $\|X_k^* X_k - I\|_1 \le 0.1$, to maximize the amount of matrix-matrix multiplications.

In [42], Gander devises an iteration similar to (4.3.3) using Halley's iteration to compute the square root. This cubically convergent iteration,

$$X_{k+1} = X_k\left(X_k^* X_k + 3I\right)\left(3X_k^* X_k + I\right)^{-1}, \tag{4.3.5}$$

is more general than (4.3.3) since it is not confined to square matrices. Gander also proposes a family of iteration methods that contain (4.3.2) (for $p = 1$), (4.3.3), and (4.3.5), as special cases [43]. These methods converge globally and they do not need rank decisions. On the other hand , they require the numerical formation of $X_k^* X_k$, which may cause loss of information in X_k.

4.4 The Matrix Sign Function and the Polar Decomposition

The matrix sign function is a relatively recent concept in linear algebra. It was introduced by Roberts [101] in 1971 for the solution of the algebraic Riccati equation and the Lyapunov equation. From then on, a number of papers have been written about the matrix sign function since it can be used for the solution of many problems in control theory. Although Roberts defined the matrix sign function via a contour integral, the most propitious definition is the following.

Let $A \in \mathbf{C}^{n\times n}$ have no pure imaginary eigenvalues and

$$A = T \begin{bmatrix} P & 0 \\ 0 & N \end{bmatrix} T^{-1} \tag{4.4.1}$$

be its Jordan canonical form. P and N in are in block diagonal form with, respectively, positive and negative real part eigenvalues. Then the sign of A is given by

$$\text{sign}(A) = T \begin{bmatrix} I & 0 \\ 0 & -I \end{bmatrix} T^{-1}, \tag{4.4.2}$$

where I and $-I$ in (4.4.2) have the same dimensions with P and N in (4.4.1). Apparently, $\text{sign}(A)$ is uniquely defined and nonsingular since its eigenvalues are ± 1. Throughout this section we assume that A has no pure imaginary eigenvalues.

The following Lemma 4.4.1 summarizes some elementary properties of the matrix sign function.

Lemma 4.4.1 *Let $A \in \mathbf{C}^{n\times n}$ and $S = \text{sign}(A)$. Then,*

(i) *S is involutory ($S^2 = I$).*

(ii) *$S^{-1} = S$.*

(iii) *A commutes with S ($AS = SA$).*

Proof.

(i)

$$S^2 = SS = T \begin{bmatrix} I & 0 \\ 0 & -I \end{bmatrix} T^{-1} T \begin{bmatrix} I & 0 \\ 0 & -I \end{bmatrix} T^{-1} = T^{-1}T = I.$$

(ii) It is immediate from (i).

(iii)

$$\begin{aligned} AS &= T \begin{bmatrix} P & 0 \\ 0 & N \end{bmatrix} T^{-1} T \begin{bmatrix} I & 0 \\ 0 & -I \end{bmatrix} T^{-1} \\ &= T \begin{bmatrix} P & 0 \\ 0 & -N \end{bmatrix} T^{-1} \\ &= T \begin{bmatrix} I & 0 \\ 0 & -I \end{bmatrix} \begin{bmatrix} P & 0 \\ 0 & N \end{bmatrix} T^{-1} \\ &= T \begin{bmatrix} I & 0 \\ 0 & -I \end{bmatrix} \left(T^{-1}T\right) \begin{bmatrix} P & 0 \\ 0 & N \end{bmatrix} T^{-1} \\ &= SA. \ \square \end{aligned}$$

For the computation of the matrix sign function, Roberts proposed the Newton iteration

$$Y_{k+1} = \frac{1}{2}\left(Y_k + Y_k^{-1}\right), \quad Y_0 = A. \tag{4.4.3}$$

Iteration (4.4.3) converges globally and quadratically to $\text{sign}(A)$, and is derived by applying Newton's method to the equation $A^2 = I$.

Lemma 4.4.2 *If*

$$Y_{k+1} = \frac{1}{2}\left(Y_k + Y_k^{-1}\right), \quad Y_0 = A,$$

and $S = \text{sign}(A)$, *then* $Y_k S = S Y_k$ *for every* $k = 0, 1, \ldots$.

Proof. By induction. For $k = 0$ the request is true from Lemma (4.4.1)(iii). Assume that it is true for $k = n$. Then for $k = n + 1$,

$$Y_{k+1} S = \frac{1}{2}\left(Y_k + Y_k^{-1}\right) S$$

$$\begin{aligned}
&= \frac{1}{2}\left(Y_kS + (Y_kS)^{-1}\right) \\
&= \frac{1}{2}\left(SY_k + S^{-1}Y_k^{-1}\right) \\
&= \frac{1}{2}\left(SY_k + SY_k^{-1}\right) \\
&= S\left(\frac{1}{2}\left(Y_k + Y_k^{-1}\right)\right) \\
&= SY_{k+1}. \ \square
\end{aligned}$$

If α is a real number, then α can be written as $\alpha = \text{sign}(\alpha)\,|\alpha|$, where $\text{sign}(\alpha) = \pm 1$ is the familiar sign of a scalar. The polar decomposition can be viewed as a generalization of this scalar decomposition to complex matrices, with $\text{sign}(\alpha)$ replaced by the unitary polar factor, and $|\alpha|$ replaced by the Hermitian one. In [64], Higham points out that the matrix sign function can be also considered as a generalization of this scalar decomposition and based on this observation investigates the relation between the polar decomposition and the matrix sign function. For this purpose, Higham defines the *matrix sign decomposition*

$$A = SN, \quad S = \text{sign}(A), \quad A \in \mathbf{C}^{n\times n}.$$

As we mentioned earlier, $S = \text{sign}(A)$ is uniquely defined and so is the matrix sign decomposition. The matrix N is given by $N = S^{-1}A = SA$. ($S^{-1} = S$ from Lemma 4.4.1(ii)). The following lemma summarizes two elementary properties of the matrix N.

Lemma 4.4.3 *Let $A \in \mathbf{C}^{n\times n}$ have the matrix sign decomposition*

$$A = SN, \quad S = \text{sign}(A), \quad A \in \mathbf{C}^{n\times n}.$$

Then, the matrix N has the following properties:

(i) *N commutes with S ($SN = NS$).*

(ii) *The eigenvalues of N have strictly positive real part ($\text{Re}\ \lambda_i(N) > 0$).*

(iii) $N = (A^2)^{1/2}$

Proof. (i) Let

$$A = T\begin{bmatrix} P & 0 \\ 0 & W \end{bmatrix}T^{-1}$$

be the Jordan canonical form of A, and

$$\operatorname{sign}(A) = T \begin{bmatrix} I & 0 \\ 0 & -I \end{bmatrix} T^{-1}.$$

The matrices P and W are in block diagonal form and they contain the eigenvalues of A with positive and negative part respectively. Since $N = SA$, N can be written as

$$N = T \begin{bmatrix} P & 0 \\ 0 & -W \end{bmatrix} T^{-1},$$

and therefore

$$\begin{aligned} NS &= T \begin{bmatrix} P & 0 \\ 0 & -W \end{bmatrix} T^{-1} T \begin{bmatrix} I & 0 \\ 0 & -I \end{bmatrix} T^{-1} \\ &= T \begin{bmatrix} P & 0 \\ 0 & W \end{bmatrix} T^{-1} \\ &= A \\ &= SN. \end{aligned}$$

(ii) It is immediate since

$$N = T \begin{bmatrix} P & 0 \\ 0 & -W \end{bmatrix} T^{-1},$$

is the Jordan canonical form of N.

(iii) Inasmuch as $SN = NS$ and $S^2 = I$, $A^2 = SNSN = S^2N^2 = N^2$. Since A has no pure imaginary eigenvalues, A^2 is nonsingular and has no real, negative eigenvalues. Therefore (see [62, Theorem 4]), N can be uniquely given as $N = (A^2)^{1/2}$. □

Lemma 4.4.3(iii) suggests an alternative definition of the matrix sign function:

$$\operatorname{sign}(A) = A(A^2)^{-1/2}, \tag{4.4.4}$$

and thus the matrix sign decomposition can be written as:

$$A = \operatorname{sign}(A)(A^2)^{1/2}. \tag{4.4.5}$$

Matrix sign decomposition	$A = SN$	$S^2 = I$	$\lambda_i(S) = \pm 1$	Re $\lambda_i(N) > 0$
Polar decomposition	$A = UH$	$U^*U = I$	$\|\lambda_i(U)\| = 1$	$\lambda_i(H) > 0$

Table 4.4.1: Similarities between the polar decomposition and the matrix sign decomposition

Definition (4.4.4) is attributed to Higham [64] who first observed and investigated the similarities between the polar decomposition and the matrix sign function. Writing the polar decomposition of $A \in \mathbf{C}^{n\times n}$ as

$$A = UH = U(A^*A)^{1/2}, \tag{4.4.6}$$

the analogies between (4.4.4) and (4.4.6) can be easily observed in Table 4.4.1.

The following Lemma 4.4.4 shows clearly a direct link between the matrix sign function and the polar decomposition.

Lemma 4.4.4 *For any nonsingular $A \in \mathbf{C}^{n\times n}$,*

$$\operatorname{sign}\left(\begin{bmatrix} 0 & A \\ A^* & 0 \end{bmatrix}\right) = \left(\begin{bmatrix} 0 & U \\ U^* & 0 \end{bmatrix}\right).$$

Proof. Let $A = V\Sigma W^*$ be the singular value decomposition of A, and

$$T = \frac{\sqrt{2}}{2}\begin{bmatrix} V & -V \\ W & W \end{bmatrix}.$$

We observe that $\begin{bmatrix} 0 & A \\ A^* & 0 \end{bmatrix}$ can be written as

$$\begin{bmatrix} 0 & A \\ A^* & 0 \end{bmatrix} = T\begin{bmatrix} \Sigma & 0 \\ 0 & -\Sigma \end{bmatrix}T^*. \tag{4.4.7}$$

Since $T^* = T^{-1}$ (T is a unitary matrix), (4.4.7) is the Jordan canonical form of $\begin{bmatrix} 0 & A \\ A^* & 0 \end{bmatrix}$.

Therefore from the definition of the matrix sign function,

$$\operatorname{sign}(A) = T\begin{bmatrix} I & 0 \\ 0 & -I \end{bmatrix}T^* = \begin{bmatrix} 0 & VW^* \\ WV^* & 0 \end{bmatrix} = \begin{bmatrix} 0 & U \\ U^* & 0 \end{bmatrix}.$$

Lemma 4.4.4 can also be proved via the matrix sign decomposition

$$\operatorname{sign}\left(\begin{bmatrix} 0 & A \\ A^* & 0 \end{bmatrix}\right) = \begin{bmatrix} 0 & A \\ A^* & 0 \end{bmatrix}\left(\begin{bmatrix} 0 & A \\ A^* & 0 \end{bmatrix}^2\right)^{-1/2}.$$

It can be easily verified by direct calculations that

$$\left(\begin{bmatrix} 0 & A \\ A^* & 0 \end{bmatrix}^2\right)^{-1/2} = \begin{bmatrix} (AA^*)^{-1/2} & 0 \\ 0 & (A^*A)^{-1/2} \end{bmatrix},$$

and since $H = (A^*A)^{1/2}$,

$$\operatorname{sign}\left(\begin{bmatrix} 0 & A \\ A^* & 0 \end{bmatrix}\right) = \left(\begin{bmatrix} 0 & U \\ U^* & 0 \end{bmatrix}\right). \square$$

Lemma 4.4.4 allows us to derive various formulae and iterations for the matrix sign function from the corresponding ones for the polar decomposition. For example, if we apply the Newton iteration (4.4.3) to $\begin{bmatrix} 0 & A \\ A^* & 0 \end{bmatrix}$, we obtain the following Newton iteration for computing U:

$$Y_{k+1} = \frac{1}{2}\left(Y_k + Y_k^{-*}\right), \quad Y_0 = A \in \mathbf{C}^{n\times n}, \tag{4.4.8}$$

which is Higham's method (4.3.3).

In [96], Pandey, Kenney and Laub present a parallel algorithm for the matrix sign function. This work is based on previous work by Kenney and Laub [79], where a family of Padé iterations for the computation of sign(A) is derived. Their methods are based on the observation that for any nonzero real x,

$$\operatorname{sign}(x) = \frac{x}{|x|} = \frac{x}{\sqrt{1-(1-x^2)}}.$$

Setting $\xi = 1 - x^2$,

$$\operatorname{sign}(x) = \frac{x}{\sqrt{1-\xi}} = x(1-\xi)^{-1/2}.$$

$(1-\xi)^{-1/2}$ can be expressed via the hypergeometric function as

$$(1-\xi)^{-1/2} = {}_2F_1\left(\frac{1}{2}, 1, 1, \xi\right)$$

where ${}_2F_1\left(\frac{1}{2},1,1,\xi\right)$ belongs to the family of hypergeometric functions

$${}_2F_1(\alpha,\beta,\gamma,\xi)=\sum_{n=0}^{+\infty}\frac{(\alpha)_n(\beta)_n}{n!(\gamma)_n}\xi^n,$$

where $(a)_n=(a)(a+1)\dots(a+n-1)$ with $(a)_0=1$. ${}_2F_1\left(\frac{1}{2},1,1,\xi\right)$ can be approximated by Padé approximants (see for example [6]). Kenney and Laub exploit this observation and they derive iterations for computing $\mathrm{sign}(A)$ of the form

$$X_{k+1}=X_k\pi_r\left(X_k^2\right)\rho_s\left(X_k^2\right)^{-1},\quad X_0=A, \tag{4.4.9}$$

where π_r and ρ_s and polynomials of degree r and s respectively. In the same work, they show that for $r=s$ and $r=s-1$ these iterations converge globally to $\mathrm{sign}(A)$ with rate of convergence $r+s+1$.

In [96], Pandey, Kenney and Laub derive an explicit partial fraction form for the iteration (4.4.9) with $r=s-1$. The iteration is

$$X_{k+1}=\frac{1}{p}X_k\sum_{i=1}^{p}\frac{1}{\xi_i}\left(X_k^2+\alpha_i^2I\right)^{-1},\quad X_0=A, \tag{4.4.10}$$

where

$$\xi_i=\frac{1}{2}\left(1+\cos\left(\frac{(2i-1)\pi}{2p}\right)\right),\quad \alpha_i^2=\frac{1}{\xi_i}-1,\quad i=1:p. \tag{4.4.11}$$

This iteration converges globally to $\mathrm{sign}(A)$ with rate of convergence $2p$. Since each fraction can be evaluated on a separate processor in parallel, iteration (4.4.10) is suitable for parallel computation. Substituting

$$X_k:=\begin{bmatrix}0 & X_k\\ X_k^* & 0\end{bmatrix} \tag{4.4.12}$$

into (4.4.10), we find that X_{k+1} has the same block 2-by-2 form (4.4.12) as X_k. Thus, equating $(1,2)$ blocks on both sides we obtain, using Lemma 4.4.4, the iteration

$$X_{k+1}=\frac{1}{p}X_k\sum_{i=1}^{p}\frac{1}{\xi_i}\left(X_k^*X_k+\alpha_i^2I\right)^{-1},\quad X_0=A. \tag{4.4.13}$$

Being aware of the link between the matrix sign function and the polar decomposition as established in Lemma 4.4.4, we conclude that for any full rank matrix $A\in\mathbf{C}^{m\times n}$,

iteration (4.4.13) converges globally to the unitary polar factor U of A with rate of convergence $2p$. Iteration (4.4.13) is derived by Higham [64] in the way described here.

Finally, it is useful to observe the relationship between the Newton iteration (4.4.8) and iteration (4.4.13). For $p = 1$, iteration (4.4.13) is

$$X_{k+1} = 2X_k(X_k^* X_k + I)^{-1}. \tag{4.4.14}$$

If $A \in \mathbf{C}^{n\times n}$, nonsingular, and

$$Y_{k+1} = \frac{1}{2}\left(Y_k + Y_k^{-*}\right), \quad Y_0 = A,$$

then

$$\begin{aligned} Y_{k+1}^{-*} &= \left(\frac{1}{2}\left(Y_k + Y_k^{-*}\right)^*\right)^{-1} \\ &= \left(\frac{1}{2}\left(Y_k^* + Y_k^{-1}\right)\right)^{-1} \\ &= 2\left(\left(Y_k^* Y_k + I\right) Y_k^{-1}\right)^{-1} \\ &= 2Y_k \left(Y_k^* Y_k + I\right)^{-1}. \end{aligned} \tag{4.4.15}$$

Equation (4.4.15) indicates the relationship between iteration (4.4.13) and iteration (4.4.14). If we set $Y_k = X_k$ in (4.4.15) then

$$X_{k+1} = Y_{k+1}^{-*}. \tag{4.4.16}$$

This result can be generalized for any p which is a power of 2. In this case, iteration (4.4.13) can be derived by combining $\log_2 p + 1$ steps of (4.4.8) and expressing the result in partial fraction form. Namely, if we set $X_k = Y_k$ in (4.4.13) as above, then

$$X_{k+1} = Y_{k+\log_2 p+1}^{-*}, \tag{4.4.17}$$

and therefore one step of (4.4.13) is equivalent to $\log_2 p + 1$ Newton iterates.

4.5 A Parallel Algorithm for the Polar Decomposition

Iteration (4.4.13) is endowed with inherent high-level parallelism. Apart from the p matrix inversions, which can be carried out simultaneously at every step, iteration (4.4.13)

is also rich in matrix–matrix multiplications. Matrix–matrix multiplications can also be performed in parallel in a very efficient way on most parallel computers. Implementation details are discussed in the next section.

As we mentioned in Section 3, the convergence of the Higham's iteration (4.4.8) can be accelerated by scaling the iterates $X \leftarrow \gamma X_k$, where γ_k is either the optimum scaling factor

$$\gamma_k = \left(\sigma_{\max}\left(Y_k\right)\sigma_{\min}\left(Y_k\right)\right)^{-1/2}, \tag{4.5.1}$$

or its computationally less expensive estimate

$$\widehat{\gamma}_k = \left(\frac{\|Y_k^{-1}\|_1\|Y_k^{-1}\|_\infty}{\|Y_k\|_1\|Y_k\|_\infty}\right)^{1/4}. \tag{4.5.2}$$

In [80], Kenney and Laub present an in–depth analysis of scaling the Newton iteration. A notable result in this work is that with the optimal scaling parameter (4.5.1), $Y_k = U$ where k is the number of distinct singular values of A, that is, exact convergence is obtained in $k \leq n$ iterations. The computation of the optimal scaling parameter (4.5.1) is expensive because it requires the singular value decomposition of Y_k at each step. The computation of the scaling parameter (4.5.2) is less expensive, and it can also be performed in parallel as shown in the following section. Practical experience shows that even using the approximate scaling parameter $\widehat{\gamma}_k$, convergence results are satisfactory, and convergence is almost always obtained in ten iterations or less (to a convergence tolerance of 10^{-16} or greater).

The relation between Higham's method and iteration (4.4.13) suggests the application of the above scaling policy to iteration (4.4.13). Its scaled version can be written as

$$X_{k+1} = \frac{1}{p}\mu_k X_k \sum_{i=1}^{p} \frac{1}{\xi_i}\left(\mu_k^2 X_k^* X_k + \alpha_i^2 I\right)^{-1}, \quad X_0 = A \in \mathbf{C}^{m\times n}, \tag{4.5.3}$$

where μ_k is given by (4.5.1) or (4.5.2). The effect of scaling is the same as if we scaled only one in $\log_2 p + 1$ Higham iterates since as we mentioned above, one step of (4.4.13) is equivalent to $\log_2 p + 1$ Higham steps. Hence, the increased opportunity for parallelism causes a decline in the effectiveness of scaling. Futhermore, the computation of the scaling parameter μ_k requires the inversion of X_k. This is an extra cost, since we do not form X_k^{-1} during the course of iteration.

Lemma 4.5.1 indicates a worthy of note advantage of (4.5.3) over Higham's method: having necessarily formed $X_k^* X_k$ in the beginning of each iteration, we can evaluate cheaply the upper bound in the inequality

$$\|X_k - U\|_F \leq \|X_k^* X_k - I\|_F, \quad A = UH,$$

and thus terminate the iteration in a reliable and timely fashion. Note that we are using here the fact that each X_k has the same polar factor U as A.

Lemma 4.5.1 *Let $A \in \mathbf{C}^{m\times n}$ have the polar decomposition $A = UH$. Then*

$$\frac{\|A^*A - I\|_F}{1 + \|A\|_2} \leq \|A - U\|_F \leq \|A^*A - I\|_F. \tag{4.5.4}$$

Proof. Let $A = P\Sigma Q^*$ be the singular value decomposition of A, so that $U = PQ^*$ is its unitary polar factor. Substituting A and U in (4.5.4),

$$\|A - U\|_F = \|P\Sigma Q^* - PQ^*\|_F = \|\Sigma - I\|_F,$$

and

$$\|A^*A - I\|_F = \|Q\Sigma P^* P\Sigma Q^* - QQ^*\|_F = \|\Sigma^2 - I\|_F.$$

It is a well-known result, that for every matrix $D \in \mathbf{C}^{m\times n}$ and $F \in \mathbf{C}^{n\times m}$, $\|DF\|_F \leq \|D\|_F \|F\|_2$ [67]. Thus,

$$\|\Sigma^2 - I\|_F \leq \|\Sigma - I\|_F \|\Sigma + I\|_2,$$

and since

$$\|\Sigma + I\|_2 \leq 1 + \|A\|_2,$$

it follows that the left inequality in (4.5.4 holds.

For the right inequality in (4.5.4) it suffices to show that $\|\Sigma - I\|_F \leq \|\Sigma^2 - I\|_F$, or equivalently,

$$(\sigma_i - 1)^2 \leq (\sigma_i^2 - 1)^2 = (\sigma_i - 1)^2(\sigma_i + 1)^2, \quad \text{for every} \quad i = 1, \ldots, n. \tag{4.5.5}$$

Since $\sigma_i \geq 0$ for every $i = 1, \ldots, n$, (4.5.5) always holds. □

The presense of$X_k^* X_k$ is also a drawback since by forming $X_k^* X_k$ numerically one can lose information on X_k. The above mentioned disanvantages of iteration (4.5.3) suggest

that the iteration is potentially numerically unstable when A is ill conditioned. This is not a totally unexpected observation, since there are many examples in numerical linear algebra where there is a tradeoff between parallelism and numerical stability [31].

To examine the stability we need a tool to measure the quality of an approximate polar factor. We assume for the rest of this section that A is a square matrix (certain of the inequalities below do not hold for rectangular A.) Given a unitary approximation V to the unitary polar factor U of A, we are interested in the nearest perturbed matrix to A, $A+E$, whose exact unitary polar factor is the matrix V. This thought leads naturally to the following definition of the backward error of V:

$$\begin{aligned} \beta(V) &= \min\{\|E\|_F : \ V \text{ is the unitary polar factor of } A+E\} \\ &= \min\{\|E\|_F : \ V^*(A+E) \text{ is Hermitian positive semidefinite}\}. \end{aligned}$$

Lemma 4.5.2 indicates how we can evaluate $\beta(V)$, or at least an lower bound for it.

Lemma 4.5.2 *Let $A \in \mathbf{C}^{m\times n}$ and let $U \in \mathbf{C}^{m\times n}$ be unitary. Then*

$$\min\{\, \|E\|_F : V^*(A+E) \textit{ is Hermitian}\} = \frac{1}{2}\|A^*V - V^*A\|_F$$

and the minimum is achieved for $E_{\text{opt}} = \frac{1}{2}(VA^*V - A)$. If $V^*(A+E_{\text{opt}}) = \frac{1}{2}(A^*V + V^*A)$ is positive semidefinite then E_{opt} solves

$$\min\{\, \|E\|_F : V^*(A+E) \textit{ is Hermitian positive semidefinite}\}.$$

Proof. Any matrix $B \in \mathbf{C}^{n\times n}$ can be written as

$$B = \frac{1}{2}(B+B^*) + \frac{1}{2}(B-B^*),$$

where $\frac{1}{2}(B+B^*)$ is the Hermitian part of B and $\frac{1}{2}(B-B^*)$ is the skew-Hermitian part of B. If $V^*E = G+K$, where $G = G^*$ and $K = -K^*$, then

$$V^*(A+E) = V^*A + V^*E = V^*A + G + K.$$

The requirement that $V^*(A+E)$ is Hermitian implies that

$$2K = A^*V - V^*A,$$

and therefore

$$V^*E = G + \frac{1}{2}(A^*V - V^*A).$$

Hence

$$\|E\|_F^2 = \|V^*E\|_F^2 = \|G\|_F^2 + \tfrac{1}{4}\|A^*V - V^*A\|_F^2,$$

and since G is arbitrary, the minimizing E is achieved when $G = 0$. This establishes the first part. The second part is immediate since the set of allowable E in the second minimization problem is a subset of the first. □

Lemma 4.5.2 states that given a unitary approximation V to U, the best choice for H is

$$H = \frac{1}{2}\left(A^*V + V^*A\right),$$

assuming that H is positive semidefinite. Moreover, if H is positive semidefinite, the backward error of V is given by

$$\beta(V) = \frac{1}{2}\|A^*V + V^*A)\|_F;$$

otherwise the quantity $\frac{1}{2}\|A^*V + V^*A)\|_F$ is a lower bound of $\beta(V)$, that is

$$\beta(V) > \frac{1}{2}\|A^*V + V^*A)\|_F.$$

We can also observe that if $A, V \in \mathbf{C}^{n\times n}$ then

$$\begin{aligned} A - VH &= A - V\left(\frac{1}{2}\left(A^*V + V^*A\right)\right) \\ &= A - \frac{1}{2}VA^*V - \frac{1}{2}A \\ &= \frac{1}{2}\left(A - VA^*V\right), \end{aligned}$$

and hence

$$\|A - VH\|_F = \frac{1}{2}\|A^*V - V^*A\|_F. \tag{4.5.6}$$

Equation (4.5.6) shows that $\beta(V)$ is always greater than or equal to the residual of the approximate polar decomposition.

A computed approximation $\widehat{U}$ to U is usually unitary only to within roundoff, but with $V = \widehat{U}$ all the inequalities in Lemma 4.5.2 are still true to within roundoff. It is

therefore reasonable to use the following formula for the computation of the Hermitian polar factor

$$\widehat{H} = \frac{(\widehat{U}^* A)^* + \widehat{U}^* A}{2}. \tag{4.5.7}$$

Formula (4.5.7) does not suggest any innovation in computing the Hermitian polar factor. This choice has been used in [61, 66]. However the present justification is independent and new.

The stability properties of (4.5.3) are clearly illustrated by its performance on the 10×10 Vandermonde matrix A, with

$$a_{ij} = \left(\frac{j-1}{n-1}\right)^{i-1}, \quad n = 10.$$

This matrix has a 2–norm condition number $\kappa_2(A) = 1.52 \times 10^{-7}$. All computations were done using MATLAB on a Sun Sparc workstation with unit roundoff $u \approx 1.1 \times 10^{-16}$. To investigate the behaviour of iteration (4.5.3), we implemented the iteration both with and without scaling. The approximate scaling factor (4.5.2) (with Y_k replaced by X_k) has been used for the scaled version but scaling is done only while $\|X_k^* X_k - I\|_F > 10^{-2}$. The reason is that after this point convergence is fast, and there is no need for extra computational cost caused by the forming of the scaling factor. In the unscaled iteration we scaled the original matrix to have unit Frobenius norm, for reasons explained later in this section. Iterations are terminated when $\|X_k^* X_k - I\|_F \leq nu$. Table 4.5.1 summarizes some characteristic results for the behaviour of iteration (4.5.3). It also displays results for both the unscaled and scaled Newton iteration for comparison. The Hermitian polar factor has been computed using the formula (4.5.7) and it is positive semidefinite; thus we can evaluate $\beta(\widehat{U})$.

Table 4.5.1 shows that the scaled version of iteration (4.5.3) is unstable for all p, since the backward error is seven order of magnitude larger than we would like. This behaviour is explained by the fact that some of the matrices $C_i = \mu_k^2 X_k^* X_k + A_i^2 I$ are ill conditioned, as illustrated in the fourth column of Table 4.5.1. (This column reveals the maximum 2–norm condition number over all terms i and iterations k). These ill conditioned terms appear when $\mu_k X_k$ is ill conditioned and has norm greatly exceeded 1. The fact that C_i is symmetric positive definite allows us to express $k_2(C_i)$ in terms of the singular values

Iteration	Scaling	Iters	$\beta(\widehat{U})/\|A\|_F$	$\max_{i,k} \kappa_2(\mu_k^2 X_k^* X_k + \alpha_i^2 I)$
Newton	unscaled	29	8.10E-12	–
	scaled	8	4.10E-16	–
$p = 1$	unscaled	29	3.15E-16	2.00E+00
	scaled	8	1.37E-09	9.21E+06
$p = 2$	unscaled	15	5.74E-16	6.83E+00
	scaled	5	6.95E-09	5.37E+07
$p = 4$	unscaled	10	1.22E-15	2.63E+01
	scaled	4	2.40E-09	2.33E+08
$p = 8$	unscaled	8	2.22E-15	1.04E+02
	scaled	4	6.00E-09	9.49E+08
$p = 16$	unscaled	6	9.64E-15	4.15E+02
	scaled	3	3.60E-09	3.82E+09

Table 4.5.1: Behaviour on a 10×10 Vandermonde matrix.

of X_k, that is

$$\kappa_2(C_i) = \frac{(\mu_k \sigma_{\max}(X_k))^2 + \alpha_i^2}{(\mu_k \sigma_{\min}(X_k))^2 + \alpha_i^2} \le \frac{(\mu_k \sigma_{\max}(X_k))^2}{\min_i \alpha_i^2} + 1. \tag{4.5.8}$$

For the scaled iteration the best bound that holds for all k is $\|\mu_k X_k\|_2 \le \kappa_2(A)^{1/2}$ (with equality when $k = 0$), and so the best bound for $\kappa_2(C_i)$ is of order $\kappa_2(A)$. It is apparent that if A is ill conditioned, $\kappa_2(C_i)$ can be large.

Without using scaling, $\kappa_2(C_i)$ is nicely bounded, provided $\|X_0\|_2$ is not too large, because then all the X_k are not large in norm. This follows from the observation that with no scaling $\sigma_{\max}(X_k) \le 1$ for all $k \ge 1$, which is clear for iteration (4.4.16) and therefore follows for iteration (4.4.10). If $p = 16$ then $\min_i \alpha_i^2 = 2.41 \times 10^{-3}$ and (4.5.8) (with $\mu_k \equiv 1$) yields $\kappa_2(C_i) \le 416$ for all i and all $k \ge 1$; this is a sharp inequality, as can be seen from the $p = 16$ entry in Table 4.5.1.

The initial scaling $X_0 \leftarrow X_0/\|X_0\|_F$ ensures that $\sigma_{\max}(X_0) \le 1$. When X_0 is nearly unitary this scaling tends to increase $\|X_0 - U\|_F$, since $\|Q\|_F = \sqrt{n}$ for unitary Q. Alternative initial scalings that do not have this disadvantage are $X_0 \leftarrow (\|X_0\|_F/\|X_0^T X_0\|_F)X_0$ (which solves the minimization problem $\min_\alpha \|I - (\alpha X_0)^*(\alpha X_0)\|_F$), $X_0 \leftarrow \sqrt{n}X_0/\|X_0\|_F$, or simply no scaling. The unit Frobenius norm scaling has the advantage of giving the smallest bound for $\kappa_2(C_i)$ when $k = 0$. In our implementation we do not apply initial scaling to matrices that are close to orthogonal, that is matrices with 2-norm condition

number less than 1.01.

It is clear that there is a tradeoff between stability and scaling. Scaling can drastically improve the speed of convergence but on the other hand it can seriously effect the stability of (4.5.3) when A is ill conditioned; ironically this is the situation where scaling is most necessary. On the contrary, Higham's method with scaling is always almost stable in practice [61]. But the instability of (4.5.3) does not effect its practical functionality for two reasons. First, as we will see in Section 8, in certain applications A is nearly unitary and therefore well conditioned. Second, a certain level of instability may be tolerated, when the polar decomposition is not demanded to high accuracy.

Our parallel algortithm for computing the polar decomposition can be described as follows.

Algorithm Parallel Polar.

Given a nonsingular matrix $A \in \mathbf{C}^{n\times n}$ and a convergence tolerance tol, this algorithm computes the polar decomposition $A = UH$. The algorithm uses iteration (4.5.3) and requires p processors.

$X := A$

if "scaling is not required", $X := X/\|X\|_F$, end

Compute the coefficients ξ_i, α_i^2 $(i = 1, \ldots, p)$ in (4.4.11).

repeat

(1) Compute $C := X^*X$ using parallel matrix multiply.

$\rho := \|C - I\|_F$

if $\rho \leq$ tol, goto (5), end

if $\rho \leq 10^{-2}$ and "scaling is required"

(2) Compute $W := C^{-1}$.

$\mu := ((\|W\|_\infty \|W\|_1)\,(\|X\|_\infty \|X\|_1))^{1/4}$

else

$\mu := 1$

end if

Form $C_i := \mu^2 C + \alpha_i^2 I$ on each processor i, $(i = 1, \ldots, p)$.

(3) Compute $T_i := C_i^{-1}$ on each processor i, $(i = 1, \ldots, p)$.

$S := \sum_{i=1}^{p} T_i$.

(4) Compute $X := \frac{\mu}{p} X S$, using parallel matrix multiply.

end

(5) $U := X$

Compute $H_1 := U^* A$ using parallel matrix multiply.

$H := \frac{1}{2}(H_1^* + H_1)$ ("best H" by Lemma 4.5.2).

4.6 Implementation on the KSR1

We coded the algorithm *Parallel Polar* in KSR Fortran. As we mentioned in Chapter 2, KSR Fortran provides high-level and low-level facilities for defining parallel programs. From the high-level ones we used parallel regions and tile families. Parallel sections, the third high-level parallel construct, has not been used in our implementation. We also used the same team of pthreads for all the parallel constructs in our code. The source code for Algorithm Parallel Polar is given in Appendix A.

For the matrix–matrix multiplications we used the highly optimized level 3 BLAS routine `SGEMM`. This routine is supplied by Kendall Square Research in the KSRlib/BLAS Library [77]. `SGEMM` superseded our implementation in KSR Fortran of Algorithm 6.3.2 in [47] for parallel matrix multiplication on a shared–memory computer. A non–standard call of `SGEMM` is the following:

```
CALL SGEMM(TRANSA, TRANSB, M, N, L, ALPHA, A, LDA, B, LDB,
&           BETA, C, LDC, CT, NCELL, TEAMID, PS)
```

The additional parameters `CT`, `NCELL`, `TEAMID`, and `PS` are utilized to increase performance. `CT` is a temporary array allocated by the user, `NCELL` is the number of processors, `TEAMID` is the identification number of the team of threads, and `PS` is a parallel partitioning configuration parameter. The meaning of `PS` is the following: matrix C is partitioned

		Matrix size n		
		256	512	1024
1 proc	(PS = 1)	23	25	27
2 proc	(PS = 1)	43	48	53
4 proc	(PS = 2)	73	82	92
8 proc	(PS = 3)	112	119	139
16 proc	(PS = 4)	197	244	302

Table 4.6.1: Timing results for SGEMM (Mflops) for the default value of PS.

		Matrix size n		
		256	512	1024
1 proc	(PS = 1)	1.00	1.00	1.00
2 proc	(PS = 1)	1.85	1.91	1.94
4 proc	(PS = 2)	3.13	3.27	3.35
8 proc	(PS = 3)	4.80	4.96	5.06
16 proc	(PS = 4)	8.47	9.75	11.00

Table 4.6.2: Speedups for SGEMM for the default value of PS.

in `PS` "horizontal" strips of rows, each row being treated by different set of processors. Tables 4.6.1 and 4.6.2 give the timing results and the speedups respectively for various numbers of processors and matrix sizes for the default value of `PS`, that is

```
PS = INT(SQRT(FLOAT(NCELL*M/N)) + 0.5).
```

`PS` can be set to any integer value between 1 and the number of processors. For square matrices of order 1024, the best performance of `SGEMM` has been observed when `PS` has been set equal to 1. Tables 4.6.3 and 4.6.4 illustrate the better performance of `SGEMM` in both megaflops and speedups, when `PS` has been set to 1 instead of its default value. It is remarkable that for matrices of order 1024 there is a performance increase of about 27%. We also observe in Table 4.6.3 that `SGEMM` runs at 384 megaflops on 16 processors for a matrix of order 1024, which is over half the peak megaflop rate.

The matrices $C_i = \mu_k^2 X_k^* X_k + \alpha_i^2 I$ in (4.5.3) are symmetric positive definite. For their inversion at step (3) of the algorithm, we used the LAPACK routines `SPOTRF` and `SPOTRI` from the KSRlib/LAPACK Library [78]. During our numerical experiments we did not observe any difference in the timing results using either the KSRlib/LAPACK Library

	Matrix size n		
	256	512	1024
1 proc	23	25	27
2 proc	43	48	53
4 proc	75	93	103
8 proc	129	170	206
16 proc	210	261	384

Table 4.6.3: Timing results for SGEMM (Mflops) with PS = 1.

	Matrix size n		
	256	512	1024
1 proc	1.00	1.00	1.00
2 proc	1.85	1.91	1.94
4 proc	3.17	3.65	3.77
8 proc	5.49	6.67	7.49
16 proc	8.93	10.23	14.00

Table 4.6.4: Speedups for SGEMM with PS = 1.

or the standard LAPACK distribution [1]. `SPOTRF` computes the Cholesky factorization of a real symmetric positive definite matrix, while `SPOTRI` computes its inverse using the Cholesky factorization. The inversion of the matrix X at step (2) is necessary only when scaling is required. We do the inversion in parallel using the LAPACK routine `SGETRF` to compute the LU factorization of the general matrix X, and `SGETRI` to compute its inverse using the LU factors. `SGETRF` is the right-looking level 3 BLAS version of the blocked LU algorithm. The parallelization of `SGETRF` and `SGETRI` has been achieved by introducing parallel processing at the level 3 BLAS layer. It is therefore desirable to have level 3 BLAS operations on biggest possible matrices. This can be achieved by changing the block size, which is otherwise determined by the enviromental enquiry routine `ILAENV`. We experimented with the block size in the LAPACK routines mentioned above and found that a block size of 16 gives the best all-round performance on the KSR1. Therefore all our results are for a block size of 16. Table 4.6.5 summarizes the timing results for the matrix inversion for various number of processors and matrix sizes, when parallel processing is being exploited at the level 3 BLAS layer only. Figure 4.6.1 illustrates the

	Matrix size n		
	256	512	1024
proc 1	10.7	12.3	14.9
proc 2	11.6	15.4	21.2
proc 4	13.2	18.5	27.3
proc 8	13.8	20.5	31.8
proc 16	14.4	22.6	39.3

Table 4.6.5: Timing results for SGETRF/SGETRI (Mflops)

Figure 4.6.1: SGETRF/SGETRI performance for the KSR1.

same timing results.

We formed X^*X in step (1) of the algorithm using `SGEMM` because we found this to be faster in parallel, for the KSRlib/BLAS Library, than `SSYRK` (which takes advantage of symmetry but is not highly optimized in this library). Consequently, our implementation of Algorithm Parallel Polar does not fully exploit the symmetry of X^*X.

As we mentioned in Chapter 2, loop parallelization in KSR Fortran is achieved by tiling in which execution of a single do loop is transformed into parallel execution of multiple tiles, or groups of loop iterations. In our first codes that we wrote for Algorithm Parallel Polar, we used semi-automatic tiling. Later we experimented using manual tiling

and in some instances we achieved slightly better timing results. In the source code for Algorithm Parallel Polar given in Appendix A, we use manual tiling. Tiling has been avoided for small loops, as for example in the evaluation of the coefficients ξ_i and α_i^2, because the startup cost outweighs the gain.

For the simultaneous inversions in step (3) of the algorithm Parallel Polar, we used parallel regions. All the 2-dimensional arrays have been declared as N-by-$N+2$ in the main program, for the reasons explained in Section 4 in Chapter 4. We found that there was a significant improvement in the running times of our codes when we used $N+2$ instead of N.

4.7 Experimental Results

We compared the performance of three methods for computing the polar decomposition on the KSR1.

(1) The SVD method that first computes the SVD and then forms U and H according to (4.3.1). We compute the SVD using LAPACK's routine `SGESVD` using one processor. We found that the run time was larger when we used more than one processor with fully automatic parallelization by the compiler; this is apparently because the compiler parallelizes every DO loop and the startup costs for the small DO loops are significant.

(2) Algorithm Parallel Polar for $p > 1$, both with and without scaling.

(3) The Newton iteration (4.3.3) with and without scaling, with the inversions done in parallel using LAPACK's `SGETRF` and `SGETRI` and with H computed as in Algorithm Parallel Polar. Scaling is done only while $\|Y_{k+1} - Y_k\|_F > 10^{-2}\|Y_k\|_F$.

We used real matrices of dimension up to 1024, and all the results presented here are for this maximum dimension. For each dimension we generated random matrices $A = PDQ^T$, where P and Q are random orthogonal matrices and $D = \mathrm{diag}(\sigma_i)$, with exponentially distributed singular values $\sigma_i = \alpha^i$ $(0 < \alpha < 1)$, so that $\kappa_2(A) = \alpha^{1-n}$. We chose a range of condition numbers $\kappa_2(A) = 1.01, 10, 10^4, 10^8$ and 10^{12}, to observe the effect of the condition number on the behaviour of the iterative methods. The matrices with $\kappa_2(A) = 1.01$ are such that $\|A - U\|_2 \approx 0.01$, so they are moderately close to being

$\kappa_2(A)$	Time	$\beta(\widehat{U})/\|A\|_F$	Iters	Speedup	Scalings
1.01E0	204.82	4.3E-16	1	7.58	OFF
1.0E01	580.36	9.3E-16	3	7.62	OFF
1.0E04	1130.03	5.3E-15	6	7.69	OFF
1.0E08	1767.03	1.0E-14	9	7.74	OFF
1.0E12	2248.80	1.4E-14	12	7.78	OFF
1.01E0	286.34	4.3E-16	1	6.71	1
1.0E01	491.11	8.8E-16	2	6.59	1
1.0E04	775.92	3.1E-13	3	6.21	2
1.0E08	1085.83	2.4E-09	4	5.93	3
1.0E12	1124.59	2.0E-05	4	5.81	4

Table 4.7.1: Algorithm Parallel Polar with 8 processors, $n = 1024$.

orthogonal. We used the convergence tolerance tol $= nu$, where $u \approx 1.1 \times 10^{-16}$ is the unit roundoff for single precision arithmetic on the KSR1.

All the results reported were obtained using version 1.0 (March 1993) of the KSR Fortran compiler.

The timings for the SVD method are all between 9400 and 9500 seconds; as expected the condition number has little effect on the times.

Timings (in seconds) for Algorithm Parallel Polar with $p = 8$ and 16 processors are given in Tables 4.7.1 and 4.7.2. (We did not have access to all 32 processors of our KSR1 configuration). The speedup is defined as the time for iteration (4.5.3) with a given value of p implemented on a single processor divided by the time for Algorithm Parallel Polar implemented on p processors. The column headed "scalings" indicates either that no scaling was attempted ("OFF"), or states the number of iterations on which scaling was used (which is the number of times the inversion step (2) of the algorithm is executed). Timings for the Newton iteration on 16 processors are given in Table 4.7.3; the speedup is the run time for the iteration with 16 processors divided by the run time for the same iteration with 1 processor.

We make several observations.

(1) Algorithm Parallel Polar with $p = 8$ or 16 is between 8 and 51 times faster than the SVD approach, the greater speedups being for well-conditioned matrices. It is also between two and four times faster than the parallel implementation of the Newton

$\kappa_2(A)$	Time	$\beta(\widehat{U})/\|A\|_F$	Iters	Speedup	Scalings
1.01E0	184.17	4.3E-16	1	15.15	OFF
1.0E01	425.26	8.1E-16	2	15.26	OFF
1.0E04	1025.48	8.3E-15	5	15.39	OFF
1.0E08	1320.96	1.9E-14	7	15.42	OFF
1.0E12	1868.04	2.6E-14	10	15.48	OFF
1.01E0	243.08	2.7E-16	1	13.28	1
1.0E01	554.07	8.2E-16	2	13.19	1
1.0E04	647.08	3.2E-13	3	12.79	2
1.0E08	717.82	2.3E-09	3	11.82	3
1.0E12	1144.05	2.0E-05	5	11.56	3

Table 4.7.2: Algorithm Parallel Polar with 16 processors, $n = 1024$.

$\kappa_2(A)$	Time	$\beta(\widehat{U})/\|A\|_F$	Iters	Speedup	Scalings
1.01E0	1862.22	3.4E-14	10	2.28	OFF
1.0E01	2356.81	5.4E-14	13	2.28	OFF
1.0E04	3739.51	2.6E-12	21	2.29	OFF
1.0E08	6262.17	6.8E-09	34	2.29	OFF
1.0E12	8500.59	3.2E-05	47	2.30	OFF
1.01E0	745.23	2.3E-14	4	2.30	1
1.0E01	1133.31	3.4E-14	6	2.30	4
1.0E04	1490.46	3.3E-14	8	2.31	5
1.0E08	1664.81	3.2E-14	9	2.31	6
1.0E12	1862.53	3.2E-14	10	2.32	7

Table 4.7.3: Newton iteration with 16 processors, $n = 1024$.

iteration (4.3.3) when both iterations are scaled. The speed is a consequence of the high-level parallelism and the implementation's richness in level 3 BLAS operations. For the first entry in Table 4.7.2 ($p = 16$, $\kappa_2(A) = 1.01$) our code is running at a speed of about 140 megaflops.

(2) The speedups for Algorithm Parallel Polar without scaling are close to optimal. When scaling is used, the inversions at step (2) of the algorithm degrade the speedup because matrix inversion using LAPACK's `SGETRF` and `SGETRI` is not highly parallel (we measured an execution rate of only 39.3 megaflops for inversion of a matrix of size 1024 on 16 processors). Similarly, the speedups for the Newton iteration are poor because the iteration is dominated by matrix inversions.

(3) The backward errors for Algorithm Parallel Polar are as predicted by the analysis in Section 2. The maximum backward error without scaling is of order 10^2u, which is consistent with the convergence tolerance tol $= nu = 1024u$ and the bound $\max_{i,k} \kappa_2(C_i) \leq 416$ mentioned in Section 2. The backward errors with scaling are all approximately $\kappa_2(A)u/5$.

(4) Algorithm Parallel Polar takes good advantage of well-conditioned or nearly orthogonal A, requiring only one iteration if A is sufficiently close to being orthogonal.

(5) Increasing p beyond 16 in iteration (4.5.3) produces diminishing improvements in the number of iterations, particularly for well-conditioned A. If more than 16 processors are available (and the KSR1 supports up to 1088 processors) the most promising way to improve our timing results is to use more than one processor to execute each of the parallel segments of Algorithm Parallel Polar. For example, the simultaneous Cholesky factorizations would be executed faster if we implemented the parallel algorithm of George, Heath and Liu for computing the Cholesky factorization on a shared memory computer [44].

4.8 Applications of the Polar Decomposition

The polar decomposition plays a key role in the solution of various problems which arise in a wide range of applications. In this section we discuss some representative applications from disparate scientific areas. An in-depth analysis of these applications is beyond the

scope of this section. However, our aim is to show why the polar decomposition can be considered as one of the most important tools provided by linear algebra to applied sciences.

We start our discussion with an application encountered in aerospace computations [8, 13]. The motion of a missile can be described by a system of differential equations. Solving this system by a step-wise procedure, we obtain the direction cosine matrix (DCM). The DCM is a matrix of transformations between an orthogonal axis system fixed with respect to the missile and the inertial system. The DCM is widely used in aerospace science and technology for purposes such as navigation, attitude control and simulation. Having set carefully the initial conditions, the DCM is orthogonal at the outset of computations. But since the DCM has to be updated at every time step, unavoidable computational errors may compel the DCM to drift away from orthogonality after a number of steps. Since orthogonality is a certain requirement for a proper DCM, it is reasonable to adjust the erroneous DCM after a fixed small number of steps. This can be accomplished by replacing the erroneous matrix by its closest orthogonal matrix in the Frobenius norm, that is, its unitary polar factor.

The polar decomposition can also be used in estimating the attitude of a satellite [56, 121]. The problem can be mathematically expressed as follows: Given two sets of real vectors $\{v_1, \ldots, v_n\}$, and $\{u_1, \ldots, u_n\}$ $(n \geq 2)$ find the orthogonal matrix T with determinant $+1$ (i.e. the rotation matrix), which brings the first set into the best least squares coincidence with the second. $\{u_i\}$ are the direction cosines of objects as observed in a satellite fixed frame of reference, and $\{v_i\}$ are the direction cosines of the same objects in a known frame of reference. The required rotation matrix T, which carries the known frame of reference into the satellite fixed frame of reference, is the solution to the minimization problem

$$\min_{T^*T=I,\ \det(T)=+1} \|V - TU\|_F, \tag{4.8.1}$$

where the columns of V and U are the vectors $\{v_i\}$ and $\{u_i\}$ respectively. The minimization problem (4.8.1) is equivalent to finding the rotation matrix T that maximizes $\mathrm{tr}(T^*UV^*)$. The solution of the problem requires at this stage the polar decomposition

of UV^*.

The Löwdin problem arises in quantum chemistry [45]. Here, a basis of unit vectors $\{v_1, \ldots, v_n\}$ for $\mathbf{C}^n$ is given, and an orthogonalization of this basis is required for computational and theoretical purposes. These vectors contain valuable information from the known subsystems and their alteration results in loss of information. Thus, the aim is to orthogonalize the $\{v_1, \ldots, v_n\}$, changing the vectors as little as possible. If $\{e_1, \ldots, e_n\}$ is an orthonormal basis for $\mathbf{C}^n$, the problem can be stated as follows: Find a matrix $B \in \mathbf{C}^{n\times n}$ such that $Bv_i = e_i$, to minimize the least squares deviation

$$\sum_{i=1}^{n} \|v_i - e_i\|^2.$$

If $B = UH$ is the polar decomposition of the invertible matrix B, then

$$\begin{aligned} \sum_{i=1}^{n} \|v_i - e_i\|^2 &= \sum_{i=1}^{n} \|B^{-1}e_i - e_i\|^2 \\ &= \sum_{i=1}^{n} \|(H^{-1} - U)U^{-1}e_i\|^2 \\ &= \|H^{-1} - U\|_F^2, \end{aligned} \tag{4.8.2}$$

since $\|A\|_F^2 = \sum_{i=1}^{n} \|Ag_i\|^2$ for any $A \in \mathbf{C}^{n\times n}$ and $\{g_1, \ldots, g_n\}$ orthonormal basis for $\mathbf{C}^n$. Goldstein and Levy [45] show that all the orthogonalization matrices have the same Hermitian factor and therefore the problem is equivalent to finding unitary matrix U to minimize (4.8.2). This is a form of the well known orthogonal Procrustes problem, which can be solved with the aid of the polar decomposition. (Goldstein and Levy do not observe this similarity and suggest an independent solution). Finally, the solution to (4.8.2) is $U = I$, and the solution to the Löwdin problem is $B = H$, where H is the Hermitian polar factor of any orthogonalization matrix. Orthogonalization matrices can be obtained, for example, by the Gram–Schmidt orthogonalization procedure.

The unitarily constrained total least squares problem arises frequently in many applications in signal processing [3]. The problem in its general form can be described as follows: Given two matrices of noisy measurements $A, B \in \mathbf{C}^{m\times n}$, $(m > n)$, estimate a unitary matrix $X \in \mathbf{C}^{n\times n}$ such that $AX = B$. The problem has a unique analytical solution. This solution is in fact the same as the solution to the orthogonal

Procrustes problem and therefore it can be obtained by using the polar decomposition. Arun [3] studies three signal processing applications where the above mentioned problem emerges. The first is the problem of estimating the translation and rotation of a rigid body between two time instants which arises in many computer vision applications (see for example [14, 27]). The second is the problem of retrieving multiple sinusoids with closely spaced frequencies from estimated convariances. The problem arises in a wide range of signal processing applications [72]. The covariance sequence is usually estimated from time series data. However, there are some sensor array applications and certain applications in astronomical star bearing estimation and interference spectroscopy where the covariance information is directly available. Finally, the third problem concentrates on finding the directions of arrival of multiple radiating sources using a sensor array.

Schreiber and Parlett [110] use the polar decomposition for the computation of block reflectors. Block reflectors are orthogonal, symmetric matrices with possibly more than one negative eigenvalue. They are a generalization of the Householder transformations that are extensively used in matrix computations. Moreover, they have similar uses with Householder transformations and therefore they can be used to compute various basic factorizations, as for example the QR factorization. Block reflectors can also be used in computing optimal error bounds for the approximate eigenvalues of a symmetric matrix, and in reducing a square matrix to block upper Hessenberg form by explicit orthogonal similarity transformations. In [110] Schreiber and Parlett present three algorithms for computing block reflectors. All these algorithms require the employment of the polar decomposition, and the role played by the polar decomposition of a matrix in these methods is revealed.

As we mentioned in Section 2 (Lemma 4.2.2), if $A \in \mathbf{C}^{n\times n}$ is Hermitian with polar decomposition $A = UH$, then the matrix $\frac{1}{2}(A + H)$ is a best Hermitian positive semidefinite approximation to A in the 2-norm. In [61] Higham suggests a possible utilization of this property in optimization theory. Newton's method for the minimization of $F(x)$, $F : \mathbf{R}^n \to R$, requires the computation of the search direction vector p_k at each stage. This vector is a solution of the system $G_k p_k = -g_k$, where $g_k = \nabla F(x_k)$ is the gradient

vector and

$$G_k = \left(\frac{\partial^2 F}{\partial x_i \partial x_j} (x_k) \right)$$

is the (symmetric) Hessian matrix. Higham suggests the replacement of G_k with its Hermitian polar factor H in order to avoid troubles that occur when G_k is not positive definite. In that case, p_k, if defined, might not be a descent direction. The new equation $H_k p_k = -g_k$ may be solved by use of the Cholesky factorization.

The problem of comparing two sets of coordinates X and Y whose rows refer to the same samples of populations, is encountered very frequently in multidimensional scaling in statistics [48]. These sets of coordinates X and Y may arise from different ordinations of the same data, or from the same ordination method used on two sets of data pertaining to the same objects. If $X, Y \in \mathbf{R}^{m \times n}$, $(m \geq n)$, the problem is equivalent to the orthogonal Procrustes problem

$$\min_{U^* U = I_n} \|Y - XU\|_F,$$

and hence its solution can be obtained using the polar decomposition of the matrix $X^* Y$.

Finally, in factor analysis the orthogonal Procrustes problem is encountered in the theory of unique rotational resolution and in conducting a factor analytic research [22]. Both cases are of particular interest to researchers in behavioural and life sciences. In the former case a factor matrix V_0 is first extracted from a correlation matrix R. Then the aim of the researcher is to find the rotation L producing the unique matrix in which the factors correspond to the matrix V_h. V_h consists of determiners in scientific data with properties of general interest. This can be stated as the classical orthogonal Procrustes problem

$$\min_{L^* L = I} \|V_h - V_0 L\|_F.$$

The latter case is associated with a special problem which arises when one needs to resort to *dovetailing* two or more factors analyses together. A detailed discussion on the above mentioned factor analysis problem is given [22].

4.9 Conclusions

Iteration (4.5.3) for computing the unitary polar factor of $A \in \mathbf{C}^{m\times n}$,

$$X_{k+1} = \frac{1}{p} X_k \sum_{i=1}^{p} \frac{1}{\xi_i} (X_k^* X_k + \alpha_i^2 I)^{-1}, \quad X_0 = A, \tag{4.9.1}$$

has inherent high-level parallelism that enables it to be implemented efficiently on the KSR1, with run times an order of magnitude smaller than are obtained using our implementation of the LAPACK SVD routine. With scaling, this iteration is two to four times as fast as the scaled Newton iteration (4.3.4) (which is applicable only to square matrices). We would expect the iteration to be successful on other shared memory parallel computers with a relatively small number of processors (indeed, this has already been demonstrated for the Cray Y-MP with 4 processors by Pandey, Kenney and Laub [96], who used the matrix sign function analogue of iteration (4.9.1)). As is often the case in numerical linear algebra, there is a tradeoff between speed and stability [31]: faster convergence is obtained when scaling is used, but this usually makes the backward error proportional to $\kappa_2(A)$. In some applications this is not an issue because A is known a priori to be well-conditioned or nearly unitary. In general, if high accuracy is important it is better to use the unscaled iteration, which is at worst about twice as slow as the scaled version for $p = 8$ or 16, but which is still at least as fast as the Newton iteration and which has excellent stability.

Chapter 5

A New Approach to Computing the SVD

5.1 Introduction

In the previous chapter we presented a parallel algorithm for computing the polar decomposition. If $A = UH$ is the polar decomposition of $A \in \mathbf{C}^{m\times n}$ $(m \geq n)$, and

$$H = V\Sigma V^* \qquad (5.1.1)$$

is the Schur decomposition of the Hermitian positive semidefinite factor H (with the diagonal entries of Σ in descending order), then the singular value decomposition (SVD) of A is

$$A = (UV)\Sigma V^*.$$

This above observation suggests a new approach to computing the SVD. Provided that an efficient parallel algorithm for the polar decomposition is available, we can obtain the SVD by computing the decomposition (5.1.1) in parallel. It can be shown that if the polar and Schur decompositions are computed in a stable way, the computed SVD is stable too. The numerical stability of this approach has been investigated in [65]. On a new parallel machine, as for example the KSR1, we start off with little software and we want to build library routines. In this chapter, our objective is to develop an efficient

parallel symmetric eigensolver for the KSR1, so that our parallel algorithm for the polar decomposition can be used to obtain an effective parallel SVD solver.

The SVD is one of the most important decompositions in numerical linear algebra with many scientific and engineering applications. These include chemical titration experiments [112], manipulability and sensitivity of industrial robots [117], digital image processing [2, 23], neural networks [18, 86] and matrix nearness problems [107]. The SVD is also a particularly revealing complete orthogonal decomposition and it can be used for the solution of rank-deficient least squares problems. Such problems arise frequently in regression analysis in statistics [34, 52, 55, 87, 100, 102]. Some and these problems are of particular interest, since they require the SVD of very large data matrices. Such problems are encountered in econometrics [33, 105].

In this chapter we confine ourselves to real matrices. Problem (5.1.1) is a symmetric eigenvalue problem (SEP), a problem that has been discussed extensively in the literature. A complete presentation can be found in [98]. We focus our attention on Jacobi methods for solving the symmetric eigenvalue problem. These methods attract current attention because they are inherently parallel. The Jacobi method was first proposed in 1846 [71]. Although it was a well-known technique for computing eigensolutions, it did not receive much attention until the advent of automatic machines. In 1946, one hundred years later, the method was rediscovered and described in [7]. In the following twenty years, the classical Jacobi method and its variants were studied by many researchers, including Henrici [40, 59], Forsythe [40], and Wilkinson [122, 123]. The bulk of this research work was focused on the convergence properties of the methods. Prior to the QR algorithm (1961), the Jacobi technique was the standard method for solving dense symmetric eigenvalue problems. The QR algorithm, although more complex, eclipsed Jacobi methods because it is more economical, and until 1992 it was believed to offer the same reliability on serial machines. In 1992, Demmel and Veselić [30] announced that the Jacobi technique is more accurate than the QR algorithm when a symmetric positive definite matrix happens to well-determine its small eigenvalues. (A matrix well-determines its eigenvalues if small relative pertubations to the entries lead to small relative

pertubations in the eigenvalues.) The emergence of parallel computers brought the Jacobi methods back into the limelight, since certain features of these methods make them particularly well-suited to a multiprocessor system. The first parallel Jacobi method appeared in 1971 [104]. During the 1980s, when more users had access to parallel systems, many researchers focused their attention on parallel Jacobi methods. Variations of these methods have been investigated for use on parallel architectures by various authors, including Eberlein [37] for hypercubes, Brent and Luk [16] for multiprocessor arrays, Schreiber [109] for systolic arrays, and Modi and Pryce [91] for the DAP. The Jacobi algorithm seems to be falling out of favour, once again, in 1990's probably because of its very fine grained model of computing. However, block Jacobi methods do not suffer from this drawback, and this observation motivated us to develop parallel block Jacobi methods especially for the KSR1. We are not aware of any previous work related to Jacobi methods on shared memory computers and especially on the KSR1.

We briefly mention two other well-known ways to compute the SVD of $A \in \mathbf{C}^{m\times n}$ given a Hermitian eigensolver. One involves computing the eigensystem of A^*A; we reject this method on the grounds that it is numerically unstable unless A is well-conditioned [47, Sec. 8.3.2] The second approach is to compute the eigensystem of the $(m+n)\times(m+n)$ matrix $\left[\begin{smallmatrix} 0 & A \\ A^* & 0 \end{smallmatrix}\right]$, from which A's SVD can be "read off" [47, p. 427]. Because of the expanded dimension this approach is substantially slower for large dimensions on the KSR1 than the method we propose (this can be seen from the results in Section 11), and the storage requirements may in any case be prohibitive.

This chapter is structured as follows. Section 2 introduces the classical Jacobi method. Section 3 discusses cyclic Jacobi methods and Section 4 surveys and examines threshold strategies for these methods. Section 5 introduces parallel Jacobi algorithms, and Section 6 and Section 7 discuss and examine two schemes designed to improve the performace of a parallel Jacobi algorithm. In Section 8 we present two block Jacobi algorithms, and in Section 9 we discuss the implementation of scalar and block parallel Jacobi algorithms on the KSR1. In Section 10 we report numerical and timing results for the SEP, and in Section 11 numerical and timing results for the SVD. Section 12 is devoted to applications

of the SVD. Finally in Section 13 we state our conclusions.

5.2 The Classical Jacobi Method

In the classical method of Jacobi [71][1], a real n-by-n symmetric matrix A is converted into a diagonal matrix by a sequence of similarity transformations

$$A_{k+1} = J_k^T A_k J_k, \quad k = 0, 1, \ldots, \tag{5.2.1}$$

where $A_0 = A$, and each J_k is a rotation in the (p, q) plane through an angle θ. These rotations have the form

$$J(p, q, \theta) = \begin{array}{c} \\ \\ p \\ \\ q \\ \\ \\ \end{array} \begin{pmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & c & \cdots & s & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & -s & \cdots & c & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{pmatrix} \tag{5.2.2}$$

(columns p and q marked above the matrix)

and they eliminate the off-diagonal entry $a_{pq}^{(k)}$ (and hence $a_{qp}^{(k)}$), affecting only entries in rows and columns p and q. Throughout this chapter c and s represent the scalar quantities $\cos\theta$ and $\sin\theta$ respectively. These plane rotations are called *Jacobi rotations* and they are no different from Givens rotations. The purpose of the classical Jacobi method is to produce a sequence of matrices A_k and J_k such that

$$\lim_{k\to\infty} A_k = D \quad \text{and} \quad \lim_{k\to\infty} J_1 J_2 \ldots J_k = V,$$

where D is a diagonal matrix representing the eigenvalues of A, and V is an orthogonal matrix with columns the eigenvectors of A.

The basic step in the classical Jacobi algorithm, which will be called the (p, q) subproblem, involves three procedures:

[1] In the original paper [71], plane rotations were used to diagonalize a real 7-by-7 matrix.

1. The determination of an index pair (p, q) that satisfies $1 \le p < q \le n$.

2. The computation of a cosine-sine pair (c, s) such that

$$\begin{bmatrix} a_{pp}^{(k+1)} & a_{pq}^{(k+1)} \\ a_{qp}^{(k+1)} & a_{qq}^{(k+1)} \end{bmatrix} = \begin{bmatrix} c & s \\ -s & c \end{bmatrix}^T \begin{bmatrix} a_{pp}^{(k)} & a_{pq}^{(k)} \\ a_{qp}^{(k)} & a_{qq}^{(k)} \end{bmatrix} \begin{bmatrix} c & s \\ -s & c \end{bmatrix} \tag{5.2.3}$$

is diagonal.

3. The overwriting of A_k with $A_{k+1} = J_k^T A_k J_k$ where J_k is the rotation matrix $J(p, q, \theta)$ in (5.2.2).

Since the Frobenius norm is preserved by orthogonal transformations, (5.2.3) implies that

$$a_{pp}^{(k)^2} + a_{qq}^{(k)^2} + 2a_{pq}^{(k)^2} = a_{pp}^{(k+1)^2} + a_{qq}^{(k+1)^2} + 2a_{pq}^{(k+1)^2} = a_{pp}^{(k+1)^2} + a_{qq}^{(k+1)^2}.$$

Furthermore, if we denote

$$\operatorname{off}(A) = \sqrt{\sum_{i=1}^{n} \sum_{\substack{j=1 \\ j \ne i}}^{n} a_{ij}^2},$$

the "norm" of the off-diagonal entries of a given n-by-n matrix A, we observe that

$$\begin{aligned} \operatorname{off}(A_{k+1})^2 &= \|A_{k+1}\|_F^2 - \sum_{i=1}^{n} a_{ii}^{(k+1)^2} \\ &= \|A_k\|_F^2 - \sum_{i=1}^{n} a_{ii}^{(k)^2} + (a_{pp}^{(k)^2} + a_{qq}^{(k)^2} - a_{pp}^{(k+1)^2} - a_{qq}^{(k+1)^2}) \\ &= \operatorname{off}(A_k)^2 - 2a_{pq}^{(k)^2}, \end{aligned} \tag{5.2.4}$$

and in this way sequence (5.2.1) converges to a diagonal form with each Jacobi step. The number of Jacobi rotations necessary to produce a diagonal form is theoretically infinite since we cannot, in general, solve a polynomial equation in a finite number of steps [123]. However, we may terminate the iterations when the off-diagonal entries are negligible to working accuracy.

In the classical Jacobi method, the off-diagonal entry of A_k of maximum modulus is annihilated at each step. If this entry is in the (p, q) position, then J_k corresponds to a rotation in the (p, q) plane and the angle θ is chosen so as to reduce $a_{pq}^{(k)}$ to zero. *This entry will be made, in general, nonzero by subsequent rotations.* Having determined

the index pair (p, q), the classical Jacobi method proceeds to the computation of the cosine-sine pair (c, s) that diagonalize the 2-by-2 submatrix

$$\begin{bmatrix} a_{pp}^{(k)} & a_{pq}^{(k)} \\ a_{qp}^{(k)} & a_{qq}^{(k)} \end{bmatrix}$$

in (5.2.3). This 2-by-2 symmetric Schur decompostion can be accomplished by computing the cosine-sine pair (c, s) that solves the equation

$$0 = a_{pq}^{(k+1)} = a_{pq}^{(k)}(c^2 - s^2) + (a_{pp}^{(k)} - a_{qq}^{(k)})cs. \tag{5.2.5}$$

If $a_{pq}^{(k)} = 0$, the cosine-sine pair $(c, s) = (1, 0)$ clearly satisfies (5.2.5). If $a_{pq}^{(k)} \neq 0$ and $a_{pp}^{(k)} \neq a_{qq}^{(k)}$, then (5.2.5) may be written as

$$\frac{2a_{pq}^{(k)}}{a_{qq}^{(k)} - a_{pp}^{(k)}} = \frac{2cs}{c^2 - s^2} = \tan 2\theta,$$

since

$$\tan 2\theta = \frac{2\cos\theta\sin\theta}{\cos^2\theta - \sin^2\theta}, \quad \theta \neq k\pi \pm \frac{\pi}{4}, \quad k = 0, 1, \ldots,$$

and the rotation angle θ is

$$\theta = \frac{1}{2}\arctan\frac{2a_{pq}^{(k)}}{a_{qq}^{(k)} - a_{pp}^{(k)}}.$$

In practice there is no need for explicit evaluation of the rotation angle θ since we are interested only in computing the cosine-sine pair (c, s). If we define

$$\tau = \frac{a_{qq}^{(k)} - a_{pp}^{(k)}}{2a_{pq}^{(k)}} \quad \text{and} \quad x = s/c,$$

then (5.2.5) can be written in quadratic form as

$$x^2 + 2\tau x - 1 = 0. \tag{5.2.6}$$

The choice of the smaller of the two roots,

$$x = \frac{\operatorname{sign}(\tau)}{|\tau| + \sqrt{1 + \tau^2}},$$

ensures that $-\pi/4 < \theta \leq \pi/4$. As we will see in the convergence analysis of the method, this choice plays a significant role in the convergence of the method. Having computed the quantity x, the cosine-sine pair (c, s) is resolved from the formulae

$$c = 1/\sqrt{1 + x^2} \quad \text{and} \quad s = xc.$$

The final stage of the (p, q) subproblem is the overwriting of A_k with $A_{k+1} = J_k^T A_k J_k$. As we mentioned earlier in this section, only the rows and columns p and q of A_k are altered during the procedure. If we exploit symmetry, the update can be implemented in $6n$ flops. The classical Jacobi method is summarized in the following algorithm. (We assume that the reader is familiar with the MATLAB notation which is used extensively in the algorithms given in this chapter.)

Algorithm Classical Jacobi.
Given a symmetric $A \in \mathbf{R}^{n\times n}$ and a convergence tolerance $tol > 0$, this algorithm overwrites A with V^TAV. The algorithm terminates when $\text{off}(V^TAV) \leq tol$. On successful exit, the matrix V^TAV contains the eigenvalues of the original matrix A, and the orthogonal matrix V is the matrix of the eigenvectors of A.

$V := I_n$;

$eps = tol\|A\|_F$;

while $\text{off}(A) \geq eps$

Find the index pair (p, q), $1 \leq p < q \leq n$ such that $|a_{pq}| = \max_{i\neq j} |a_{ij}|$;

Compute the cosine-sine pair (c, s) for the index pair (p, q)

and form the matrix $R = \begin{bmatrix} c & s \\ -s & c \end{bmatrix}$;

Update the p and q rows of A: $A([p\ q], :) = R^T A([p\ q], :)$;

Update the p and q columns of A: $A(:, [p\ q]) = A(:, [p\ q])R$;

Update the p and q columns of V: $V(:, [p\ q]) = V(:, [p\ q])R$;

end

A single pass throught the body of the **while** loop in Algorithm Classical Jacobi is called a *sweep*. In the above algorithm each sweep requires $N = \frac{1}{2}n(n-1)$ Jacobi updates. If A is a symmetric n-by-n matrix and a_{pq} is the off-diagonal entry of maximum modulus, then

$$\sum_{i=1}^{n} \sum_{\substack{j=1 \\ j\neq i}}^{n} a_{ij}^2 \leq (n^2 - n)a_{pq}^2, \tag{5.2.7}$$

and (5.2.7) can be written in terms of sweeps as

$$\text{off}(A)^2 \leq 2Na_{pq}^2. \tag{5.2.8}$$

From (5.2.4) we have that

$$2{a_{pq}^{(k)}}^2 = \text{off}(A_k)^2 - \text{off}(A_{k+1})^2,$$

and since from (5.2.8)

$$\text{off}(A_k)^2 \leq 2N{a_{pq}^{(k)}}^2,$$

we conclude that

$$\text{off}(A_{k+1})^2 \leq \left(1 - \frac{1}{N}\right)\text{off}(A_k)^2.$$

Having denoted by A_0 the original matrix A, it follows by induction that

$$\text{off}(A_k)^2 \leq \left(1 - \frac{1}{N}\right)^k \text{off}(A_0)^2. \tag{5.2.9}$$

According to (5.2.9), the classical Jacobi method converges at a linear rate. It is also interesting to observe that if $k = rN$, then

$$\text{off}(A_{rN})^2 \leq \left(1 - \frac{1}{N}\right)^{rN} \text{off}(A_0)^2 < e^{-r}\text{off}(A_0)^2.$$

The above inequality shows that for $r > 2\ln(1/\epsilon)$,

$$\text{off}(A_{rN})^2 < \epsilon^2 \text{off}(A_0)^2. \tag{5.2.10}$$

However, (5.2.10) gives a considerable underestimate of the rate of convergence. The covergence of the classical Jacobi method becomes more rapid in the later stages. In [108], Schönhage shows that for k large enough, there is a constant μ such that

$$\text{off}(A_{k+N}) \leq \mu\ \text{off}(A_k)^2.$$

This behaviour is called *asymptotic quadratic convergence* [98].

The choice of θ plays an important role in the convergence theory for the classical Jacobi process. There is a rigorous analysis in [123], where Wilkinson shows that the condition $-\pi/4 < \theta \leq \pi/4$, ensures convergence of the classical method to a fixed diagonal

matrix. This choice also prevents the nearly converged diagonal entries from interchanging, keeping them within their Gershgorin discs. A rigorous theory for the prediction of the required number of sweeps for a given tolerance has not been developed. However, in [16], Brent and Luk argue heuristically that the number of sweeps is proportional to $\log(n)$. This view is in accordance with our numerical results and seems generally to be the case in practice.

5.3 Cyclic Schemes

In the classical Jacobi method the searching for the largest off-diagonal element is time-consuming. Since every sweep requires $\frac{1}{2}n(n-1)$ searches, the classical method is particularly slow when n is large. This drawback of the original method motivated the development of cyclic Jacobi methods. In a cyclic Jacobi scheme, each sweep consists of $N = \frac{1}{2}n(n-1)$ consecutive elements of the array $((p, q) : 1 \leq p < q \leq n)$, where each pair occurs exactly once. The most common examples are the *row cyclic* and the *column cyclic* Jacobi methods[2] that have been analysed by Forsythe and Henrici in [40]. In the row cyclic Jacobi method we take the N index pairs sequentially in the order

$$(1, 2), (1, 3), \ldots, (1, n); (2, 3), (2, 4), \ldots, (2, n); \ldots; (n-1, n),$$

and then we return to the $(1, 2)$ pair again. The row cyclic Jacobi method is described by the following algorithm.

Algorithm Row Cyclic.
Given a symmetric $A \in \mathbf{R}^{n\times n}$ and a convergence tolerance $tol > 0$, this algorithm iteratively overwrites A with V^TAV. The algorithm terminates when $\mathrm{off}(V^TAV) \leq tol$. On successful exit, the matrix V^TAV contains the eigenvalues of the original matrix A, and the orthogonal matrix V is the matrix of the eigenvectors of A.

$V := I_n$;

[2] In [123], Wilkinson refers to the row cyclic method as the *special* serial Jacobi method.

$eps = tol\|A\|_F$;

while $\text{off}(A) \geq eps$

  for $p = 1\colon n-1$

    for $q = p+1\colon n$

      Compute the cosine-sine pair (c, s) for the index pair (p, q)

      and form the matrix $R = \begin{bmatrix} c & s \\ -s & c \end{bmatrix}$;

      Update the p and q rows of A: $A([p\ q], :) = R^T A([p\ q], :)$;

      Update the p and q columns of A: $A(:, [p\ q]) = A(:, [p\ q])R$;

      Update the p and q columns of V: $V(:, [p\ q]) = V(:, [p\ q])R$;

    end

  end

end

The row cyclic Jacobi method, since it does not require off-diagonal search, is considerably faster than the classical method. Forthsythe and Henrinci [40] have shown that the ultimate rate of convergence of the row cyclic method is quadratic, provided that $-\pi/4 < \theta \leq \pi/4$. However for this choice of θ, it is not known whether every cyclic Jacobi method, other than the row and column cyclic methods, converges to a fixed diagonal matrix. In [57], Hansen discuss the possibility that occasionally cyclic Jacobi methods may not perform well. In [17], Brodlie and Powell report a computer calculation where a cyclic Jacobi method failed to converge due to rounding errors, and they propose a new bound for the angle restriction. The convergence of any cyclic Jacobi method in exact arithmetic can be guaranteed, if at the kth step the index pair (i, j) is chosen so that

$$a_{ij}^{(k)^2} \geq \text{ the average of } \left\{a_{pq}^{(k)^2} : p < q\right\} = \text{off}(A_k)^2/(n(n-1)). \tag{5.3.1}$$

In this case,

$$\text{off}(A_{k+1})^2 \leq \left(1 - \frac{2}{n(n-1)}\right) \text{off}(A_k)^2, \tag{5.3.2}$$

since

$$2a_{ij}^{(k)^2} = \text{off}(A_k)^2 - \text{off}(A_{k+1})^2.$$

Inequality (5.3.2) implies that

$$\lim_{k\to\infty} \mathrm{off}(A_k) = 0,$$

and hence condition (5.3.1) ensures convergence to diagonal form for any cyclic Jacobi method.

5.4 Threshold Jacobi Methods

As we mentioned in Section 2, each Jacobi update can be implemented in $6n$ flops if we exploit symmetry. Since every sweep requires $\frac{1}{2}n(n-1)$ Jacobi updates, the cost of a single sweep is $3n^3+O(n^2)$ flops. This number may be reduced, if we skip the annihilation of entries much smaller than the general level of off-diagonal elements. In this case, since

$$\mathrm{off}(A_{k+1}) = \mathrm{off}(A_k)^2 - 2{a_{pq}^{(k)}}^2,$$

little progress is made in performing the (p,q) rotation at the kth step. This observation led to *threshold Jacobi methods*, variants of the row cyclic Jacobi method. Threshold Jacobi methods annihilate the off-diagonal entries in a row order, skipping rotations when $|a_{pq}^{(k)}|$ is less than some threshold value τ. Provided that threshold strategies are applied with the row cyclic ordering, the convergence of a threshold Jacobi method is guaranteed [40]. Threshold Jacobi methods differ in the way that they determine τ.

Wilkinson [123] suggests the threshold values 2^{-3}, 2^{-6}, 2^{-10} and 2^{-18} for each of the first four sweeps, and the smallest permissible (machine-dependent) number for the subsequent sweeps. The process is terminated when $\frac{1}{2}n(n-1)$ successive entries are skipped. As Wilkinson states in [123], *convergence is guaranteed with this variant since there can only be a finite number of iterations corresponding to any given threshold.* However, the improvement in speed is modest. This is due to the fact that the threshold values are not determined dynamically during the course of the iteration, and hence are independent of the matrix.

Rutishauser [103] suggests for the first three sweeps the threshold value

$$\tau = \frac{1}{5}\sum_{i=2}^{n}\sum_{j=1}^{i-1} |a_{ij}^{(k)}|/n^2, \tag{5.4.1}$$

where A_k is the updated matrix in the beginning of each sweep. For the rest of the sweeps, τ is set to zero.

Modi [92] suggests the calculation of the threshold value

$$\tau = 10^{-s} \sum_{i=1}^{n} \sum_{\substack{j=1 \\ j \neq i}}^{n} {a_{ij}^{(k)}}^2,$$

at the start of each sweep[3]. Modi gives as typical values of s the numbers 2, 4, and 6 for the first three sweeps, and infinity for the rest.

The more interesting technique for determining the threshold value is the *variable* threshold strategy of Kahan and Corneil [26]. Initially the threshold value is set to

$$\tau = \sqrt{\omega / N}, \tag{5.4.2}$$

where

$$\omega = \sum_{i=2}^{n} \sum_{j=1}^{i-1} {a_{ij}^{(k)}}^2,$$

and

$$N = \frac{1}{2} n(n-1).$$

τ is the true root mean square (RMS) of the off-diagonal entries. At each actual rotation ω is reduced by ${a_{ij}^{(k)}}^2$ and τ is recomputed at the cost of 1 multiplication, 1 division, and 1 square root per rotation.

The *variable* threshold strategy updates the threshold value at every actual rotation. This update may cause problems when rotations are performed simultaneously on different processors. Since our purpose is to develop fast parallel Jacobi methods for the KSR1, this observation motivated us to investigate variants of the Kahan-Corneil strategy suitable for parallel environments. The simplest idea is to calculate τ in (5.4.2) only at the start of each sweep and not after update, in order to minimize communication. This calculation is economical since the quantity $\text{off}(A_k) = \sqrt{2\omega}$ is calculated at the start of each sweep for the convergence test. Surprisingly, this modification is competitive with the Kahan-Corneil strategy. This strategy will be called *modified Kahan-Corneil* (modified KC) strategy. As the following numerical examples demonstrate, there are instances where the modified KC strategy performs better than the original one.

[3] To make the threshold test scale independent, we need to take the square root of Modi's value.

		$n = 8$	$n = 16$	$n = 32$
Row Cyclic	Rotations	140	720	2976
	Sweeps	5	6	6
Wilkinson	Rotations	132	650	2746
	Reduction	6%	10%	8%
	Sweeps	5	6	6
Rutishauser	Rotations	131	643	2110
	Reduction	6%	11%	29%
	Sweeps	5	6	5
Modi	Rotations	135	503	2786
	Reduction	4%	30%	6%
	Sweeps	6	6	6
Kahan-Corneil	Rotations	84	337	1456
	Reduction	40%	53%	51%
	Sweeps	9	10	10
Modified KC	Rotations	80	350	1354
	Reduction	43%	51%	55%
	Sweeps	13	15	14

Table 5.4.1: Comparison of five threshold strategies.

We incorporated the Wilkinson, Rutishauser, Modi, Kahan-Corneil and modified KC threshold strategies in algorithm Row Cyclic. We implemented the new algorithms in MATLAB on a SUN SPARC workstation. The iterations were terminated when $\text{off}(A_k) \leq nu\|A\|_F$, where the unit roundoff error $u \approx 1.1 \times 10^{-16}$. We first examined the matrix defined by

$$a_{ij} = \begin{cases} i + j & \text{if } i \neq j, \\ i^2 + n & \text{otherwise,} \end{cases}$$

for $n = 8$, 16, and 32. The results are summarized in Table 5.4.1, which also includes results for the row cyclic Jacobi method for comparison. Table 5.4.1 reports the number of actual Jacobi rotations, the reduction in actual rotations with relation to the row cyclic Jacobi method, and the number of sweeps.

Similar behaviour has been observed in a large number of numerical experiments with random symmetric matrices of various order. According to Table 5.4.1, the fastest threshold strategies are the Kahan-Corneil and the modified KC. These two strategies seem to follow closely the progress of the Jacobi process, determining dynamically the next threshold value in a very efficient way. Moreover, they were found to be as as accurate as the slower ones. The increase in the number of sweeps, especially in the modified KC,

does not mean any extra cost; the real cost of any Jacobi method is determined by the number of actual rotations. The behaviour of the modified KC strategy encouraged us to develop threshold strategies for parallel Jacobi methods based on this pattern.

5.5 Parallel Jacobi Methods

Jacobi methods are ideally suited for parallel implementation, since each rotation affects only two rows and two columns. This feature of Jacobi methods may be exploited in order to eliminate more than one entry at a time. (Because of symmetry only the entries above the main diagonal need be considered). For example, if A is a 4-by-4 symmetric matrix, and J is a matrix of the form

$$J = \begin{bmatrix} c_1 & 0 & s_1 & 0 \\ 0 & c_2 & 0 & s_2 \\ -s_1 & 0 & c_1 & 0 \\ 0 & -s_2 & 0 & c_2 \end{bmatrix}, \quad c_i = \cos\theta_i, \quad s_i = \sin\theta_i, \quad i = 1, 2, \tag{5.5.1}$$

then $J^T AJ$ would have zero entries in positions $(1, 3)$ and $(2, 4)$, provided that the angles θ_1, θ_2, have been chosen properly. The calculation of the cosine-sine pairs (c_i, s_i), $i = 1, 2$, and the corresponding rotations, may be performed simultaneously on different processors. A matrix of the type (5.5.1) will be called a *compound Jacobi rotation*.

The above example can be generalized for symmetric matrices of order $n \geq 4$. We observe that the maximun number of off-diagonal entries that can be annihilated by a compound Jacobi rotation is $\lfloor n/2 \rfloor$, where $\lfloor x \rfloor$ is the greatest integer less than or equal to x. A compound Jacobi rotation that annihilates $\lfloor n/2 \rfloor$ off-diagonal entries will be called a *complete Jacobi rotation*. If we define $m = \lfloor (n + 1)/2 \rfloor$, then $2m - 1$ distinct complete Jacobi rotations are needed in order to annihilate each of the off-diagonal entries exactly once. A sequence of $2m-1$ Jacobi updates that eliminates each of the off-diagonal entries

exactly once will be called a *sweep*[4]. Note that

$$2m-1=\begin{cases} n & \text{if } n \text{ is odd,} \\ n-1 & \text{if } n \text{ is even.} \end{cases}$$

For the rest of this chapter n will be assumed to be even for convenience.

Two Jacobi rotations $J(p_1, q_1, \theta_1)$ and $J(p_2, q_2, \theta_2)$ are said to be *disjoint* if the indices p_1, q_1, p_2, and q_2 are all distinct. A complete Jacobi rotation J can be considered as a product of $n/2$ disjoint Jacobi rotations, that is,

$$J=\prod_{i=1}^{n/2} J(p_i, q_i, \theta_i), \quad \{p_i, q_i\} \cap \{p_j, q_j\} = \emptyset, \quad i \neq j.$$

The first problem that we face in designing a parallel Jacobi method is the construction of the $n-1$ complete Jacobi rotations that together annihilate each off-diagonal entry exactly once during a sweep. This combinatorial problem leads to the notion of *Jacobi sets*: the partition of the $\binom{n}{2}$ 2-subsets of the integers $\{1, \ldots, n\}$ into $n-1$ sets of $n/2$ distinct unordered pairs. Each of the $n-1$ Jacobi sets corresponds to a complete Jacobi rotation. For example, if $n=4$ then there are three Jacobi sets $\{(1,2),(3,4)\}$, $\{(1,3),(2,4)\}$, and $\{(1,4),(2,3)\}$, that correspond respectively to the complete Jacobi rotations

$$\begin{bmatrix} c_1 & s_1 & 0 & 0 \\ -s_1 & c_1 & 0 & 0 \\ 0 & 0 & c_2 & s_2 \\ 0 & 0 & -s_2 & c_2 \end{bmatrix}, \quad \begin{bmatrix} c_1 & 0 & s_1 & 0 \\ 0 & c_2 & 0 & s_2 \\ -s_1 & 0 & c_1 & 0 \\ 0 & -s_2 & 0 & c_2 \end{bmatrix}, \quad \text{and} \quad \begin{bmatrix} c_1 & 0 & 0 & s_1 \\ 0 & c_2 & s_2 & 0 \\ 0 & -s_2 & c_2 & 0 \\ -s_1 & 0 & 0 & c_1 \end{bmatrix}.$$

Several schemes have been developed for the generation of the $n-1$ Jacobi sets. Some of them have been designed especially for the particular architecture of the target machine. For example, for distributed memory machines, it is desirable that each Jacobi set be obtained from the previous Jacobi set by messages from neighbouring processors. Schemes that refer to parallel architectures based on the distributed memory model are

[4]This definition of sweep is not different than the one we gave in Section 2. As we will see later in this section, when we state Algotithm Parallel Jacobi, the annihilation of each of the off-diagonal entries exactly once requires a single pass throught the body of a **while** loop.

not of our interest, since our target machine is the KSR1. The following two schemes are suitable for the KSR1.

The first scheme was presented by Sameh [104] in 1971. This schemes determines the $n/2$ index pairs in the $n-1$ Jacobi sets

$$\text{jac.set}(k) = \{(p_{ki}, q_{ki}), i = 1, \ldots, n/2\}, \quad k = 1, \ldots, n-1,$$

using the following algorithm.

Algorithm Sameh.
Given the order of a symmetric matrix n and an integer $k \leq n-1$ the following algorithm computes the kth Jacobi set,

$$\text{jac.set}(k) = \{(p_{ki}, q_{ki}), i = 1, \ldots, n/2\}, \quad k = 1, \ldots, n-1,$$

according to Sameh's scheme.

$i := 1;$

if $k \leq n/2 - 1$

    for $l = n/2 - k + 1 : n - k$

    $q_{ki} = l;$

        if $n/2 - k + 1 \leq l$ **and** $l \leq n - 2k$

            $p_{ki} = n - 2k + 1 - l;$

        else if $n - 2k < l$ **and** $l \leq n - k - 1$

            $p_{ki} = 2n - 2k - l;$

        else

            $p_{ki} = n;$

        end if

    $i := i + 1;$

    end

else

    for $l = n - k : 3n/2 - k - 1$

    $q_{ki} = l;$

        if $l < n - k + 1$

$p_{ki} = n;$

else if $n - k + 1 \leq l$ **and** $l \leq 2n - 2k - 1$

$p_{ki} = 2n - 2k - l;$

else

$p_{ki} = 3n - 2k - 1 - l;$

end if

$i = i + 1;$

end

end

It is noteworthy that when $k = n - 1$, the Jacobi set given by Sameh's scheme corresponds to a complete Jacobi rotation of a special structure. For example, if $n = 8$ and $k = 7$, the 7th Jacobi set is

$$\text{jac.set}(7) = \{(8,1),(7,2),(6,3),(5,4)\},$$

and the corresponding complete Jacobi rotation J_7 has the form

$$J_7 = \begin{bmatrix} c_1 & & & & & & & s_1 \\ & c_2 & & & & & s_2 & \\ & & c_3 & & & s_3 & & \\ & & & c_4 & s_4 & & & \\ & & & -s_4 & c_4 & & & \\ & & -s_3 & & & c_3 & & \\ & -s_2 & & & & & c_2 & \\ -s_1 & & & & & & & c_1 \end{bmatrix}. \tag{5.5.2}$$

Complete Jacobi rotations of the form (5.5.2) play an important role in a reduced cyclic Jacobi scheme that we discuss in the following section.

Perhaps the simplest way to generate the $n - 1$ Jacobi sets is to visualize a chess tournament with n players in which everybody must play everybody else exactly once. The following example is taken from [47]. Suppose that we have $n = 8$ players and in round one we have the following four games

Figure 5.5.1: A merry-go-round scheme for players 2 through 8.

1	3	5	7
2	4	6	8

These games correspond to the Jacobi set

$$\text{jac.set}(1) = \{(1,2),(3,4),(5,6),(7,8)\}.$$

To set up rounds 2 to 7, player 1 stays put and players 2 through 8 embark on a merry-go-round, as illustrated in Figure 5.5.1. These operations can be encoded in a pair (top, bot) of integer vectors $top(1\colon n/2)$ and $bot(1\colon n/2)$. During round k, $top(i)$ plays $bot(i)$, $i = 1, \ldots, n/2$, and the pairs $\{(top(i), bot(i)), \quad i = 1, \ldots, n/2\}$ constitute the kth Jacobi set. The following algorithm describes how the next pair (top, bot) can be obtained from the current pair (top, bot).

Algorithm Tournament.
Given a pair of integer vectors (top, bot) that correspond to a Jacobi set, this algorithm updates (top, bot) in order to determine the the next Jacobi set.

$m = \text{length}(top)$

for $k = 1\colon m$

    if $k = 2$

        $new.top(k) = bot(1)$

    else if $k > 2$

        $new.top(k) = top(k-1)$

    end if

    if $k = m$

        $new.bot(k) = top(k)$

else if $k > 2$

$new.bot(k) = bot(k+1)$

end if

end

$new.top(1) = 1$

$top = new.top$

$bot = new.bot$

As we mentioned earlier in this section, each sweep requires $n - 1$ steps in order to annihilate each off-diagonal entry exactly once. These steps will be called *Jacobi steps*. Each Jacobi step is associated with a Jacobi set. This Jacobi set may be determined either by Algorithm Sameh or Algorithm Tournament. The first task of each Jacobi step is the allocation of the $n/2$ distinct index pairs, the elements of the associated Jacobi set, to different processors, taking full advantage of the particular architecture of the target machine. If the number of processors P is less than $n/2$, then each Jacobi step should allocate the $n/2$ index pairs to different processors more than once. Hence, systems with many processors may be more appropriate than systems with fewer processors when n is large.

Suppose that the local cache of processor A on the KSR1 accommodates the index pair (p, q), and the entries $a_{pp}^{(k)}$, $a_{qq}^{(k)}$, and $a_{pq}^{(k)}$ of the updated matrix A_k. Having computed the corresponding cosine-sine pair (c, s), processor A has two alternatives:

1. Update the p and q rows, and then the p and q columns of A_k,
2. Embed c and s into the corresponding positions of an n-by-n identity matrix, constructing the Jacobi rotation $J(p, q, \theta)$. If the other processors do the same, after $n/2$ embeddings the identity matrix will be transformed to a complete Jacobi rotation.

In the first alternative, processor A will execute a load instruction for the addresses of the rows and columns p and q. If the corresponding page(s) is (are) not found in processor A's local cache, then ALLCACHE memory will allocate the page(s) that contain(s) these

addresses in processor A's local cache. The addresses and the data of the updated rows and columns p and q will reside in processor A's local cache until another processor references their addresses. If n is large, this procedure will keep the Search Engine very busy since there will be a large number of transfer demands. We also note that when operating on pairs of rows and columns we cannot exploit higher level BLAS. Using up to 16 processors on the KSR1, we found that even for modest n, for example $n = 64$, this procedure is very slow. (In Section 9 we report an example where this alternative requires 210.03 seconds for a 64-by-64 symmetric matrix, while the second alternative discussed in the following paragraph requires 8.13 seconds).

The second alternative was found to be more suitable for the KSR1 when we use up to 16 processors. Having filled up the complete Jacobi rotation J_k using all the processors, the updated matrix $A_{k+1} = J_k^T A_k J_k$ can be formed using parallel matrix multiply. This technique is used in the following algorithm.

Algorithm Parallel Jacobi .
Given a symmetric $A \in \mathbf{R}^{n \times n}$ and a convergence tolerance $tol > 0$, this algorithm iteratively overwrites A with $V^T AV$. The algorithm terminates when $\text{off}(V^T AV) \leq tol$. On successful exit, the matrix $V^T AV$ contains the eigenvalues of the original matrix A, and the orthogonal matrix V is the matrix of the eigenvectors of A. The number of processors is P.

$V := I_n$;

$eps = tol\|A\|_F$;

while $\text{off}(A) \geq eps$

  for $i = 1{:}\, n-1$ ($n-1$ iterations are required for a sweep)

    Generate the ith Jacobi set $\text{jac.set}(i) = \{(p_{il}, q_{il}), \quad l = 1, \ldots, n/2\}$,

    using Sameh's algorithm or the Tournament scheme;

    $J := I_n$;

    for $l = 1{:}\, n/2$

      On processor $k = 1 + ((l-1) \bmod P)$:

$p = \min(p_{il}, q_{il})$

$q = \max(p_{il}, q_{il})$

Compute the cosine-sine pair (c, s), and embed c and s

into the corresponding positions in J;

end (of the parallel section)

end

Form $A = J^T AJ$ using parallel matrix multiply;

Form $V = VJ$ using parallel matrix multiply;

end

end

In Section 9 we present some numerical results concerning the performance of Algorithm Parallel Jacobi. These results clearly indicate that Algorithm Parallel Jacobi is a suitable algorithm for solving symmetric eigenproblems when $P \ll n$. In this case a block version of Algorithm Parallel Jacobi may be appropriate. Parallel block Jacobi algorithms are discussed in Section 8.

5.6 A Reduced Cyclic Jacobi Method

As we mentioned in the previous section, if we use Sameh's scheme to generate the $n-1$ Jacobi sets for each sweep, the $(n-1)$st Jacobi set corresponds to a complete Jacobi rotation of the form (5.5.2). If A_k is the updated matrix after $n-2$ Jacobi steps within a sweep, then the complete Jacobi rotation that corresponds to the $(n-1)$st Jacobi set annihilates the entries on the anti-diagonal of A_k. Moreover, the first $n-2$ Jacobi steps in a sweep never annihilate entries on the anti-diagonal. Omitting the $(n-1)$st Jacobi

step in every sweep, the original matrix A converges to a pattern of the form

$$X = \begin{bmatrix} x_{11} & & & & & & x_{1n} \\ & x_{21} & & & & x_{2,n-1} & \\ & & x_{31} & & x_{3,n-2} & & \\ & & & \ddots \quad \cdot^{\cdot^{\cdot}} & & & \\ & & & \cdot^{\cdot^{\cdot}} \quad \ddots & & & \\ & & x_{n-2,3} & & x_{n-2,n-2} & & \\ & x_{n-1,2} & & & & x_{n-1,n-1} & \\ x_{n1} & & & & & & x_{nn} \end{bmatrix}. \tag{5.6.1}$$

Since the matrix X has been produced after a sequence of orthogonal similarity transformations, the matrix X has the same set of eigenvalues as the original matrix A. The elimination procedure may terminate when

$$\text{off}_x(A_k) := \sqrt{\sum_{i=1}^{n} \sum_{\substack{j=1 \\ j\neq i,n+1-i}}^{n} a_{ij}^{(k)^2}} \leq eps,$$

with eps being chosen according to the requirement for accuracy. Note that we added the subscript x in order not to confuse $\text{off}_x(A_k)$ with $\text{off}(A_k)$.

The special structure of the matrix X in (5.6.1) can be exploited in order to obtain the eigenvalues of A. Indeed, the characteristic equation $\det(X - \lambda I) = 0$ can be factorized to give

$$(x_{ii} - \lambda)(x_{n+1-i,n+1-i} - \lambda) - x_{i,n+1-i}^2 = 0, \quad i = 1, \ldots, n/2. \tag{5.6.2}$$

The eigenvalues of the original matrix A can be obtained by solving the quadratic equations (5.6.2), which must have real roots since the original matrix A is symmetric.

The quadratic equations (5.6.2) can be written as

$$\lambda^2 - (x_{ii} + x_{n+1-i,n+1-i})\lambda + x_{ii}x_{n+1-i,n+1-i} - x_{i,n+1-i}^2 = 0, \quad i = 1, \ldots, n/2. \tag{5.6.3}$$

If we set

$$b_i := x_{ii} + x_{n+1-i,n+1-i}, \quad \text{and} \quad c_i := x_{ii}x_{n+1-i,n+1-i} - x_{i,n+1-i}^2, \quad i = 1, \ldots, n/2, \tag{5.6.4}$$

then the eigenvalues of A may be obtained from the formulae

$$\lambda_{i1} = \frac{1}{2}\left(b_i + \sqrt{b_i^2 - 4c_i}\right), \quad \lambda_{i2} = \frac{c_i}{\lambda_{i1}}, \quad i = 1, \ldots, n/2. \tag{5.6.5}$$

We also observe that the $n/2$ quadratic equations in (5.6.3) may be solved simultaneously and hence the scheme is suitable for parallel computation. The above scheme has been suggested by Modi in [92], where it is called a *reduced cyclic Jacobi method*. We will refer to this method as the *X method*[5].

The algorithm given in [92] does not compute the matrix V of the eigenvectors of the original matrix A. In order to compute this matrix we have to update the matrix V during the elimination procedure, as we do in Algorithm Parallel Jacobi. The iterative procedure converges to a matrix X of the form (5.6.1), and if A is the original matrix and V is the updated matrix of the eigenvectors at the end of the iterative procedure, then

$$X = V^T AV.$$

As we mentioned earlier in this section, the complete Jacobi rotation that corresponds to $(n-1)$st Jacobi set annihilates the entries on the anti-diagonal of the updated matrix. Thus, we can construct a complete Jacobi rotation R such that

$$D = R^T XR,$$

where D is a diagonal matrix which contains the eigenvalues of A. The matrix V whose columns are the eigenvectors of A will be $V := VR$. The following algorithm describes how the eigenvalues, and optionally the matrix of the eigenvectors of a symmetric matrix, can be obtained using this method.

Algorithm X

[5] In [92], Modi sets in (5.6.5) $\lambda_{i2} = b_i - \lambda_{i1}$ instead of $\lambda_{i2} = c_i/\lambda_{i1}$. Our choice is numerically better since there is less chance of cancellation errors.

Given a symmetric $A \in \mathbf{R}^{n \times n}$ and a tolerance $tol > 0$, this algorithm first reduces the original matrix to a matrix of the form (5.6.1), and then computes the n eigenvalues of A. On succesful exit, if the computation of the matrix V of the eigenvectors is required, the matrix $V^T AV$ contains the eigenvalues of the original matrix A. Otherwise, the algorithm computes a vector which contains the eigenvalues of A. The number of processors is P.

if the computation of V is required, $V := I_n$;

$eps = tol \|A\|_F$;

while $\sqrt{\sum_{i=1}^{n} \sum_{\substack{j=1 \\ j \neq i, n+1-i}}^{n} a_{ij}^2} \geq eps$

  for $i = 1 : n-2$ (we omit the $(n-1)$st Jacobi step)

    Generate the ith Jacobi set jac.set$(i) = \{(p_{il}, q_{il}), \quad l = 1, \ldots, n/2\}$, using Sameh's scheme;

    $J := I_n$;

    for $l = 1 : n/2$

      On processor $k = 1 + ((l-1) \bmod P)$:

        $p = \min(p_{il}, q_{il})$;

        $q = \max(p_{il}, q_{il})$;

        Compute the cosine-sine pair (c, s) that corresponds to the index pair (p, q), and embed c and s into the corresponding positions of J;

      end (of the parallel section)

    end

    Form $A := J^T AJ$ using parallel matrix multiply;

    if the computation of V is required,

      form $V := VJ$ using parallel matrix multiply;

    end if

  end

end (of **while** loop)

if the computation of V is required

$J{:}= I_n$;

for $l = 1{:}\, n/2$

On processor $k = 1 + ((l-1) \bmod P)$:

$p = l$;

$q = n + 1 - l$;

Compute the cosine-sine pair (c, s) that corresponds

to the index pair (p, q), and embed c and s into

the corresponding positions of J;

end (of the parallel section)

end

Form $A{:}= J^T AJ$ using parallel matrix multiply;

Form $V{:}= VJ$ using parallel matrix multiply;

else

Form the vectors b and c as in (5.6.4) using P processors

(This can be accomplished on the KSR1 with tiling);

for $i = 1{:}\, n/2$

On processor $k = 1 + ((i-1) \bmod P)$:

$\lambda_{i1} = \frac{1}{2}\left(b_i + \sqrt{b_i^2 - 4c_i}\right)$;

$\lambda_{i2} = c_i/\lambda_{i1}$;

end (of the parallel section)

end

end

If the computation of the eigenvectors is not required, the reduction of one step per sweep necessitates the solving of $n/2$ quadratic equations. The extra cost of solving the $n/2$ quadratic equations on the KSR1 is negligible in comparison with the cost of a Jacobi step. For example, for a symmetric matrix of order 64, a Jacobi step needs

Parallel Jacobi		X Method			
sweep	$O(\text{off}(A))$	sweep	$O(\text{off}_x(A))$	sweep	$O(\text{off}_x(A))$
0	10^5	0	10^5	8	10^{-4}
1	10^5	1	10^5	9	10^{-5}
2	10^4	2	10^4	10	10^{-6}
3	10^3	3	10^3	11	10^{-7}
4	10^2	4	10^2	12	10^{-8}
5	10^{-4}	5	10^0	13	10^{-9}
6	10^{-10}	6	10^{-2}	14	10^{-10}
7	-	7	10^{-3}		

Table 5.6.1: Convergence for Algorithms Parallel Jacobi and X.

0.05 seconds, while the simultaneous solution of the quadratic equations requires less than 0.01 seconds. Provided that Algorithm X and Algorithm Parallel Jacobi require the same number of sweeps, the former algorithm is faster when we also compute the eigenvectors. This happens because algorithm X skips a Jacobi step per sweep and applies only one Jacobi step (the $(n-1)$st Jacobi step), after the iterative procedure. However, we have reason to doubt whether Algorithm X is faster than Algorithm Parallel Jacobi. During our numerical experiments we observed that Algorithm X needs more sweeps than algorithm Parallel Jacobi for the same accuracy requirements. (The Jacobi sets in Algorithm Parallel Jacobi have been generated using Sameh's scheme). A typical example is the following: If A is the 64-by-64 symmetric matrix defined by

$$a_{ij} = \begin{cases} i+j & \text{if } i \neq j, \\ i^2 + 64 & \text{otherwise,} \end{cases}$$

and $tol = 1.1 \times 10^{-16}$, then Algorithm X needs 14 sweeps, while algorithm Parallel Jacobi needs only 6. The progress of convergence of the two algorithms is illustrated in Table 5.6.1, where we observe that the ultimate convergence rate for algorithm X is linear. The same behaviour of the two algorithms has been noticed in a number of numerical experiments. Our conclusion is that *the X method, although attractive in theory, it is not of practical importance since it requires more sweeps.* These extra sweeps cause unnecessary delay in the convergence of a Parallel Jacobi method. The convergence properties of the X method are not discussed in [92].

5.7 The Davies-Modi Method

In [28], Davies and Modi suggest the replacement of the final sweep(s) of a Jacobi method by a direct scheme. This scheme is applicable only when the eigenvalues are known to be distinct. This is not a serious drawback of the method since many symmetric matrices arising in applications have distinct eigenvalues. (In Section 10 we investigate the performance of the Davies-Modi scheme on matrices with multiple and close eigenvalues).

We state the Davies-Modi method for completing symmetric eigenvalue problems without explanation. The underlying theory, which is based on some pertubation theory, is given in [28]. If A is an n-by-n symmetric matrix, we apply a Jacobi method up to the point at which the updated matrix A_k is close-to-diagonal. Then we apply the Davies-Modi method that constructs a nearly orthogonal matrix U such that

$$UU^T \approx I_n, \quad \text{and} \quad UA_kU^T \approx \Delta,$$

where Δ is a nearly diagonal matrix of the approximate eigenvalues.

The Davies-Modi scheme can be described as follows: We first write A_k as

$$A_k = A_0 + A_1, \tag{5.7.1}$$

where A_0 is diagonal and A_1 has zeros on its main diagonal. Solving for the skew-symmetric matrix R the equation

$$A_0R - RA_0 = A_1,$$

we obtain

$$r_{ij} = \begin{cases} \dfrac{a_{ij}^{(k)}}{a_{ii}^{(k)} - a_{jj}^{(k)}} & i \neq j, \\ 0 & i = j. \end{cases} \tag{5.7.2}$$

Then we write

$$\frac{1}{2}(RA_1 - A_1R) = B_0 + B_1,$$

where B_0 is diagonal and B_1 has zeros on its main diagonal. Here we observe that RA_1 is the transpose of $-A_1R$, since R is skew-symmetric, that is $R = -R^T$, and A_1 is symmetric. Thus the matrix B_1 is symmetric. Then we solve the equation

$$A_0W - WA_0 = B_1,$$

for the skew-symmetric matrix W, and we obtain

$$w_{ij} = \begin{cases} \dfrac{b_{ij}}{a_{ii}^{(k)} - a_{jj}^{(k)}} & i \neq j, \\ 0 & i = j. \end{cases} \tag{5.7.3}$$

Defining $X = R + W$, the matrix U is given by

$$U = e^X = \sum_{k=0}^{\infty} \frac{1}{k!} X^k.$$

If sufficient terms are taken, the matrix U will be orthogonal to machine precision. In practice, the method performs well for $k = 3$, that is

$$U = I_n + X + \frac{1}{2}X^2 + \frac{1}{6}X^3. \tag{5.7.4}$$

If $\alpha = \max\left\{|a_{ij}^{(k)}| : i \neq j\right\}$, $\delta = \min\left\{|a_{ii}^{(k)} - a_{jj}^{(k)}| : \quad i \neq j\right\}$, and

$$\epsilon = \max\left\{n^3\alpha^4/\delta^4, n^2\alpha^3/\delta^2, n^2\alpha^4/\delta^3\right\},$$

Davies and Modi suggest the application of their scheme from the point at which $\epsilon < 10^{-k}$, with $k = 3$. Provided that $\epsilon \leq n^3/16(n-1)^4$, the Gershgorin discs, with centers $a_{ii}^{(k)}$ and radii $\sum_{j \neq i} |a_{ij}^{(k)}|$, are disjoint at the stage when the Modi-Davies method is applied [28]. This also ensures that the updated matrix A_k is diagonally dominant.

For the computation of δ Modi [92] suggests a procedure with the following steps:

1. Extract the vector $d = (a_{11}^{(k)}, \ldots, a_{nn}^{(k)})$;
2. Form two matrices, one with all rows equal to d^T and the other with all columns equal to d, and take their difference;
3. Replace the zero diagonal entries by a large constant;
4. Find the minimum of the absolute values of the entries of this matrix.

We investigated the performance of the above scheme on the KSR1. We used tiling to form the vector d, and the two matrices in step 2. We also used tiling to form the $(n-1)$-by-$(n-1)$ triangular matrix of the difference of their sub-diagonal entries, and to find the minimum of the absolute values of these entries. We compared this method with the following procedure:

1. Extract the vector $d = (a_{11}^{(k)}, \ldots, a_{nn}^{(k)})$;
2. Order the elements of the vector d in descending order;
3. Find the minimum difference $a_{ii}^{(k)} - a_{i+1,i+1}^{(k)}$, $i = 1, \ldots, n-1$.

In our implementation on the KSR1, we used tiling for the steps 1 and 3. For the sorting in step 2, we used a modification of Bubble Sort [6]. We tested the performance of the two methods for the vectors d, $d_i = \sin(i)$, $i = 1, \ldots, n$, for various values of n. We chose these vectors because their elements are unordered. We found that the second procedure is faster than Modi's procedure. For example, for $n = 1024$ and using 16 processors, Modi's procedure needs 0.75 seconds to determine δ, while the method based on the sorting procedure requires 0.22 seconds. Figure 5.7.1 illustrates the performance of the two methods, for various values of n between 64 and 1024. It is clear that the second scheme is more suitable for the KSR1. For the determination of α we used tiling. The computation of the quantities α, δ, and ϵ, is not necessary in the first stages of the elimination procedure. Usually, the condition for applying the direct method is satisfied after the fourth sweep. The Davies-Modi method may be combined with algorithm Parallel Jacobi giving the following algorithm.

Algorithm Davies-Modi.
Given a symmetric $A \in \mathbf{R}^{n \times n}$ and a convergence tolerance $tol > 0$, this algorithm overwrites A with $V^T AV$ using the Davies-Modi scheme. On successful exit, the diagonal of $V^T AV$ contains the eigenvalues of the original matrix A, and the columns of V are the eigenvectors of A.

$V := I_n$;

$eps = tol\|A\|_F$;

$MODI := FALSE$;

$SWEEP := 0$;

while $\text{off}(A) \geq eps$ **and** $MODI = FALSE$

[6] We are aware that Bubble Sort is not the fastest sorting method but it can be easily implemented in Fortran. The development of parallel sorting methods for the KSR1 is beyond the scope of this thesis.

Figure 5.7.1: Performance of two methods for determining δ.

SWEEP := SWEEP + 1;

Update A and V as in algorithm Parallel Jacobi;

if $SWEEP \geq 4$

$\alpha = \max\{|a_{ij}|: \quad i \neq j\}$;

$\delta = \min\{|a_{ii} - a_{jj}|: \quad i \neq j\}$;

$\epsilon = \max\{n^3\alpha^4/\delta^4, n^2\alpha^3/\delta^2, n^2\alpha^4/\delta^3\}$;

if $\epsilon < 10^{-3}$, $MODI := TRUE$;

end if

end

Express $A = A_0 + A_1$ as in (5.7.1);

Compute R as in (5.7.2);

Compute $B = \frac{1}{2}(RA_1 - A_1R)$;

Compute W as in (5.7.3);

Compute $X = R + W$;

Compute U as in (5.7.4);

$A = UAU^T$;

$V = U^TV$;

end

We also incorporated the Davies-Modi method in various Jacobi schemes with threshold strategy that we investigated on the KSR1. Our conclusion, based on lots of numerical tests, is that the incorporation of this scheme inside a parallel Jacobi method improves the timing results significantly, preserving the accuracy of results. Numerical and timing results concerning the incorporation of the Davies-Modi method in various Jacobi schemes are given in Section 9 and Section 10.

5.8 Block Jacobi Methods

One of the important requirements in exploiting a high performance computer is to avoid unecessary memory references. In the KSR1, data flows among local caches, and between each processor and its local cache. Performance of algorithms can be dominated by the amount of memory traffic, rather than the number of operations involved. This cost provides considerable motivation to devise block versions of the Jacobi algorithms discussed in the previous sections, in order to minimize data movement. It is well known that block algorithms are advantageous for systems with few processors [12, 32]. In almost every modern computer, computation is much cheaper than input/output. This is particularly so in high performance environments where the data transfer rate is low compared to the arithmetic speed.Given a d-by-d matrix block, the cost of data movement is $O(d^2)$, whereas typically $O(d^3)$ operations are required. The $O(d)$ ratio of work to storage means that processors with an $O(d)$ ratio of computing speed to input/output bandwidth can be tolarated [113]. Furthermore block algorithms usually lead to programs that are rich in matrix-vector and matrix-matrix operations, in which case we can exploit higher level BLAS. Parallel Jacobi methods are of particular interest to us, since we are interested in large symmetric eigenproblems, and we do not have access to more than 16 processors on the KSR1.

Scalar Jacobi algorithms can be viewed as repeatedly solving 2-by-2 eigenvalue problems. Block Jacobi algorithms repeatedly solve larger eigenvalue problems. For example,

if we write the n-by-n symmetric matrix A as

$$A = \begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1k} \\ A_{21} & A_{22} & \cdots & A_{2k} \\ \vdots & \vdots & & \vdots \\ A_{k1} & A_{k2} & \cdots & A_{kk} \end{bmatrix}, \quad A_{ij} \in \mathbf{R}^{r \times r},$$

where $n = rk$, then the (p, q) subproblem may involve the computation of the $2r$-by-$2r$ Schur decomposition

$$\begin{bmatrix} A_{pp} & A_{pq} \\ A_{qp} & A_{qq} \end{bmatrix} = \begin{bmatrix} V_{pp} & V_{pq} \\ V_{qp} & V_{qq} \end{bmatrix} \begin{bmatrix} D_{pp} & 0 \\ 0 & D_{qq} \end{bmatrix} \begin{bmatrix} V_{pp} & V_{pq} \\ V_{qp} & V_{qq} \end{bmatrix}^T. \tag{5.8.1}$$

(In this section, we also discuss block Jacobi methods in which the complete Schur decomposition (5.8.1) may not be necessary). The corresponding block Jacobi rotation is

$$J = \begin{bmatrix} I_{rr} & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & V_{pp} & \cdots & V_{pq} & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & V_{qp} & \cdots & V_{qq} & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & I_{rr} \end{bmatrix}. \tag{5.8.2}$$

Note that throughout this section the block size is assumed to be r, and every $n \times n$ matrix is assumed to be a block $k \times k$ matrix where $n = rk$. The following lemma indicates the reduction in $\text{off}(A)$ when we apply the block Jacobi rotation J to A.

Lemma 5.8.1 *If A is an n-by-n symmetric matrix, V is a block Jacobi rotation of the form (5.8.2), and (5.8.1) holds, then*

$$\text{off}(J^T A J)^2 = \text{off}(A)^2 - \left(2\|A_{pq}\|^2 + \text{off}(A_{pp})^2 + \text{off}(A_{qq})^2\right).$$

Proof. If we set $B = J^T A J$, then

$$\begin{bmatrix} B_{pp} & 0 \\ 0 & B_{qq} \end{bmatrix} = \begin{bmatrix} V_{pp} & V_{pq} \\ V_{qp} & V_{qq} \end{bmatrix}^T \begin{bmatrix} A_{pp} & A_{pq} \\ A_{qp} & A_{qq} \end{bmatrix} \begin{bmatrix} V_{pp} & V_{pq} \\ V_{qp} & V_{qq} \end{bmatrix},$$

and since the Frobenius norm is preserved by orthogonal transformations $\|B\|_F = \|A\|_F$, and

$$\|B_{pp}\|_F^2 + \|B_{qq}\|_F^2 = 2\|A_{pq}\|_F^2 + \|A_{pp}\|_F^2 + \|A_{qq}\|_F^2.$$

If we define the sets P and Q as

$$P = \{(p-1)r+1, (p-1)r+2, \ldots, pr\}, \quad \text{and} \quad Q = \{(q-1)r+1, (q-1)r+2, \ldots, qr\},$$

then

$$\begin{aligned}
\sum_{i=1}^{n} b_{ii}^2 &= \sum_{i \notin P \cup Q} a_{ii}^2 + \sum_{i \in P} b_{ii}^2 + \sum_{i \in Q} b_{ii}^2 \\
&= \sum_{i \notin P \cup Q} a_{ii}^2 + \|B_{pp}\|_F^2 + \|B_{qq}\|_F^2 \\
&= \sum_{i \notin P \cup Q} a_{ii}^2 + 2\|A_{pq}\|_F^2 + \|A_{pp}\|_F^2 + \|A_{qq}\|_F^2 \\
&= \sum_{i \notin P \cup Q} a_{ii}^2 + 2\|A_{pq}\|_F^2 + \sum_{i \in P} a_{ii}^2 + \text{off}(A_{pp})^2 + \sum_{i \in Q} a_{ii}^2 + \text{off}(A_{qq})^2 \\
&= \sum_{i=1}^{n} a_{ii}^2 + 2\|A_{pq}\|_F^2 + \text{off}(A_{pp})^2 + \text{off}(A_{qq})^2,
\end{aligned}$$

and hence

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

∎

In this section we present two block Jacobi methods. Both methods are based on Algorithm Parallel Jacobi, given in Section 5. We will refer to these methods as *Method 1* and *Method 2*.

Method 1 is similar to a block Jacobi method for the SVD discussed in [119]. In Method 1, instead of quitting when $\text{off}(A)$ is small enough, we use a block analogue of this test:

$$\text{OFF}(A) = \sqrt{\sum_{i=1}^{k} \sum_{\substack{j=1 \\ j \neq i}}^{k} \|A_{ij}\|_F^2}.$$

By terminating when $\mathrm{OFF}(A)$ is small the final matrix A will be nearly block diagonal. The diagonalization process is then completed by computing, in parallel, the Schur decompositions of the diagonal blocks. In Algorithm Parallel Jacobi the 2-by-2 subproblems are exactly diagonalized. In Method 1, a complete diagonalization of the subproblems may not be necessary. So instead of computing the Schur decomposition (5.8.1), it might be sufficient to compute

$$\begin{bmatrix} B_{pp} & B_{pq} \\ B_{qp} & B_{qq} \end{bmatrix} = \begin{bmatrix} V_{pp} & V_{pq} \\ V_{qp} & V_{qq} \end{bmatrix}^T \begin{bmatrix} A_{pp} & A_{pq} \\ A_{qp} & A_{qq} \end{bmatrix} \begin{bmatrix} V_{pp} & V_{pq} \\ V_{qp} & V_{qq} \end{bmatrix},$$

where

$$\|B_{pq}\|_F \leq \theta \|A_{pq}\|_F, \tag{5.8.3}$$

for some fixed $0 \leq \theta < 1$. Justification for this suggestion is given later in this section. We can also incorporate a threshold in Method 1, and, as we show in Lemma 2.2, convergence can be ensured. In Method 1, we skip the (p, q) subproblem if

$$\|A_{pq}\|_F < \tau,$$

where the threshold τ satisfies

$$\tau \leq tol\|A\|_F/k, \quad tol > 0.$$

Before stating an algorithm for Method 1, we wish to specify the following problem, which arises when we want to implement a block Jacobi method on the KSR1. Consider, for convenience, the 4-by-4 block (symmetric) matrix

$$A = \begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & A_{34} \\ A_{41} & A_{42} & A_{43} & A_{44} \end{bmatrix},$$

and a system with two processors, p1 and p2. We send the block matrices

$$A_1 = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \quad \text{and} \quad A_2 = \begin{bmatrix} A_{33} & A_{34} \\ A_{43} & A_{44} \end{bmatrix},$$

Figure 5.8.1: The first alternative.

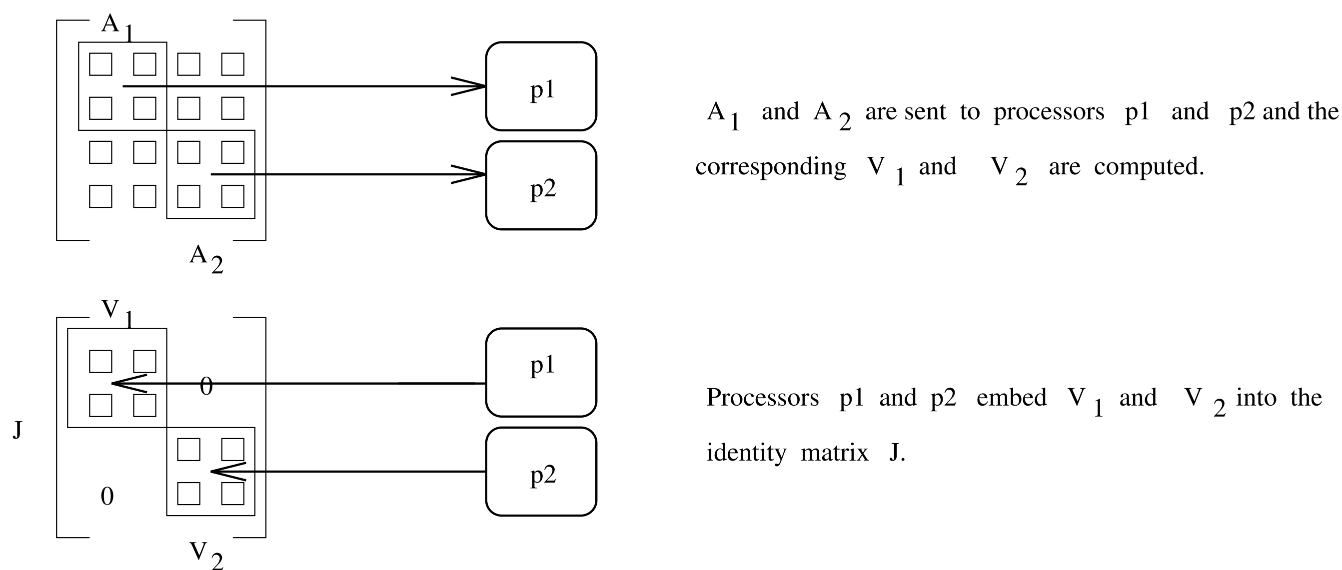

Figure 5.8.2: The second alternative.

to processors p1 and p2 respectively, and we ask them to compute the block matrices

$$V_1 = \begin{bmatrix} V_{11} & V_{12} \\ V_{21} & V_{22} \end{bmatrix}, \quad \text{and} \quad V_2 = \begin{bmatrix} V_{33} & V_{34} \\ V_{43} & V_{44} \end{bmatrix}.$$

The block matrices V_1 and V_2 may either satisfy a condition of the form (5.8.3), or actually diagonalize A_1 and A_2. The question which arises at this point is: Is it better to update the corresponding block rows and columns of A on p1 and p2, or to embed the blocks V_{ij} into the corresponding positions of an identity matrix, building a 4-by-4 complete block Jacobi rotation J, and then use both processors (parallel multiply) to form $J^T AJ$. These alternatives are illustrated in Figures 5.8.1 and 5.8.2. respectively. Thoughout this chapter we will refer to these ways of implementation as the *first* and the *second alternatives*. Method 1 may be sketched in the following algorithm.

Algorithm Method 1.
Given block k-by-k (symmetric) matrix $A \in \mathbf{R}^{n\times n}$, a tolerance $tol > 0$, and a threshold $\tau \leq tol\|A\|_F/k$, this algorithm first converts A into a block diagonal matrix, and then diagonalizes the block entries of the diagonal matrix. On successful exit, the matrix V^TAV contains the eigenvalues of the original matrix, and the columns of the orthogonal matrix V are the eigenvectors of A. The number of processors is P.

$V := I_n$;

$eps = tol\|A\|_F$;

while OFF(A) $\geq eps$

  for $i = 1{:}\,k-1$

    Generate the ith Jacobi set jac.set$(i) = \{(p_{il}, q_{il}), \quad l = 1, \ldots, n/2\}$,

    using Sameh's algorithm or the Tournament scheme;

    if the second alternative is being used $J := I_n$;

    for $l = 1{:}\,n/2$

      On processor $ps = 1 + ((l-1) \bmod P)$:

        $p = \min(p_{il}, q_{il})$

$q = \max(p_{il}, q_{il})$

if $\|A_{pq}\|_F > \tau$, either apply the first

alternative to update the p and q block rows and

columns of A and the p and q block columns of V,

or the second alternative to compute J;

end (of the parallel section)

end

if the second alternative is being used, **then**

Form $A = J^T AJ$ using parallel matrix multiply;

Form $V = VJ$ using parallel matrix multiply;

end if

end

end

%

% Second Stage of Method 1: The block diagonal matrix A is converted

% into a diagonal matrix, and the orthogonal matrix V is updated.

%

if the second alternative is being used $J:= I_n$;

for $i = 1: k/2$

On processor $ps = 1 + (i-1) \bmod P$:

Compute the Schur decomposition

$$\begin{bmatrix} A_{2i-1,2i-1} & A_{2i-1,2i} \\ A_{2i,2i-1} & A_{2i,2i} \end{bmatrix} = V_i \begin{bmatrix} D_{2i-1,2i-1} & 0 \\ 0 & D_{2i,2i} \end{bmatrix} V_i^T;$$

if the first alternative is being used, **then**

Update the $(2i-1)$st and $(2i)$th block rows and columns of A,

and the $2i-1$ and $2i$ block columns of V;

else

Embed the block V_i into the corresponding position of J;

end if

end (parallel section)

end

if the second alternative is being used, **then**

Form $A = J^T AJ$ using parallel matrix multiply;

Form $V = VJ$ using parallel matrix multiply;

end

As in scalar Jacobi methods, in block Jacobi methods we denote by a *block sweep* a single pass through the **while** loop. The following lemma ensures the convergence of Algorithm Method 1 in finite number of steps.

Lemma 5.8.2 *Algorithm Method 1 terminates after a finite number of block sweeps.*

Proof. If no subproblems are solved during a particular block sweep, then we have $\|A_{ij}\|_F \leq \tau$ for all i and j, $1 \leq i < j \leq k$. Hence,

$$\mathrm{OFF}(A)^2 = \sum_{i=1}^{k} \sum_{\substack{j=1 \\ j \neq i}}^{k} \|A_{ij}\|_F^2 \leq k(k-1)\tau^2 \leq \frac{k(k-1)}{k^2} tol^2 \|A\|_F^2 \leq tol^2 \|A\|_F$$

$$\Rightarrow \mathrm{OFF}(A) \leq tol \|A\|_F,$$

and termination is achieved. Assume now that the subproblem (p, q) is solved during a block sweep and the corresponding Jacobi rotation is an orthogonal matrix of the form (5.8.2). If we set $B = J^T AJ$ (A is the updated matrix), then

$$\begin{bmatrix} B_{pp} & B_{pq} \\ B_{qp} & B_{qq} \end{bmatrix} = \begin{bmatrix} V_{pp} & V_{pq} \\ V_{qp} & V_{qq} \end{bmatrix}^T \begin{bmatrix} A_{pp} & A_{pq} \\ A_{qp} & A_{qq} \end{bmatrix} \begin{bmatrix} V_{pp} & V_{pq} \\ V_{qp} & V_{qq} \end{bmatrix},$$

and since the Frobenius norm is preserved by orthogonal transformations

$$\|B_{pp}\|_F^2 + \|B_{qq}\|_F^2 + 2\|B_{pq}\|_F^2 = \|A_{pp}\|_F^2 + \|A_{qq}\|_F^2 + 2\|A_{pq}\|_F^2,$$

and $\|A\|_F = \|B\|_F$. Provided that $\|B_{pq}\|_F \leq \theta \|A_{pq}\|_F$, for a fixed $0 \leq \theta < 1$,

$$\|B_{pp}\|_F^2 + \|B_{qq}\|_F^2 \geq \|A_{pp}\|_F^2 + \|A_{qq}\|_F^2 + 2\|A_{pq}\|_F^2(1 - \theta^2).$$

Thus,

$$\begin{aligned}
\mathrm{OFF}(B)^2 &= \|B\|_F^2 - \sum_{i=1}^{k} \|B_{ii}\|_F^2 \\
&= \|A\|_F^2 - \sum_{i \neq p,q} \|A_{ii}\|_F^2 - (\|B_{pp}\|_F^2 + \|B_{qq}\|_F^2) \\
&\leq \|A\|_F^2 - \sum_{i \neq p,q} \|A_{ii}\|_F^2 - (\|A_{pp}\|_F^2 + \|A_{qq}\|_F^2 + 2\|A_{pq}\|_F^2(1-\theta^2)) \\
&= \mathrm{OFF}(A)^2 - 2\|A_{pq}\|_F^2(1-\theta^2) \\
&\leq \mathrm{OFF}(A)^2 - 2\tau^2(1-\theta^2). \qquad (5.8.4)
\end{aligned}$$

Inequality (5.8.4) implies that after ρ block sweeps, in each of which at least one subproblem is necessarily solved, the sum of norms of the off-diagonal blocks of the original matrix A, $\mathrm{OFF}(A)$, is diminished at least by

$$\rho\tau\sqrt{1-\theta^2}.$$

Hence, it follows that the condition $\mathrm{OFF}(A) \leq tol\|A\|_F$ must be satisfied after a finite number of steps. ∎

Lemma 5.8.2 clearly states that condition (5.8.3) is sufficient to ensure the convergence of Algorithm Method 1. Thus, it may not be necessary to compute the Schur decomposition of the block matrix

$$\begin{bmatrix} A_{pp} & A_{pq} \\ A_{qp} & A_{qq} \end{bmatrix}, \qquad (5.8.5)$$

but simply a block matrix

$$\begin{bmatrix} B_{pp} & B_{pq} \\ B_{qp} & B_{qq} \end{bmatrix}, \qquad (5.8.6)$$

such that

$$\|B_{pq}\|_F \leq \theta\|A_{pq}\|_F. \qquad (5.8.7)$$

During our numerical experiments, we observed that the number of block sweeps is a mildly increasing function of θ. The number of block sweeps has been observed to be usually minimal so long as $\theta \leq 0.25$. The block matrix (5.8.6) can be computed via any scalar Jacobi algorithm, parallel or serial. In our implementation we solve this subproblem

Sweep	OFF(A)	OFF(A)
1	1.79E+03	1.95E+03
2	5.95E+02	7.00E+02
3	3.19E+01	4.93E+01
4	1.03E-01	3.25E+00
5	3.60E-06	4.86E-01
6	4.03E-11	9.48E-02
7		1.95E-02
8		4.03E-03
9		8.32E-04
10		1.71E-04
11		3.41E-05
12		4.93E-11

Table 5.8.1: Numerical results for the partial Jacobi method.

on one processor using Algorithm Row Cyclic. We keep sweeping until condition (5.8.7) is satisfied. The fact that the subproblems are increasingly block diagonal as the iteration progresses encourages us to apply a threshold strategy. We found that Corneil-Kahan and Modified KC threshold strategies work well in this context. We will refer to the above method for solving the block 2-by-2 subproblem as the *partial Jacobi method*.

Although in the (p, q) subproblem we are interested in reducing the norm of A_{pq}, we eliminate all the off-diagonal entries of the matrix block (5.8.5) and not only on the entries of A_{pq}. Eliminating only the entries of A_{pq} increases the number of sweeps. This result is illustrated in Table 8.1 for the 64-by-64 symmetric matrix A,

$$a_{ij} = \begin{cases} i + j & \text{if } i \neq j, \\ i^2 + 64 & \text{otherwise.} \end{cases}$$

(The above matrix A is a matrix block in a subproblem and not the original symmetric matrix). For this symmetric matrix, Algorithm Method 1 needs 6 sweeps if the partial Jacobi method is applied on the off-diagonal entries in the (p, q) subproblem, and 12 sweeps if it is applied only on the entries of A_{pq}.

However, since the Schur decomposition of the block matrix (5.8.5) corresponds to $\theta \approx 0$, and hence Lemma 5.8.2 holds, the following question arises at this point: Is the partial Jacobi method faster than the LAPACK routine `SSEYV`, which computes the

Schur decomposition of a real symmetric matrix and also drives the updated matrix to a diagonal, instead of a block diagonal, form? The answer depends mainly on the size of the block matrix in a subproblem. It also depends on the value of θ and the threshold strategy but not in the same scale. Our conclusion, drawn by numerous numerical examples, is that it is better to use the partial Jacobi method with $\theta = 0.25$ for subproblems with size less than or equal to 8, and `SSYEV` for larger subproblems. Numerical and timing results for Algorithm Method 1 are reported in the next section, where we discuss the implementation of this algorithm on the KSR1.

Method 2, in its simplest form, is a block version of Algorithm Parallel Jacobi. The only difference is that instead of solving repeatedly 2-by-2 eigenvalue problems inside the **while** loop, Method 2 solves larger eigenvalue problems. The main difference between Method 1 and Method 2, is that Method 2 terminates when $\text{off}(A) < tol\|A\|_F$, computing always the Schur decomposition of the subproblems. Hence, Method 2 drives the updated matrix to a diagonal form. As for Method 1, the matrices A and V can be updated on each processor (first alternative), or the processors embed the matrix block rotations into a complete block Jacobi rotation J, and then they cooperate to form J^TAJ and VJ (second alternative). We can also apply a threshold strategy to Method 2, and since the updated matrix is driven to a diagonal form, we can also incorporate the Davies-Modi scheme into Method 2. As we show in Section 10, this scheme always improves the timing results. We experimented with various forms of Method 2 on the KSR1, using 16 processors, and our main experimental observations may be summarized as follows:

1. For relatively small n (for example $n = 64$), it is better to use the second alternative, and not to use threshold strategy.
2. The larger the n, the more preferable the first alternative with a suitable threshold strategy.
3. The Davies-Modi scheme always improves the timing results.
4. Method 2 is the fastest method known to us for solving large symmetric eigenproblems on the KSR1.

The algorithm for Method 2 is given in the next section, where we discuss its implementation on the KSR1 in detail. Numerical and timing results for Method 2 are given in Section 10.

5.9 Implementing Jacobi methods on the KSR1

In this section we discuss the implementation of the parallel Jacobi methods presented in the previous sections on the KSR1. We also report some numerical and timing results in order to state the advantages and disadvantages of these methods when they are implemented on the KSR1. Throughout this section the number of processors is assumed to be 16, which was the maximum number of processors available to us. The variable *tol* in the algorithms discussed in the previous sections is always set to nu, where $u \approx 1.1 \times 10^{-16}$ is the unit roundoff error for single precision arithmetics. For the matrix-matrix multiplications we used the highly optimized level 3 BLAS routine `SGEMM` supplied by Kendall Square Research in the KSRlib/BLAS Library [77]. For the generation of symmetric random matrices we used the LAPACK routine `SLATMR` [29]. From the parallel constructs provided by KSR Fortran, we used pthreads, parallel regions, and tile families.

We started with the implementation of Algorithm Parallel Jacobi. Algorithm Parallel Jacobi was found to be extremely slow when we wish to update A and V on each processor (first alternative). For example, for the 64-by-64 symmetric matrix A defined by

$$a_{ij} = \begin{cases} i+j & \text{if } i \neq j, \\ i^2 + 64 & \text{otherwise,} \end{cases} \tag{5.9.1}$$

Algorithm Parallel Jacobi with the first alternative requires 210.03 seconds without threshold strategy, and 175.43 seconds if we incorporate the Modified KC threshold strategy. On the other hand, the same algorithm with the second alternative requires 8.13 seconds. This performance was expected for the reasons explained in Section 5.

We investigated the performance of the following algorithms:

1. Algorithm Parallel Jacobi (PJ) (as given in Section 5) .

2. Algorithm Parallel Jacobi with modified KC threshold strategy (PJKC).

Algorithm	Sweeps	Seconds
PJ	6	8.13
PJKC	10	14.12
PJDM	5	6.79
PJKCDM	7	9.47

Table 5.9.1: Timing results for four scalar parallel Jacobi algorithms.

3. Algorithm Parallel Jacobi with Davies-Modi scheme (PJDM).

4. Algorithm Parallel Jacobi with modified KC threshold strategy and Davies-Modi scheme (PJKCDM).

In order to generate the Jacobi sets we used both schemes, that is Sameh's algorithm and the Tournament scheme. Since the timing results were almost identical for both schemes, we report the timing results for Sameh's algorithm. Table 9.1 reports the number of sweeps and the run times in seconds that the above algorithms need in order to compute the eigenvalues and the eigenvectors of the 64-by-64 matrix A in (5.9.1). Similar timing results have been observed in numerous numerical experiments with symmetric random matrices generated by `SLATMR`. We observe that the fastest algorithm is Algorithm PJDM which requires the smallest number of sweeps. That was expected, since the most expensive part of the above algorithms is the formation of the products J^TAJ and VJ, which occurs $n-1$ times at each sweep. Therefore, it is not surprising that the threshold strategy does not improve the timing results, since it increases the number of sweeps. However, even Algorithm PJDM becomes very slow when n is much greater than the number of processors. For example, for $n = 128$, a rather modest value of n, Algorithm PJDM requires 52.98 seconds and it is about 12 times slower than `SSYEV` on one processor. (Timing results for `SSEYV` are given in Table 10.1.) Therefore, scalar parallel Jacobi methods are not suitable for solving large symmmetric eigenvalue problems on KSR1 configurations with a modest number of processors. Nevertheless, Algorithm PJDM may perform better on systems with more processors. The source code for Algorithm PJDM is given in Appendix A.1.

We started the implementation of the parallel block Jacobi methods on the KSR1

Sweep	$\theta = 1.0E - 10$		$\theta = 0.25$		$\theta = 0.50$		$\theta = 0.75$	
1	1.74E+03	4	1.79E+03	2	1.85E+03	1	1.85E+03	1
2	4.85E+02	4	5.95E+02	2	6.91E+02	1	6.91E+02	1
3	2.29E+01	3	3.19E+01	1	6.25E+01	1	6.25E+01	1
4	5.51E-02	3	1.03E-01	1	4.35E-01	1	4.35E-01	1
5	2.00E-06	2	3.60E-06	1	3.54E-05	1	3.54E-05	1
6	2.72E-11	2	4.03E-11	1	4.43E-11	1	4.43E-11	1
Run Time	12.58		4.40		4.32		4.32	

Table 5.9.2: Some numerical and timing results for Method 1

with Algorithm Method 1. Since the number of processors is $P = 16$, we consider the original matrix as a block 32-by-32 matrix. For example, each block in a 64-by-64 matrix is 2-by-2, and the matrix blocks in the subproblems are 4-by-4. In general, if

$$A = \begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1,32} \\ A_{21} & A_{22} & \cdots & A_{2,32} \\ \vdots & \vdots & & \vdots \\ A_{32,1} & A_{32,2} & \cdots & A_{32,32} \end{bmatrix}, \quad A_{ij} \in \mathbf{R}^{r \times r},$$

then the block size is $r = n/32$, and the size of the subproblems is $2r$. Since each Jacobi set consists of 16 index pairs, the corresponding 16 subproblems may be solved simultaneously. Note that the same arrangement will be used for Method 2.

As we mentioned in the previous section, we can either diagonalize the matrix block (5.8.5) in the (p, q) subproblem, or we can apply a Row Cyclic Jacobi method (the partial Jacobi method) until the condition (5.8.7) is satisfied. We also stated that the latter approach is preferable for subproblems with size less or equal to 8. In order to investigate the role of θ in the partial Jacobi method, we experimented with different values of $\theta \in (0\ 1)$. Table 9.2 reports some numerical and timing results for the 64-by-64 matrix A given in (5.9.1). The left column for each θ reports the quantity $\mathrm{OFF}(A)$, and the right column the maximum number of sweeps for the partial Jacobi method at each block sweep. Table 9.2 also reports the run times. Similar results have been observed in a number of numerical experiments with 64-by-64 symmetric random matrices generated by `SLATMR`. We observe that Method 1 requires the same number of sweeps for each of

these four values of θ. However, for larger eigenproblems, the number of sweeps has been observed to be a mildly increasing function of θ. The choice $\theta = 0.25$ seems to be a good choice in practice, since it usually keeps the number of block sweeps at a minimum level. We also observe in Table 9.2, that the choices $\theta = 0.50$, and $\theta = 0.75$, have the same effect on the stages of convergence and the timing results. As expected, very small values of θ, as for example θ =1.0E-10, cause unnecessary delay in Algorithm Method 1, since they require more iterations for the subproblems.

As we mentioned in the previous section, when the size of the subproblems is greater that 8, it is preferable to use the LAPACK routine `SSYEV` instead of the partial Jacobi method for the subproblems. But in this case, Method 1 becomes a variant of Method 2, since the diagonalization of the subproblems drives the original symmetric matrix to a diagonal form, instead of a block diagonal form. Since Method 1 with the partial Jacobi method is not suitable for large matrices, we do not discuss the technical details concerning its implementation on the KSR1. However, the implementation of Method 1 with the partial Jacobi method is similar to the implementation of Method 2, which is discussed in the following paragraphs. The source code for Method 1 with the partial Jacobi method is given in Appendix A.2

Method 2 has been designed especially for large symmetric matrices. Our purpose was to develop a fast symmetric eigensolver for the KSR1 exploiting:

1. The particular architecture of the KSR1.
2. The highly optimized level 3 BLAS routines in the KSRlib/BLAS Library [77].
3. The Davies-Modi scheme.
4. A suitable threshold strategy that minimizes the number of the matrix-matrix multiplications.

In our implementation we used the first alternative for Method 2. The second alternative is not suitable for Method 2 for the following reasons:

1. Since Method 2 deals exclusively with large matrices, the formation of J^TAJ and VJ, which occurs $k-1$ times at each block sweep, is time consuming.

2. The second alternative does not exploit the special structure of the complete block Jacobi rotation J.

We experimented with various threshold strategies and we found that a block version of the modified KC threshold strategy gives the best all-round performance for Method 2. In the scalar case, if we follow the modified KC threshold strategy, we skip the annihilation of the entry $a_{pq}^{(j)}$ at the jth update, if $|a_{pq}^{(j)}| < \sqrt{\omega/N}$, where $\omega = \frac{1}{2}\text{off}(A_j)^2$, and $N = \frac{1}{2}n(n-1)$. In Method 2, having determined the $2r$-by-$2r$ matrix block

$$M = \begin{bmatrix} A_{pp} & A_{pq} \\ A_{qp} & A_{qq} \end{bmatrix}$$

for the (p,q) subproblem, we skip the solution of the (p,q) subproblem if

$$\frac{1}{2}\text{off}(M) = \sum_{i=2}^{2r}\sum_{j=1}^{i-1} m_{ij}^2 < \frac{2r(2r-1)}{2}\,\frac{\omega}{N}. \tag{5.9.2}$$

Provided that we use P processors and $2r = n/P$, (5.9.2) can be written as

$$\sum_{i=2}^{n/P}\sum_{j=1}^{i-1} m_{ij}^2 < \frac{1}{P}\,\frac{(n/P)-1}{n-1}\,\omega. \tag{5.9.3}$$

The threshold (5.9.3) is computed in the beginning of each block sweep. Since the computation of off(A) is necessary for the convergence test, the cost of computing the threshold (5.9.3) is negligible. Method 2 may be outlined in the following algorithm.

Algorithm Method 2.
Given a large n-by-n symmetric matrix A and a tolerance $tol > 0$, this algorithm computes the eigenvalues and the eigenvectors of A. On successful exit, the updated diagonal matrix A contains the eigenvalues of the original matrix A, and the columns of the orthogonal matrix V are the eigenvectors of the original matrix. The number of processors is P.

$V := I_n$;

$eps = tol\|A\|_F$;

$SWEEP := 0$;

$MODI := FALSE$;

while off(A) $> eps$ **and** $MODI = FALSE$

$SWEEP = SWEEP + 1$;

Compute the threshold (5.9.3);

for $set = 1\colon 2P - 1$

Determine the P subproblems using Sameh's algorithm or the tournament scheme, and send the matrix block M_i to processor P_i ($i = 1, \ldots, P$);

On processor P_i:

if the half sum of squares of the off-diagonal entries of M_i is greater or equal to threshold, **then**

Compute the Schur decomposition $M_i = V_i D_i V_i^T$;

Use V_i to update the ith block columns of A and V, and the ith block row of A;

end if

end (parallel section)

end

if $SWEEP > 9$ **then**

Compute the quantities α, δ, and ϵ as in Algorithm Davies-Modi;

if $\epsilon < 10^{-3}$, $MODI = TRUE$;

end if

end

if off(A) $> eps$, compute the matrices A and V as in Algorithm Davies-Modi.

Note that in the above algorithm, the quantities α, δ, and ϵ are computed after the 9th block sweep for the reasons explained in Section 7. The ith block column of A in the

above algorithm is the n-by-$2r$ matrix block

$$\begin{bmatrix} A_{1p} & A_{1q} \\ A_{2p} & A_{2q} \\ \vdots & \vdots \\ A_{kp} & A_{kq} \end{bmatrix}, \quad A_{ij} \in \mathbf{R}^{r \times r}, \tag{5.9.4}$$

where (p, q) is the index pair that corresponds to the ith subproblem ($i = 1, \ldots, P$). Similarly, the ith block row of A is the $2r$-by-n matrix block

$$\begin{bmatrix} A_{p1} & \cdots & A_{pk} \\ A_{q1} & \cdots & A_{qk} \end{bmatrix}, \quad A_{ij} \in \mathbf{R}^{r \times r}. \tag{5.9.5}$$

The construction of the matrix blocks (5.9.4) and (5.9.5) can be accomplished using the array syntax, a language extension of KSR Fortran discussed in Chapter 2.

The main parallel part of Algorithm Method 2, that is the part of the algorithm stated as parallel section[7], has been implemented in KSR Fortran using two parallel regions. In the first parallel region, each processor P_i is instructed to perform the following tasks:

1. To construct the matrix block M_i that corresponds to the ith index pair of the current Jacobi set.
2. To check whether condition (5.9.3) is satisfied, and if so to set a logical variable $THR(i)$ equal to $FALSE$ and return. Otherwise, to compute the Schur decomposition $M_i = V_i D_i V_i^T$, and use V_i to update the ith block columns of A and V.

In the second parallel region, if the value of the logical variable $THR(i)$ is $TRUE$, processor P_i updates the ith block row of A. In our implementation we used a team of 16 pthreads. Tiling has been used where it was necessary, that is in big loops. The source code for Algorithm Method 2 is given in Appendix A.3. Numerical and timing results for Method 2 and some of its variants are presented in the following section.

[7] This is not the parallel construct of KSR Fortran.

5.10 Numerical and Timing Results for the SEP

We start this section discussing a modified form of the LAPACK symmetric LAPACK eigensolver `SSYEV` and investigating its performance on the KSR1. `SSEYV` computes the eigenvalues and, optionally, the eigenvectors of a symmetric matrix A. The computation proceeds in the following stages:

1. The decomposition $A = QTQ^T$ is computed, where Q is an orthogonal matrix, and T is a symmetric tridiagonal matrix.
2. The eigenvalues and eigenvectors of T are computed. This is equivalent to factorizing T as $T = S\Lambda S^T$, where S is orthogonal and Λ is diagonal.

The diagonal entries of Λ are also the eigenvalues of A. The columns of $Z = QS$ are the eigenvectors of A, and hence the Schur decomposition of A is $A = Z\Lambda Z^T$ [1].

Using `SSYEV` for solving symmetric eigenproblems, the only calls to the highly optimized level 3 BLAS routine `SGEMM` are made from the LAPACK routine `SLARFB`. The purpose of `SLARFB` is to apply a block reflector or its transpose, to a matrix. We modified `SLARFB` adding the four additional parameters to the calls of `SGEMM` from the body of this routine. This modification allows us to use more than one processors for `SSYEV`[8]. Table 10.1 reports the run times in seconds for standard `SSYEV` on one processor, and for the modified `SSYEV` on 16 processors for various values of n. The symmetric dence matrices for these measurements have been generated using the LAPACK routine `SLATMR`, and their entries follow the uniform distribution. Similar timings have been observed in a number of numerical experiments with symmetric random matrices, and thus the timing results displayed in Table 10.1 can be considered as typical.

As we mentioned in the previous section, Method 2 has been designed especially for large matrices. Table 10.2 reports the run times for Method 2, as stated in Algorithm Method 2, for the same symmetric random matrices as in Table 10.1. As expected, Method 2 is slower than `SSYEV` and modified `SSYEV` for $n = 64$, $n = 128$, and $n = 256$.

[8]If we do not specify the four extra parameters in the calls of `SGEMM`, then there is no parallelism within `SGEMM`. In this case, `SSYEV` cannot take any advantage of the use of more than one processors.

Order	Standard SSYEV	Modified SSYEV
64	0.71	0.29
128	5.58	1.78
256	40.51	17.01
512	369.56	148.53
1024	2534.84	1150.37

Table 5.10.1: Timing results for SSEYV.

Order	Method 2
64	24.58
128	39.51
256	96.47
512	145.35
1024	518.20

Table 5.10.2: Timing results for Method 2.

On the other hand, Method 2 is about as fast as modified `SSYEV` for $n = 512$, and significantly faster than modified `SSYEV` for $n = 1024$. In order to have a fair comparison between `SSYEV` and Method 2, our timings include the sorting of the eigenvalues, and the corresponding exchange of columns for the orthogonal matrix V. The Frobenius norm of the 1024-by-1024 random symmetric matrix in Table 10.2 is about $5.0E+03$.

In order to illustrate the effect of the Davies-Modi scheme and the threshold strategy's on timing results, we report some numerical and timing results for some variants of Method 2, for the above 1024-by-1024 random symmetric matrix. We consider the following variants of Algorithm Method 2:

1. Algorithm Method 2 without the Davies-Modi scheme and threshold strategy (Variant 1).
2. Algorithm Method 2 without the Davies-Modi scheme (Variant 2).
3. Algorithm Method 2 without the Davies-Modi scheme and with the threshold strategy applied only in the first 6 block sweeps (Variant 3).
4. Algorithm Method 2 without the threshold strategy (Variant 4).

Sweep	off(A)
0	5.0E+03
1	1.8E+03
2	7.9E+02
3	4.5E+02
4	2.5E+02
5	1.2E+02
6	4.5E+01
7	1.8E+01
8	7.2E+00
9	2.5E+00
10	9.7E-01
11	3.2E-01
Final off(A)	3.8E-10
Run Time	518.20

Table 5.10.3: Numerical and timing results for Method 2.

Sweep	off(A)
0	5.0E+03
1	1.2E+03
2	7.4E+02
3	2.6E+01
4	1.8E-01
5	3.6E-06
6	9.1E-12
Run Time	2131.22

Table 5.10.4: Numerical and timing results for Variant 1.

Table 10.3 reports the progress of convergence, the final off(A) (after the application of the Davies-Modi scheme), and the run time in seconds for Method 2, as stated in Algorithm Method 2. According to Table 10.3, the condition for the Davies-Modi scheme to be used, is satisfied at the 11th block sweep.

Table 10.4 reports the progress of convergence and the run time in seconds for Variant 1. We observe that Variant 1 is about four times slower than Method 2. Variant 1 requires less block sweeps than Method 2. But since there is no threshold strategy, Variant 1 requires more matrix-matrix multiplications than Method 2. Matrix-matrix

Sweep	off(A)	Sweep	off(A)
0	5.0E+03	13	2.0E-02
1	1.8E+03	14	5.0E-03
2	7.9E+02	15	1.2E-03
3	4.5E+02	16	2.9E-04
4	2.5E+02	17	6.6E-05
5	1.2E+02	18	1.4E-05
6	4.5E+01	19	2.7E-06
7	1.8E+01	20	4.6E-07
8	7.2E+00	21	8.6E-08
9	2.5E+00	22	1.2E-08
10	9.7E-01	23	2.0E-09
11	3.2E-01	24	3.5E-10
12	8.6E-02		
Run Time		1201.25	

Table 5.10.5: Numerical and timing results for Variant 2.

multiplications is the most time consuming part of Method 2 and this explains the slowness of Variant 1. During our experiments with various threshold strategies for Method 2, we noticed that the adopted threshold strategy keeps the number of matrix-matrix multiplications at a minimum level. However, applying only the threshold strategy without Davies-Modi scheme (Variant 2), Method 2 requires a large number of block sweeps and this causes significant delay. As Table 10.5 illustrates, the absence of Davies-Modi scheme in Variant 2 increases significantly the run time.

As we mentioned in Section 7, the Davies-Modi scheme theoretically fails when the original symmetric matrix has multiple eigenvalues. We experimented with Method 2 on various symmetric matrices with multiple eigenvalues. In some instances the execution of our program was stopped due to division by zero. In other instances, Method 2 kept sweeping until the Davies-Modi condition was satisfied. But in this case, the final values of off(A), $\|V^TV - I_n\|_F$, and $\|A - VDV^T\|_F/\|A\|_F$, where D is the nearly diagonal matrix of the computed eigenvalues and V the matrix of the eigenvectors, are not satisfactory. For example, for a 64-by-64 symmetric randon matrix with one eigenvalue with multiplicity 16, Method 2 gave the following results:

$$\text{off}(A) = 1.74E{-}01, \quad \|V^TV - I_n\|_F = 2.32E{-}01, \quad \text{and} \quad \|A - VDV^T\|_F/\|A\|_F = 1.16E{-}01.$$

t	Sweeps	off(A)	$\|V^TV - I_n\|_F$	$\|A - VDV^T\|_F/\|A\|_F$
-1	4	2.19E-14	1.68E-14	6.86E-15
3	4	2.82E-13	5.96E-13	4.30E-13
6	5	3.06E-10	6.41E-10	2.80E-10
9	5	1.49E-07	3.32E-07	2.22E-07
12	6	3.60E-04	7.50E-04	4.00E-04

Table 5.10.6: Numerical results with matrices with close eigenvalues.

The above results clearly state the ineffectiveness of Method 2 for matrices with multiple eigenvalues. Similar behaviour of Method 2 has been observed in a number of numerical experiments with other symmetric matrices with multiple eigenvalues. We also experimented with matrices with some eigenvalues close to each other. Table 10.6 reports some numerical results for the above-mentioned 64-by-64 symmetric matrix whose 16 identical eigenvalues have been perturbed by the addition of the quantities $\alpha \times 10^{-t}$, where $\alpha \in [\ 0\ 1\ ]$ is a random number from the uniform distribution, and $t = 3, 6, 9, 12$ and -1. ($t = -1$ corresponds to a matrix with different eigenvalues). Table 10.6 indicates that Method 2 may be inefficient for matrices with a number of almost identical eigenvalues. Thus it might be better in this case to avoid Davies-Modi scheme, which makes Method 2 unstable. On the other hand Variant 2, which avoids the Davies-Modi scheme, requires a large number of block sweeps. In this case, the best compromise is to apply the threshold strategy during the m first block sweeps, and then to disengage the threshold procedure from the program (Variant 3). A number of numerical experiments showed that $m = 6$ is a good choice in practice. Table 10.7 reports some numerical and timing results for Variant 3 for the same 1024-by-1024 symmetric random matrix. We observe that even without Davies-Modi scheme, Method 2 is a little faster than modified `SSEYV` for this matrix.

Table 10.8 reports the progress of convergence and the run time in seconds for Variant 4 for the same 1024-by-1024 matrix. As expected, Variant 4 requires the smallest number of sweeps since the application of the threshold strategy increases the number of block sweeps. However, it is noteworthy that Variant 4 is something less than three times slower than Method 2. This timing result seems to be weird at first sight. One may

Sweep	off(A)
0	5.0E+03
1	1.8E+03
2	7.9E+02
3	4.5E+02
4	2.5E+02
5	1.2E+02
6	4.5E+01
7	2.3E+00
8	7.1E-04
9	3.0E-11
Run Time	1014.07

Table 5.10.7: Numerical and timing results for Variant 3.

expect from 16 processors to need the same time to update up to 16 block rows and block columns, and the threshold strategy not to effect the timing results. Moreover, since Variant 4 requires less sweeps than Method 2, Variant 4 may be expected to be faster than Method 2. This situation may be explained as follows. Each block column and block row is updated on a separate processor by a pthread. The application of the threshold strategy causes some pthreads to remain idle. These idle pthreads proceed to a barrier and wait for the busy pthreads, which update the block columns or rows, to join them. The overall data movement inside the memory system of the KSR1 is reduced because of the application of the threshold strategy. The time required from 16 processors to update all the block columns and rows has always been measured to be less than the time needed for updating only some of them. For example, consider the 64-by-64 matrix A given in (5.9.1). Because of the threshold startegy, the first block row update of Method 2 involves 7 block row updates, and 0.07 seconds are needed for this update. Variant 4 updates 16 block rows in the first update in 0.15 seconds.

In Table 10.9 we summarize the timing results for Method 2 and its variants for the same 1024-by-1024 symmetric random matrix. The right column of Table 10.9 states the reduction in run time for each approach in comparison with Variant 1, which takes no advantage of either Davies-Modi scheme or the threshold strategy. Similar timing results have been observed in a number of numerical experiments with other large symmetric

Sweep	off(A)
0	5.0E+03
1	1.2E+03
2	7.4E+02
3	2.6E+01
4	1.8E-01
Final off(A)	6.3E-11
Run Time	1455.00

Table 5.10.8: Numerical and timing results for Variant 4.

Method	Run Time	Reduction
Method 2	518.20	76%
Variant 1	2131.22	-
Variant 2	1201.25	44%
Variant 3	1014.07	52%
Variant 4	1455.00	32%

Table 5.10.9: Timing results for Method 2 and its variants.

random matrices. According to Table 10.9, the fastest variant of Method 2 is Variant 3. This variant of Method 2 may be used as an alternative of Method 2 when we deal with large symmetric matrices, with identical or almost identical eigenvalues, in order to avoid stability problems due to Davies-Modi scheme.

5.11 Numerical and Timing Results for the SVD

As we mentioned in the introduction of this chapter, our purpose is to compute the SVD from the polar decomposition. In Chapter 4 we presented a parallel algorithm for computing the polar decomposition. In Section 4.8 we presented timing results for 1024-by-1024 random matrices with various 2-norm condition numbers. According to these results, using 16 processors, the polar decomposition of these matrices can be computed in the best case in 184.17 seconds (when the matrix is close to orthogonal and scaling is not used), and in the worst case in 1868.04 seconds (when the 2-norm condition number of the matrix is 10^{12} and scaling is not used). Thus, given a 1024-by-1024 matrix A, the

run time needed for the computation of the polar decomposition

$$A = UH,$$

on the KSR1 using 16 processors, is expected to be between 200 and 2000 seconds. Having obtained the polar factors U and H of A, we may continue to compute the spectral decomposition

$$H = VDV^T, \qquad (5.11.1)$$

of the Hermitian positive semidefinite factor H. For the computation of the above spectral decomposition, we may use either Method 2 (or in case of multiple or almost identical eigenvalues Variant 3), or the modified LAPACK symmetric eigensolver `SSYEV`. In the former case, the required run time is about 520 seconds (or about 1020 seconds for Variant 3), and in the latter case about 1150 seconds, both times being essentially independent of the eigenvalue distribution. Having obtained the spectral decomposition (5.11.1), we can compute the SVD of A,

$$A = (UV)DV^T,$$

in 5.6 seconds using the highly optimized level 3 BLAS routine `SGEMM`. The total run times are given in Figure 5.11.1.

As we mentioned in Section 3.3, the standard approach to the computation of the SVD is the Golub-Reinsch algorithm [46], as implemented in LAPACK. The corresponding LAPACK routine is `SGESVD` discussed in Section 3.3. It is beyond the scope of this thesis to develop a highly optimized version of this sophisticated routine for the KSR1. However, we did the following modifications in order to exploit the highly optimized level 3 BLAS routine `SGEMM`, and use more than one processors:

1. We added the additional parameters to the calls of `SGEMM`.
2. We set the block size equal to 16 in the routines that make calls to `SGEMM`[9].

(The same approach has been followed for the matrix inversions in Algorithm Parallel Polar (`SGETRF`, `SGETRI`), and the symmetric eigenvalue problem (`SSEYV`).) From the set

[9] Having experimented with the block size in the routines that make calls to `SGEMM`, we found that a block size of 16 gives the best all-round performance on the KSR1.

Figure 5.11.1: Computing the SVD via the Polar Decomposition and the SEP.

of LAPACK routines that are necessary for SGESVD, only three routines make calls to SGEMM. These are, the driver routine SGESVD, SLARFB discussed in modified SSYEV, and SGEBRD. SGEBRD reduces a general m-by-n matrix A to upper or lower bidiagonal form B by an orthogonal transformation $Q^T AP = B$. The matrices Q and P are represented as products of elementary reflectors. Without any modification, SGESVD needs about 9500 seconds to compute the SVD of a 1024-by-1024 matrix on one processor. The best timing result that we achieve for this dimension, using the modified SGESVD, was 7549.02 seconds on 16 processors.

According to the above mentioned timings, the fastest way known to us to compute the SVD of a large dense matrix on the KSR1 is to compute first its polar decomposition using Algorithm Parallel Polar, and then to compute the Schur decomposition of its symmetric positive semidefinite factor using Algorithm Method 2. However, this combination should be avoided for matrices whose 2-norm condition number is close to 1, although it is more than 10 times faster than the modified SGESVD. In this case, the eigenvalues of H

(which are the singular values of A) are clustered around 1, and thus the application of the Davies-Modi scheme may cause numerical instability, if not failure due to division by zero. But even when we use Algorithm Variant 3 or modified `SSYEV` for the Schur decomposition, this approach still gives better timings than the modified `SGESVD`, the SVD solver in LAPACK. Thus this new approach to computing the SVD seems to be suitable for applications where the SVD of large dense matrices is required.

5.12 Applications of the SVD

In this section we discuss some applications of the SVD. This section is divided into two subsections. In the first subsection we discuss the use of the SVD in regression analysis. In the second subsection we discuss some less well-known applications of the SVD.

5.12.1 The SVD in Regression Analysis

In this subsection we discuss the use of the SVD in multiple linear regression, with special reference to the problems of collinearity and near collinearity. The greatest source of difficulties in using least squares for solving multiple linear regression problems is the existence of collinearity in many data sets [87]. Most of the modifications of the ordinary least squares approach are attempts to deal with the problem of collinearity. Among these modifications is the method of *principal components* [34, 52, 55].

We assume that the model is known and of the form

$$Y = X\beta + e, \tag{5.12.1}$$

where Y, $e \in \mathbf{R}^n$, $X \in \mathbf{R}^{n\times p}$ $(n \geq p)$, and $\beta \in \mathbf{R}^p$. The matrix X, consisting of nonstochastic entries, is given. The vector Y consists of the measurements y_i, and the vector e is the error term. The errors e_i are assumed to be uncorrelated, of zero mean and constant variance σ^2, the value of which is not known. The object of regression analysis is to estimate the coefficients β_i, as well as σ^2, to predict the value of y for any future regressor variables $x = (x_1\ x_2 \ldots x_p)$, and to estimate the error of such a predicted value.

Assuming that the rank of the matrix X in (5.12.1) is r, the SVD of X may be written as

$$X = U\Theta V^T, \tag{5.12.2}$$

where $U \in \mathbf{R}^{n\times r}$, $\Theta \in \mathbf{R}^{r\times r}$, ($\Theta = \text{diag}(\theta_1, \theta_2, \cdots, \theta_r)$, $\theta_1 \geq \theta_2 \geq \ldots \theta_r > 0$), and $V^T \in \mathbf{R}^{r\times p}$, since in practice one needs only the first r columns of the orthogonal factors U and V. Introducing (5.12.2) into (5.12.1), we obtain

$$Y = U\Theta V^T\beta + e.$$

Written in this form, the model is referred to as the *principal component regression model* and the procedure of carrying out the regression of Y on X using the SVD of X is called *principal components regression*. The columns of the matrix $W = XV = U\Theta$ are called the *principal components* of the matrix X and their practical importance may be illustrated by the following example, taken from [100].

The first column of the 6-by-4 matrix

$$X = \begin{bmatrix} .1781 & -.5232 & .0591 & -.0610 \\ .4499 & -.2093 & .7780 & .3012 \\ -.1480 & .3009 & -.2106 & -.0534 \\ -.0574 & .0654 & .1206 & -.0572 \\ -.7820 & -.3270 & -.2105 & -.7323 \\ .3593 & .6933 & -.5368 & .6029 \end{bmatrix}$$

reports the average minimum daily temperature, the second the average maximum daily temperature, the third the total rainfall, and the fourth the total growing degree days. The above data have been measured at six different locations and the purpose of this research is to relate environmental conditions to cultivar-by-environment interactions in sorghum. The variables that corresponds to each column have been centered to have zero mean, and standardized to have unit sum of squares. The SVD of X is

$$X = U\Theta V^T,$$

where

$$U = \begin{bmatrix} -.1140 & .3089 & -.8107 & .2598 \\ .2519 & .7076 & .3397 & -.3195 \\ .0076 & -.3032 & .2774 & .5681 \\ -.0281 & .0278 & .3266 & .3566 \\ -.7354 & -.2350 & .0655 & -.4815 \\ .6180 & -.5060 & -.1986 & -.3855 \end{bmatrix}, \quad V = \begin{bmatrix} .5949 & .3362 & -.3832 & .6214 \\ .4518 & -.5407 & .6580 & .2657 \\ .0048 & .7687 & .6390 & -.0265 \\ .6647 & .0610 & -.1090 & -.7366 \end{bmatrix},$$

and

$$\Theta = \text{diag}(1.4970,\ 1.2449,\ .4541,\ .0579).$$

The first column of U, u_1, the first column of V, v_1, and the first singular value $\theta_1 = 1.4970$, give the best rank-1 approximation of X,

$$X_1 = \theta_1 u_1 v_1^T.$$

The goodness of fit of X_1 to X is measured by

$$\frac{\theta_1^2}{\sum_{i=1}^4 \theta_i} = .56,$$

or the sum of squares of the differences between the entries of X and X_1, the lack of fit, is 44% of the total sum of squares of the entries in X. The rank-2 approximation to X is obtained by adding to X_1 the matrix $X_2 = \theta_2 u_2 v_2^T$, and $X_1 + X_2$ has goodness of fit

$$\frac{\theta_1^2 + \theta_2^2}{\sum_{i=1}^4 \theta_i} = .95.$$

In terms of approximating X with the rank-2 matrix $X_1 + X_2$, the goodness of fit of .95 is interpreted that the sum of squares of discrepancies between X and $X_1 + X_2$ is 5% of the total sum of squares of all entries in X. In terms of geometry of the data vectors, the goodness of fit of .95 means that 95% of the dispresion of the cloud of points in the original four-dimensional space, in reality, very nearly fall on a two dimensional plane. Because of the relatively small size of the third and fourth singular values, the last two dimensions contain little of dispersion and can safely be ignored in the interpretation of the data.

Each column of V contains the coefficients that define one of the principal components as a linear function of the original variables. The first vector $v_1^T = (.5949, .4518, .0048, .6647)$ has similar first, second, and fourth coefficients with the third coefficient being near zero. Thus, the first principal component is essentially the average of the three temperature variables. The second column $v_2^T = (.3362, -.5407, .7687, .0610)$ gives heavy positive weight to third variable, heavy negative weight to second, and moderate positive weight to first. Hence, the second principal component will be large for those observations that have high raifall and large difference between the maximun and minimum daily temperatures. The interpretation of the third and fourth principal components is not meaningful since they account for only 5% of the total dispersion.

The principal components vectors for the above example are the columns of the matrix

$$W = \begin{bmatrix} -0.1707 & 0.3845 & -0.3681 & 0.0151 \\ 0.3771 & 0.8808 & 0.1542 & -0.0185 \\ 0.0114 & -0.3775 & 0.1260 & 0.0329 \\ -0.0420 & 0.0346 & 0.1483 & 0.0207 \\ -1.1008 & -0.2925 & 0.0297 & -0.0279 \\ 0.9252 & -0.6299 & -0.0902 & -0.0223 \end{bmatrix}.$$

The sum of squares of the first, second, third, and fourth principal components are θ_1^2, θ_2^2, θ_3^2 and θ_4^2 respectively. These sum to 4.0, the total sum of squares of the original four variables after their standardization. The proportion of the total sum of squares accounted for by the first principal component is $\theta_1^2/\sum_{i=1}^4 \theta_i = .56$ or 56%. The first two principal components account for $(\theta_1^2 + \theta_2^2)/\sum_{i=1}^4 \theta_i = .95$ or 95% of the total sum of squares of the four original variables. Each of the original vectors in X is a vector in six-dimensional space, and together the four vectors define a four-dimensional subspace. These vectors are not orthogonal. The four vectors in W, the principal component vectors, are linear functions of the original vectors and as such, they fall in the same four-dimensional subspace. Moreover, the principal component vectors are orthogonal. The first component vector has the largest possible sum of squares and defines the direction of the first principal component axis. This axis coincides with the major axis of the ellipsoid

Figure 5.12.1: The first two principal components.

of observations. The second principal component has the largest possible sum of squares of all vectors orthogonal to the first and so on.

The plot of the first two principal components for the above example is given in Figure 5.13.1. We observe that locations 5 and 6 differ from each other primarily in the first principal component. This component was noted earlier to be mainly a temperature difference. Thus, location 6 is the warmer and has the longer growing season. The other four locations differ primarily in the second principal components which reflects the amount of rainfall and the difference in maximum and minimum daily temperature. Location 2 has the highest raifall and tends to have a large difference in maximum and minimum daily temperature. Location 6 is also the lowest in the second principal component, indicating a lower rainfall and small difference between the maximum and minimum temperature. Hence, location 6 appears to be relatively hot, dry environment with little diurnal temperature variation.

The above example has been chosen for purposes of illustration and the data matrix X has small size (6×4). However, in reality, the data matrices that arise in the applications of regression analysis have much larger size. Hence, the need for fast and reliable

methods for the computation of the SVD is apparent. Such examples are encountered in various fields of the scientific research. These include seismology [19], demography [85], research in education [111], environmental research [58], anthropometric and physical fitness measurements [52], and social-economic research [89]. Applications with very large data matrices are mainly encountered in econometrics [105]. The principal component method is also used in correspondence analysis [50, 84]. Correspondence analysis itself has numerous applications. One of the most interesting applications of correspondence analysis is the analysis of lexical data, where the computation of the SVD of very large data matrices is required [9].

5.12.2 Some Less Well-known Applications of the SVD

In this subsection we discuss four less-known applications of the SVD. The first of these applications arises in chemistry and is discussed in [112]. In chemical titration experiments a known substance, which is called *titrant*, is being added to an unknown substance or mixture. Gradually, the substances in the mixture change for some initial state through possible intermediate states to a final state, with the fraction of each substance in each state being controled by the concentration of titrant. After each addition of titrant, allowing sufficient time for the mixture to equilibrate, a sprectrum of the mixture is taken. The spectrum of the mixture is the sum of the spectra of the individual species, and the amplitude of a spectrum of a species is proportional to the concentration of that species. The object of a titration experiment is the identification of the unknown substances in the original mixture.

The measured optical spectra are stored in successive columns of the matrix A, so that the entry a_{ij} is the optical absorbance of the mixture at the ith wavelength in the jth spectrum. The target decomposition is

$$A = DF^T + E,$$

where each column of D is a difference spectrum associated with one of the transitions, the corresponding column of F is the appropriate transition curve, and E is the matrix

of the experimental errors. The transition curve is given by the function

$$f(\text{pH}; \text{pK}) = 1/(1 + 10^{\text{pK}-\text{pH}}),$$

where $\text{pH} = -\log_{10}[H^+]$ ($[H^+]$ is the concentration of protons of hydrogen ions), and pK is the value of pH at which half the sites of a given indicator are saturated.

In many examples, the basic spectrum, namely the final column of D, raises the effective rank of A without adding any information. The work required of the user is proportional to the effective rank of A, and it is current practice to subtract a reference spectrum from all the columns of A. This reference spectrum is usually the initial spectrum of A or the average of all spectra in A, and the effective rank of A is reduced by one.

The required matrices D and F are determined using the SVD of $A \in \mathbf{R}^{m \times n}$, $(m \geq n)$,

$$A = DF^T + E = U\Sigma V^T, \quad U \in \mathbf{R}^{m \times n}, \quad \Sigma, V \in \mathbf{R}^{n \times n}. \tag{5.12.3}$$

Assuming that a reference spectrum has been subtracted from the columns of A, the rank of the matrix $DF^T + E$ in (5.12.3) is $n - 1$. However, the matrix DF^T is typically low rank. In order to deduce the matrices D and F, only the first r columns of U, Σ, and V (and rows of Σ) are retained. The value of r is chosen such that

$$\sum_{i=r+1}^{n} \sigma_{ii}^2 \leq mn\sigma^2 < \sum_{i=r}^{n} \sigma_{ii}^2,$$

where σ^2 is the variance of each entry of A, and the corresponding truncated $r \times r$ matrices are $\bar{U}$, $\bar{\Sigma}$, and $\bar{V}$. The columns of $\bar{V}$ are linear combinations of the columns of F and the matrix F is determined by a series of curve-fitting operations. The curve-fitting operations, the most difficult and time-consuming phase of a titration experiment, determine a matrix H of parameters which satisfies the relation

$$\bar{V}^T = HF^T. \tag{5.12.4}$$

It follows directly from (5.12.4) that

$$D = \bar{U}\bar{\Sigma}H.$$

A matrix A with hundreds of entries in each dimension is likely to be encountered in a titration experiment with Gaussian absorbance peaks, and Henderson-Hasselback transition curves [112]. The size of A can sometimes be limited by processing only the data of a submatrix at a time. However, this may not be necessary if one uses a fast algorithm on a parallel machine for the computation of the SVD, as for example one of the methods discussed in Section 10.

An application of the SVD to manipulability and sensitivity of industrial robots is discussed in [117]. In designing and evaluating industrial robots, it is important to find optimum configurations, or postures, and locate optimum points in the workspace for the anticipated tasks. The ideal manipulator would have no sigularities or degeneracies in its workspace in order to obtain full mobility throughout its range of motion. However, this is not always achievable in practice and a measure of the nearness to the degeneracy is needed. This measure is called *manipulability*. In [117], Togai defines the manipulability M of a robot arm as the condition number of its Jacobian, and proposes the 2-norm condition number for its measurement. The best conditioning possible in terms of manipulability occurs when $M = \sigma_{\max}(J)/\sigma_{\min}(J) = 1$. This situation corresponds to the ideal manipulator, and the corresponding optimum points in the workspace are called *isotropic points*. These points may or may not exist for a given design. In [117], Togai does not give any information about the dimensions of the Jacobian matrix J.

In [2], Andrew and Patterson investigate the use of the SVD for the solution of the following problem which arises in digital image compression: Given a gray level image A of dimension $n \times n$ with t bits for the gray level of each of the n^2 pixels, store an approximation to A in less than tn^2 bits. If

$$A = U\Sigma V^T, \quad U, \Sigma, V \in \mathbf{R}^{n\times n},$$

is the SVD of A, then the image A may be approximated by the outer-product expansion

$$A \approx \sum_{k=1}^{r} \sigma_k u_k v_k^T.$$

Generally, r needs to be rather large in order to approximate A well. However, more

recent research in digital image processing has shown that factorizations of the form

$$A = X\Omega Y^T,$$

where the orthogonal matrices X and Y are independent of A (Hamadard transform and Discrete Cosine transform), are more suitable than the SVD for the solution of the above problem [23].

In [18], Broomhead and Lowe discuss a least squares problem that arises in Radial Basis Function Networks (RBFN). In neural networks there are two learning processes which are known as *supervised* and *unsupervised learning*. The mechanics of RBFN architecture uses the supervised learning procedure. A neural network has two phases, which are called *training* and *testing*. During the training phase in a supervised learning process, the network is being *fed* with input vectors and their corresponding output vectors which are also known. The output vectors constitute a $k \times n$ matrix O. A $k \times m$ matrix R which is associated with the matrix I whose rows are the input vectors, is also given [60]. Solving the following least squares problem

$$O = R\Lambda, \tag{5.12.5}$$

we obtain a $m \times n$ matrix Λ. The entries of the computed matrix Λ are the values of the weighted connections between each column of R and each row of O, and they provide important information about the neural network. For the solution of the least squares problem (5.12.5) one may uses the SVD of R. The matrix R is usually a large dense matrix [86].

5.13 Conclusions

In this chapter we discussed a new parallel approach to computing the SVD. The first step of this approach involves the computation of the polar decomposition of a given matrix $A \in \mathbf{R}^{m\times n}$,

$$A = UH. \tag{5.13.1}$$

The polar decomposition (5.13.1) can be computed using Algorithm Parallel Polar, a parallel algorithm for computing the polar decomposition presented in Chapter 4. The second step involves the spectral decomposition

$$H = V\Sigma V^T \tag{5.13.2}$$

of the symmetric positive semidefinite factor H. The SVD of A,

$$A = (UV)\Sigma V^T, \tag{5.13.3}$$

is obtained at the third step by parallel multiplication of the orthogonal matrices U and V.

We investigated various parallel methods for the decomposition (5.13.2) on the KSR1. A parallel block Jacobi method, Method 2, was found to be the fastest method to compute (5.13.2) for large symmetric matrices. This method performs well for large dense symmetric matrices with distinct eigenvalues, but it must be avoided for symmetric matrices with multiple or close eigenvalues. But even in this case, using a variant of Method 2 (Variant 3) or the modified LAPACK symmetric eigensolver `SSYEV` for the second step, the SVD can be obtained in less time than using the modified LAPACK SVD solver `SGESVD`. The above described approach to computing the SVD is the fastest stable method we know for computing the SVD of large dense matrices on the KSR1. (The numerical stability of this approach has been investigated in [65].)

Applications of the SVD have also been discussed in this chapter. Some of them deal with large data matrices, and thus the need for a fast SVD solver is apparent.

Chapter 6

Concluding Remarks and Future Work

As we mentioned in the introduction of this thesis, this work was motivated by a class of two-sided Procrustes-type problems. The investigation of these problems revealed the necessary tools for their solution. Among them there are three of the most basic decompositions in numerical linear algebra, the SVD, the polar decomposition, and the spectral decomposition of a symmetric matrix.

The importance of the above-mentioned decompositions in both theoretical numerical linear algebra and applications, and also our interest in parallel computing motivated us to focus our attention on parallel algorithms for these decompositions. The availability of a 32-processors configuration KSR1 at the Centre for Novel Computing at the University of Manchester provided us with a powerful tool for our research.

We started our research on parallel algorithms for these decompositions with the polar decomposition. Having derived a parallel algorithm for this decomposition and implemented it with satisfactory results on the KSR1, we observed the relation between the polar decomposition and the SVD. This observation led our reserach to investigating parallel algorithms for the symmetric eigenvalue problem. The outcome of our research on the symmetric eigenvalue problem was the development of two parallel block Jacobi Algorithms, which turned out to be the fastest way known to us to compute the eigenvalues

and the eigenvectors of a large dense symmetric matrix on the KSR1. The combination of our parallel algorithm for computing the polar decomposition and these parallel block Jacobi methods, was found to be the fastest stable method we know for computing the SVD of large dense matrices on the KSR1.

We considered it necessary to include a separate chapter for the KSR1. As a pioneer user of the KSR1, we found it fascinating to explore the capabilities of this advanced parallel system. We do not claim that we developed the best software for the solution of our problems on the KSR1. Even in the late days of this project we discovered new ways to improve our existing codes. We have no doubt that users with a deeper knowledge of the KSR1 would be able to achieve better timing results for our parallel algorithms.

Applications have also been discussed in this thesis. The wide variety of these applications indicates the practical importance of our methods. It would be pleasure for us if one of our algorithms will be used for the solution of a problem in applications. For example, the main problem in statistical analysis of lexical data [9] is the compution of the SVD of very large dense matrices, where the new approach for computing the SVD described in Chapter 5 would may help.

Finally, we would like to refer to our future research plans. The satisfactory timing results for the two block Jacobi algorithms discussed in the previous chapter (Method 2 and Variant 3), stimulated us to to investigating similar block Jacobi methods for the SVD on the KSR1. It is also among our future plans to rewrite our codes using exclusively low-level parallel constructs (pthreads). This requires a deep knowledge of the hardware of the KSR1, a knowledge that we will try to obtain.

Appendix A

Listings of KSR Fortran Routines

A.1 The source code for Algorithm PJDM

The following KSR Fortran code refers to Algorithm PJDM discussed in page 151.

```
      SUBROUTINE PJDM (N, A, LDA, EIG, V, SWEEP, OFFA,
&                      NPROCS, TEAM)

*     .. Scalar Arguments ..
      INTEGER N, LDA, SWEEP, NPROCS, TEAM
      REAL    OFFA
*     ..
*     .. Array Arguments ..
      REAL A(LDA,*), V(LDA,*), EIG(*)
*     ..
*     Purpose
*     =======
*
*     PJDM computes all eigenvalues and eigenvectors of a
*     real symmetric matrix A using Algorithm PJDM.
*
*     Reference
*     =========
*     P. Papadimitriou.
*     Parallel Solution of SVD-Related Problems, with Applications.
*     Ph.D. Thesis, Dept. of Mathematics, University of Manchester, 1993.
*     (Chapter 5, Section 9).
*
*
*     Arguments
*     =========
*
*     N   (input) INTEGER
*     The number of rows and columns of the matrix A.
*
*     A   (input) REAL array, dimension (LDA,N).
*     On entry, the symmetric matrix A.
*
*     On exit, the diagonal of A contains the eigenvalues
```

```
*     of the original matrix A.
*
*     LDA  (input) INTEGER
*     The leading dimension of the array A. LDA >=max(1,N).
*
*     EIG  (output) REAL array, dimension (N).
*     The eigenvalues in descending order.
*
*     V  (output) REAL array, dimension (LDA,N).
*     On exit, the columns of V are the  eigenvectors
*     of the original matrix A.
*
*     SWEEP (output) INTEGER
*     On exit, the number of required sweeps.
*
*     OFFA  (output) REAL
*     On exit, the Frobenius norm of the off-diagonal
*     entries of the updated matrix A.
*
*     NPROCS (input) INTEGER
*     The number of processors.
*
*     TEAM (input) INTEGER
*     The identification number of the team of the p_threads.
*
*     ========================================================
*
*     .. Parameters ..
*     Parameter NX must be set equal to the dimension of
*     the original matrix.
      INTEGER N2, PS, NX
      PARAMETER (NX = 64, PS = 1, N2 = NX/2)
      REAL ONE, ZERO
      PARAMETER ( ONE = 1., ZERO = 0.)
*     ..
*     .. Local Scalars ..
      LOGICAL MODI
      INTEGER SET, MYNUM
      REAL FG, RM, TOL
*     .. Local Arrays ..
      INTEGER INDEX(NX), TOP(N2), BOT(N2)
      REAL JC(NX,NX), B1(NX,NX), CT(NX,NX)
*
*     .. Arrays for the Davies-Modi Scheme ..
*
      REAL A1(NX,NX), V1(NX,NX), VA1(NX,NX), B(NX,NX),
     &     W(NX,NX),  X(NX,NX),  U(NX,NX),  X2(NX,NX),
     &     VC(NX)
*
*     .. External Subroutines ..
*
      EXTERNAL   SGEMM,  SAMEH, LOCAL, INITLS, SYM2,
```

```
     &           SORT01, ADDMAT, ADXMAT, SORT02
*
*     .. External Functions ..
*
      REAL OFF, SLAMCH
      LOGICAL MODDAV
*     EXTERNAL OFF, MODDAV, SLAMCH
*     ..
*     .. Intrinsic Functions ..
      INTRINSIC FLOAT
*     ..
*     .. Executable Statements ..
*
*     .. Compute the Frobenius norm of A ..
*
      CALL EVNORM(A,N,RM,TEAM)
*
*     .. Compute the tolerance ..
*
      TOL = FLOAT(N)*RM*SLAMCH('E')
*
*      .. Set U and V to the N x N identity matrix ..
*
      CALL INITLS(U,N,TEAM)
      CALL INITLS(V,N,TEAM)
*
*     .. Compute the Frobenius norm of the off-diagonal elements
*        of A ..
*
      OFFA = OFF(A,N,TEAM)
*
*     .. Initialize the counter SWEEP ..
*
      SWEEP = 0
*
*     .. Initialize the logical variable MODI ..
*
      MODI = .FALSE.
*
*     .. Keep iterating until the Frobenius norm of the
*        off-diagonal entries of A is less than TOL or
*        the condition for applying the Davies-Modi Scheme
*        is satisfied ..
*
      DO 100 WHILE ((OFFA .GT. TOL) .AND. (.NOT. MODI))
*
         SWEEP = SWEEP + 1
*
         DO 30 SET = 1,N-1
*
*      .. Set JC to the N x N identity matrix ..
*
```

```
            CALL INITLS(JC,N,TEAM)
*
*      .. Determine the SETth Jacobi Set ..
*
            CALL SAMEH(SET,N,N2,TOP,BOT)
*
*      .. Solve the subproblems in parallel and
*         build the complete Jacobi rotation JC ..
*
C*KSR* PARALLEL REGION(TEAMID = TEAM, PRIVATE = (K, MYNUM))
         MYNUM = IPR_MID()
         DO  K=1,N2
           IF (MOD(K,NPROCS) .EQ. MYNUM)
     &           CALL LOCAL(K,A,JC,N,TOP,BOT)
         END DO
C*KSR* END PARALLEL REGION
*
*     .. Update the matrices A and V ..
*
      CALL SGEMM('T','N', N, N, N, 1.,JC, N, A, N,0.,B1,N,
     &            CT,NPROCS,TEAM,PS)
      CALL SGEMM('N','N',N,N,N,1.,B1,N,JC,N,0.0,A,N,
     &            CT,NPROCS,TEAM,PS)
      CALL SGEMM('N','N',N,N,N,1.,V,N,JC,N,0.,V,N,
     &            CT,NPROCS,TEAM,PS)
*
 30   CONTINUE
*
*     .. Compute the Frobenius norm of the off-diagonal entries ..
*
      OFFA  = OFF(A,N,TEAM)
*
*     .. Check if the condition for applying the Davies-Modi
*        method is satisfied ..
*
      IF (SWEEP .GE. 4) MODI = MODDAV(A,VC,N,TEAM)
*
 100  CONTINUE
*
*     .. Apply the Davies-Modi Method ..
*
      A1 = A
*
C*KSR* TILE(I, TEAMID = TEAM)
      DO I=1,N
         A1(I,I) = ZERO
      END DO
C*KSR* END TILE
*
C*KSR* TILE ( J, I, TEAMID = TEAM )
      DO J=1,N
         DO I=1,N
```

```
            IF (I .EQ. J) THEN
               V1(I,J) = ZERO
            ELSE
               V1(I,J) = A(I,J)/(A(I,I)-A(J,J))
            END IF
         END DO
      END DO
C*KSR* END TILE
*
      CALL SGEMM('N','N',N,N,N,1.,V1,N,A1,N,0.,VA1,N,
     &            CT,NPROCS, TEAM,PS)
*
      CALL ADXMAT(VA1,B,N,TEAM)
*
C*KSR* TILE ( J, I, TEAMID = TEAM )
      DO J=1,N
         DO I=1,N
            IF (I .EQ. J) THEN
               W(I,J) = ZERO
            ELSE
               W(I,J) = B(I,J)/(A(I,I)-A(J,J))
            END IF
         END DO
      END DO
C*KSR* END TILE
*
      CALL ADDMAT(V1,W,X,N,TEAM)
      CALL ADDMAT(U,X,U,N,TEAM)
*
      CALL SGEMM('N','N',N,N,N,0.5,X,N,X,N,0.,X2,N,CT,NPROCS,
     &            TEAM, PS)
*
      CALL ADDMAT(U,X2,U,N,TEAM)
*
      FG = 1./3.
*
      CALL SGEMM('N','N',N,N,N,FG,X2,N,X,N,0.,X,N,CT,NPROCS,
     &            TEAM, PS)
*
      CALL ADDMAT(U,X,U,N,TEAM)
*
      CALL SGEMM('N','N',N,N,N,1.,U,N,A,N,0.,X2,N,CT,NPROCS,
     &            TEAM,PS)
      CALL SGEMM('N','T',N,N,N,1.,X2,N,U,N,0.,A,N,CT,NPROCS,
     &            TEAM,PS)
      CALL SGEMM('N','T',N,N,N,1.,U,N,V,N,0.,V,N,CT,NPROCS,
     &            TEAM,PS)
*
*     .. Form the vector of the eigenvalues ..
*
      DO I=1,N
         EIG(I) = A(I,I)
```

```
      END DO
*
*     .. Sort the vector of the eigenvalues in descending order ..
*
      CALL SORT01(EIG,INDEX,N,TEAM)
*
*     .. Permute the columns of V in order to agree with EIG ..
*
C*KSR* TILE ( J,I, TEAMID = TEAM )
       DO  J=1,N
         DO  I=1,N
           X2(I,J) = V(I,J)
       END DO
       END DO
C*KSR* END TILE
*
C*KSR* TILE ( J,I, TEAMID = TEAM )
       DO  J=1,N
         DO  I=1,N
           V(I,J) = X2(I,INDEX(J))
         END DO
       END DO
C*KSR* END TILE
*
*     .. Compute the Frobenius norm of the off-diagonal entries
*        of the nearly diagonal matrix A.
*
      OFFA  = OFF(A,N,TEAM)
*
*
*     End of PJDM
*
      END


      SUBROUTINE LOCAL(K, A, JC, N, TOP, BOT)
*
*     .. Scalar Arguments ..
*
      INTEGER K, N, N2
*     ..
*     .. Array Arguments ..
      INTEGER TOP(*), BOT(*)
      REAL A(N,*), JC(N,*)
*
*     Purpose
*     =======
*
*     LOCAL solves the (p,q) subproblem on the K-th processor
*     and embeds a 2 x 2 orthogonal matrix into a complete
*     Jacobi rotation. (p,q) is K-th index pair of the current
*     Jacobi set.
```

```
*
*     .. Local Scalars ..
*
      INTEGER P, Q
*
*     .. Local Arrays ..
      REAL  BLOCK(2,2)
*     ..
*     .. External Subroutines ..
      EXTERNAL SYM2
*     ..
*     .. Intrinsic Functions ..
*
      INTRINSIC MIN, MAX
*
*     .. Executable Statements ..
*
      P = MIN(TOP(K),BOT(K))
      Q = MAX(TOP(K),BOT(K))
*
      BLOCK(1,1) = A(P,P)
      BLOCK(2,2) = A(Q,Q)
      BLOCK(1,2) = A(P,Q)
      BLOCK(2,1) = A(Q,P)
*
      CALL SYM2(BLOCK)
*
      JC(P,P) = BLOCK(1,1)
      JC(P,Q) = BLOCK(1,2)
      JC(Q,P) = BLOCK(2,1)
      JC(Q,Q) = BLOCK(2,2)
*
*     End of LOCAL
*
      END


      SUBROUTINE SAMEH(K,N,N2,TOP,BOT)
*
*     .. Scalar Arguments ..
      INTEGER K, N, N2
*     .. Array Arguments ..
      INTEGER TOP(*), BOT(*)
*
*     Purpose
*     =======
*
*     SAMEH computes the K-th Jacobi set using
*     Sameh's scheme.
*
*     .. Local Scalars ..
      INTEGER L, I, J
```

```
*
*     .. Executable Statements ..
*
      L = 1
*
      IF (K .LE. N2-1) THEN
       DO 10 I=N2-K+1,N-K
        IF ((N2-K+1 .LE. I) .AND. (I .LE. N-2*K)) THEN
         J = N - 2*K +1 -I
        ELSE IF ((N-2*K .LT. I) .AND. (I . LE. N-K-1)) THEN
         J = 4*N2-2*K-I
        ELSE
         J = N
        END IF
        TOP(L) = J
        BOT(L) = I
        L = L + 1
 10    CONTINUE
*
      ELSE
*
       DO 20 I=4*N2-N-K,3*N2-K-1
        IF ( I .LT. N-K+1) THEN
         J = N
        ELSE IF ((N-K+1 .LE. I) .AND. (I .LE. 4*N2-2*K-1)) THEN
         J = (4*N2 -2*K) - I
        ELSE
         J = (6*N2 - 2*K -1) - I
        END IF
        TOP(L) = J
        BOT(L) = I
        L = L + 1
 20    CONTINUE
*
      END IF
*
*     End of SAMEH
*
      END
```

A.2 The source code for Algorithm Method 1

The following KSR Fortran code refers to Algorithm Method 1 discussed in page 144.

```
      SUBROUTINE METHOD1 (N, A, LDA, THETA, EIG, V, SWEEP, OFFA,
     &                    NPROCS, TEAM)
*
*     .. Scalar Arguments ..
      INTEGER N, LDA, SWEEP, NPROCS, TEAM
      REAL    THETA, OFFA
*     ..
*     .. Array Arguments ..
      REAL A(LDA,*), V(LDA,*), EIG(*)
*     ..
*     Purpose
*     =======
*
*     METHOD1 computes all eigenvalues and eigenvectors of a
*     real symmetric matrix A using Algorithm Method 1.
*
*     Reference
*     =========
*     P. Papadimitriou.
*     Parallel Solution of SVD-Related Problems, with Applications.
*     Ph.D. Thesis, Dept. of Mathematics, University of Manchester, 1993.
*     (Chapter 5, Section 8).
*
*
*     Arguments
*     =========
*
*     N   (input) INTEGER
*     The number of rows and columns of the matrix A.
*
*     A   (input) REAL array, dimension (LDA,N).
*     On entry, the symmetric matrix A.
*
*     On exit, the diagonal of A contains the eigenvalues
*     of the original matrix A.
*
*     LDA  (input) INTEGER
*     The leading dimension of the array A. LDA >=max(1,N).
*
*     THETA (input) REAL
*     The parameter theta for the Partial Jacobi Method. 0 < THETA < 1.
*
*     EIG  (output) REAL array, dimension (N).
*     The eigenvalues in descending order.
*
*     V  (output) REAL array, dimension (LDA,N).
*     On exit, the columns of V are the  eigenvectors
*     of the original matrix A.
```

```
*
*     SWEEP (output) INTEGER
*     On exit, the number of required sweeps.
*
*     OFFA  (output) REAL
*     On exit, the Frobenius norm of the off-diagonal
*     entries of the updated matrix A.
*
*     NPROCS (input) INTEGER
*     The number of processors.
*
*     TEAM (input) INTEGER
*     The identification number of the team of the p_threads.
*
*     ========================================================
*
*     .. Parameters ..
*     Parameter NX must be set equal to the dimension of
*     the original matrix.
      INTEGER NX, N2, PS
      PARAMETER (NX = 64, N2 = 16, PS = 1)
*     ..
*     .. Local Scalars ..
      INTEGER SET, MYNUM, NB, NB2
      REAL EPS, RM, TOL, TH
*     .. Local Arrays ..
      INTEGER TOP(N2), BOT(N2), INDEX(NX),
     &        NEWTOP(N2), NEWBOT(N2)
      REAL    P(NX,NX), JC(NX,NX), B1(NX,NX), CT(NX,NX)
*     .. External Subroutines ..
      EXTERNAL EVNORM, INITLS, OFFBLC, LOCAL, SGEMM,
     &         MUSIC, LOCALX, SORT01
*     .. External Functions ..
      REAL SLAMCH, OFF
      EXTERNAL SLAMCH, OFF
*     .. Intrinsic Functions ..
      INTRINSIC FLOAT
*     ..
*     .. Executable Statements ..
*
*     .. Compute the tolerance and the threshold ..
*
      CALL EVNORM(A,N,RM,TEAM)
      EPS = SLAMCH('E')
      TOL = FLOAT(N)*RM*EPS
      TH  = TOL/32.
*
*
*     .. Compute the size of the subproblems ..
      NB   =  N/NPROCS
*     .. Compute the block size ..
      NB2  =  NB/2
```

```
*
*     .. Set the matrix V to the N x N identity matrix ..
*
      CALL INITLS(V,N,TEAM)
*     ..
*     .. Initialize the vectors TOP and BOT ..
*
      DO I=1,N2
         TOP(I) = 2*I-1
         BOT(I) = 2*I
      END DO
*
*     .. Compute the Frobenius norm of the off-diagonal
*        blocks of A ..
*
      CALL OFFBLC(A,N,NB,NPROCS,OFFA,TEAM)
*
*     .. Initialize the counter SWEEP ..
*
      SWEEP = 0
*
*     .. Keep iterating until OFF(A) is less than TOL ..
*
      DO 100 WHILE (OFFA .GT. TOL)
*
         WRITE(*,1000) SWEEP, OFFA
 1000    FORMAT(1X,I5,E12.4)
         SWEEP = SWEEP + 1
*
         DO 30 SET = 1, 2*NPROCS - 1
*
*     .. Initialize the Complete Jacobi Rotation JC ..
            CALL INITLS(JC,N,TEAM)
*
*     .. Solve the subproblems in parallel using the
*           Partial Jacobi Method and build the complete
*           Jacobi rotation JC ..
C*KSR* PARALLEL REGION(TEAMID = TEAM, PRIVATE = (K, MYNUM))
         MYNUM = IPR_MID()
         DO  K=1,NPROCS
          IF (MOD(K,NPROCS) .EQ. MYNUM)
     &       CALL LOCAL(K,A,JC,N,N2,NB,NB2,TOP,BOT,TH,THETA)
         END DO
*C*KSR* END PARALLEL REGION
*     ..
*     ..  Update the matrices A and V ..
*
      CALL SGEMM('T','N',N,N,N,1.,JC,N,A,N,0.0,B1,N,
     &            CT,NPROCS,TEAM,PS)
      CALL SGEMM('N','N',N,N,N,1.,B1,N,JC,N,0.0,A,N,
     &            CT,NPROCS,TEAM,PS)
      CALL SGEMM('N','N',N,N,N,1.,V,N,JC,N,0.,V,N,
```

```
     &            CT,NPROCS,TEAM,PS)
*
*     .. Compute the next Jacobi set ..
      CALL MUSIC(TOP,BOT,N2,NEWTOP,NEWBOT,TEAM)
*
 30   CONTINUE
*
*     .. Compute OFF(A) ..
      CALL OFFBLC(A,N,NB,NPROCS,OFFA,TEAM)
*
 100  CONTINUE
*
*     .. End of the first part of Algorithm Method 1 ..
*
*     .. The matrix A is nearly block diagonal at this point ..
*
*     .. Initialize the vectors TOP and BOT ..
*
      DO I=1,N2
         TOP(I) = 2*I-1
         BOT(I) = 2*I
      END DO
*
*     .. Initialize the complete Jacobi Rotation JC ..
*
      CALL INITLS(JC,N,TEAM)
*
*
*     .. Solve the subroblems on the block diagonal using
*        SSYEV and build the complete Jacobi rotation JC ..
*
C*KSR* PARALLEL REGION(TEAMID = TEAM, PRIVATE = (K, MYNUM))
         MYNUM = IPR_MID()
         DO  K=1,NPROCS
          IF (MOD(K,NPROCS) .EQ. MYNUM)
     &       CALL LOCAL(K,A,JC,N,N2,NB,NB2,TOP,BOT)
         END DO
C*KSR* END PARALLEL REGION
*
*     .. Update A nad V ..
*
      CALL SGEMM('T','N',N,N,N,1.,JC,N,A,N,0.0,B1,N,
     &            CT,NPROCS,TEAM,PS)
      CALL SGEMM('N','N',N,N,N,1.,B1,N,JC,N,0.0,A,N,
     &            CT,NPROCS,TEAM,PS)
      CALL SGEMM('N','N',N,N,N,1.,V,N,JC,N,0.,V,N,
     &            CT,NPROCS,TEAM,PS)
*
*     .. Form the vector of the eigenvalues ..
*
      DO I=1,N
         EIG(I) = A(I,I)
```

```
      END DO
*
*    .. Sort the vector of the eigenvalues in descending order ..
*
      CALL SORT01(EIG,INDEX,N,TEAM)
*
*     .. Permute the columns of V in order to agree with EIG ..
*
C*KSR* TILE ( J,I, TEAMID = TEAM )
       DO  J=1,N
         DO  I=1,N
           P(I,J) = V(I,J)
         END DO
       END DO
C*KSR* END TILE
*
C*KSR* TILE ( J,I, TEAMID = TEAM )
       DO  J=1,N
         DO  I=1,N
           V(I,J) = P(I,INDEX(J))
         END DO
       END DO
C*KSR* END TILE
*
*     .. Compute the Frobenius norm of the off-daigonal
*        entries of A ..
*
      OFFA = OFF(A,N,TEAM)
*
*     End of METHOD1
*
      END


      SUBROUTINE LOCAL(K,A,JC,N,N2,NB,NB2,TOP,BOT,TH,THETA)
*
*     .. Scalar Arguments ..
      INTEGER K, N, N2, NB, NB2
      REAL TH, THETA
*     .. Array Arguments ..
      INTEGER TOP(N2), BOT(N2)
      REAL A(N,N), JC(N,N)
*
*     Purpose
*     =======
*
*     LOCAL solves the K-th subproblem using the partial
*     Jacobi method and builds the complete Jacobi rotation JC
*     which corresponds to the current Jacobi step.
*
*     .. Parameters ..
*     Parameter NX must always be set to the size of the
```

```
*     the subproblem.
      INTEGER NX
      PARAMETER (NX = 4)
      REAL ZERO
      PARAMETER(ZERO = 0.)
*
*     .. Local Scalars ..
      INTEGER I, J, P, Q, P1, Q1, P2, Q2
      REAL    DF
*     .. Array Scalars ..
      REAL    BLOCK(NX,NX), BL(NX,NX)
*     ..
*     .. External Subroutines ..
      EXTERNAL PARJAC
*
*     .. Intrinsic Functions ..
      INTRINSIC MIN, MAX
*
*     .. Executable Statements ..
*
      P = (MIN(TOP(K),BOT(K))-1)*NB2
      Q = (MAX(TOP(K),BOT(K))-1)*NB2
*
      P1 = P + 1
      Q1 = Q + 1
      P2 = P + NB2
      Q2 = Q + NB2
*
      BLOCK(1:NB2,1:NB2)             = A(P1:P2,P1:P2)
      BLOCK(1:NB2,NB2+1:2*NB2)       = A(P1:P2,Q1:Q2)
      BLOCK(NB2+1:2*NB2,1:NB2)       = A(Q1:Q2,P1:P2)
      BLOCK(NB2+1:2*NB2,NB2+1:2*NB2) = A(Q1:Q2,Q1:Q2)
*
      DF = ZERO
*
      DO I=NB2+1,NB
         DO J=1,NB2
            DF = DF + BLOCK(I,J)**2
         END DO
      END DO
*
      IF (DF .LE. (TH**2)) RETURN
*
      CALL PARJAC(BLOCK,NB,BL,THETA)
*
      JC(P1:P2,P1:P2) =  BL(1:NB2,1:NB2)
      JC(P1:P2,Q1:Q2) =  BL(1:NB2,NB2+1:2*NB2)
      JC(Q1:Q2,P1:P2) =  BL(NB2+1:2*NB2,1:NB2)
      JC(Q1:Q2,Q1:Q2) =  BL(NB2+1:2*NB2,NB2+1:2*NB2)
*
*     End of LOCAL
*
```

```
      END

      REAL FUNCTION DIAG (A, N, K, NB, TEAM)
*
*     .. Scalar Arguments ..
      INTEGER K, N, TEAM
*     .. Array Arguments ..
      REAL A(N,N)
*
*     Purpose
*     =======
*
*     DIAG computes the sum of squares of the off-diagonal
*     entries of the K-th NB x NB block on the block diagonal
*     of A.
*
*     .. Parameters ..
      REAL ZERO
      PARAMETER(ZERO = 0.)
*     .. Local Scalars ..
      INTEGER I, J
      REAL SUM
*     ..
*     .. Executable Statements ..
*
      SUM = ZERO
*
C*KSR* TILE ( I,PRIVATE=( J ),REDUCTION=( SUM ), TEAMID = TEAM)
       DO 2 I=(K-1)*NB+2,K*NB
         DO 2 J=(K-1)*NB+1,I-1
           SUM = SUM + A(I,J) ** 2
    2      CONTINUE
C*KSR* END TILE
*
      SUM = 2 * SUM
*
      DIAG = SUM
*
*     End of DIAG
*
      END


      SUBROUTINE PARJAC(A,NB,V,THETA)
*
*     .. Scalar Arguments ..
      INTEGER NB
      REAL THETA
*     .. Array Arguments ..
      REAL A(NB,NB), V(NB,NB)
*
*     Purpose
```

```
*     =======
*
*     PARJAC solves a subproblem using the
*     partial Jacobi method ..
*
*     .. Parameters ..
*     Parameter NX must always be set to size
*     of the subproblem ..
      INTEGER NX, NS
      PARAMETER (NX = 4, NS = NX/2)
      REAL ZERO
      PARAMETER (ZERO = 0.)
*
*     .. Local Scalars ..
      INTEGER  I, J, P, Q
      LOGICAL OKEY
      REAL TOL, SUM
*     .. Local Arrays ..
      REAL AR(NS,NX), SH(NS,NS), AC(NX,NS), VC(NX,NS)
*
*     .. External Subroutines ..
      EXTERNAL SGEMM
*
*     .. Executable Statements ..
*
      DO 10 I=1,NB
         DO 10 J=1,NB
            IF (I .EQ. J) THEN
               V(I,J) = 1.
            ELSE
               V(I,J) = 0.
            END IF
 10   CONTINUE
*
      SUM = ZERO
*
      DO 20 I=NS+1,NB
         DO 20 J=1,NS
            SUM = SUM + A(I,J)**2
 20   CONTINUE
*
      TOL = SUM*(THETA**2)
*
      SWEEP = 0
*
      DO 100 WHILE( SUM .GE. TOL)
*
         SWEEP = SWEEP + 1
*
         DO 30 P = 1,NB-1
            DO 30 Q = (P+1),NB
*
```

```
            SH(1,1) = A(P,P)
            SH(1,2) = A(P,Q)
            SH(2,1) = A(Q,P)
            SH(2,2) = A(Q,Q)
*
            CALL SYM2(SH)
*
            AC(:,1) = A(P,:)
            AC(:,2) = A(Q,:)
            CALL SGEMM('N','N',4,2,2,1.,AC,4,
     &                  SH,2,0.,AC,4)
            A(:,P) = AC(:,1)
            A(:,Q) = AC(:,2)
*
            AR(1,:) = A(P,:)
            AR(2,:) = A(Q,:)
            CALL SGEMM('T','N',2,4,2,1.,SH,2,
     &                  AR,2,0.,AR,2)
            A(P,:) = AR(1,:)
            A(Q,:) = AR(2,:)
*
            VC(:,1) = V(:,P)
            VC(:,2) = V(:,Q)
*
            CALL SGEMM('N','N',4,2,2,1.,VC,4,
     &                  SH,2,0.,VC,4)
            V(:,P) = VC(:,1)
            V(:,Q) = VC(:,2)
*
 30      CONTINUE
*
            SUM = ZERO

            DO 40 I=NS+1,NB
              DO 40 J=1,NS
                 SUM = SUM + A(I,J)**2
 40         CONTINUE
*
 100        CONTINUE
*
*     End of PARJAC
*
      END


      SUBROUTINE LOCALX(K,A,JC,N,N2,NB,NB2,TOP,BOT)
*
*     .. Scalar Arguments ..
      INTEGER K, N, N2, NB, NB2
*     .. Array Arguments ..
      REAL A(N,N), JC(N,N)
      INTEGER TOP(N2), BOT(N2)
```

```
*
*     Purpose
*     =======
*
*     LOCALX computes the Schur decomposition of
*     the K-th block on the block diagonal of A,
*     and builds the complete Jacobi rotation JC.
*
*     .. Parametes ..
*     Parameter NX must be set equal to the
*     size of the subproblem.
      INTEGER NX, LWORK
      PARAMETER(NX = 4, LWORK = 3*NX - 1)
*     .. Scalar Arguments ..
      INTEGER P, Q, P1, Q1, P2, Q2, INFO
*     .. Array Arguments ..
      REAL W(NX), WORK(LWORK), BLOCK(NX,NX)
*     ..
*     .. External Subroutines ..
      EXTERNAL SSYEV
*     .. Executable Statements ..
*
      P = (MIN(TOP(K),BOT(K))-1)*NB2
      Q = (MAX(TOP(K),BOT(K))-1)*NB2
*
      P1 = P + 1
      Q1 = Q + 1
      P2 = P + NB2
      Q2 = Q + NB2
*
      BLOCK(1:NB2,1:NB2)             = A(P1:P2,P1:P2)
      BLOCK(1:NB2,NB2+1:2*NB2)       = A(P1:P2,Q1:Q2)
      BLOCK(NB2+1:2*NB2,1:NB2)       = A(Q1:Q2,P1:P2)
      BLOCK(NB2+1:2*NB2,NB2+1:2*NB2) = A(Q1:Q2,Q1:Q2)
*
      CALL SSYEV('V','U',NB,BLOCK,NB,W,WORK,LWORK,INFO)
*
      JC(P1:P2,P1:P2) =  BLOCK(1:NB2,1:NB2)
      JC(P1:P2,Q1:Q2) =  BLOCK(1:NB2,NB2+1:2*NB2)
      JC(Q1:Q2,P1:P2) =  BLOCK(NB2+1:2*NB2,1:NB2)
      JC(Q1:Q2,Q1:Q2) =  BLOCK(NB2+1:2*NB2,NB2+1:2*NB2)
*
*     End of LOCALX
*
      END


      SUBROUTINE OFFBLC(A,N,NB,NPROCS,OFFA,TEAM)
*
*     .. Scalar Arguments ..
*
      INTEGER N, NB, TEAM, NPROCS
```

```
      REAL OFFA
*     .. Array Arguments ..
      REAL A(N,*)
*
*     Purpose
*     =======
*
*     OFFBLC computes the Frobenius norm of
*     the off-diagonal blocks.
*
*     .. Parameters ..
      REAL ZERO
      PARAMETER (ZERO = 0.)
*
*     .. Local Scalars ..
*
      INTEGER I
      REAL SUM
*     ..
*     .. External Functions ..
      REAL DIAG, OFF
      EXTERNAL OFF
*     ..
*     .. Executable Statements ..
*
      OFFA = OFF(A,N,TEAM)**2
      SUM = ZERO
*
      DO I=1,NPROCS
         SUM = SUM + DIAG(A,N,I,NB,TEAM)
      END DO
*
      OFFA = SQRT(OFFA-SUM)
*
*     End of OFFBLC
*
      END
```

A.3 The Source Code for Algorithm Method 2

The following KSR Fortran code refers to Algorithm Method 2 discussed in page 154.

```
      SUBROUTINE METHOD2 (N , A, LDA, EIG, V, SWEEP,
     &                    OFFA, NPROCS, TEAM)
*     .. Scalar Arguments ..
      INTEGER N, LDA, SWEEP, NPROCS, TEAM
      REAL    OFFA
*     ..
*     .. Array Arguments ..
      REAL A(LDA,*), V(LDA,*), EIG(*)
*     ..
```

```
*     Purpose
*     =======
*
*     METHOD2 computes all eigenvalues and eigenvectors of a
*     real symmetric matrix A using Algorithm Method 2.
*
*     Reference
*     =========
*     P. Papadimitriou.
*     Parallel Solution of SVD-Related Problems, with Applications.
*     Ph.D. Thesis, Dept. of Mathematics, University of Manchester, 1993.
*     (Chapter 5, Section 9).
*
*
*     Arguments
*     =========
*
*     N   (input) INTEGER
*     The number of rows and columns of the matrix A.
*
*     A   (input) REAL array, dimension (LDA,N).
*     On entry, the symmetric matrix A.
*
*     On exit, the diagonal of A contains the eigenvalues
*     of the original matrix A.
*
*     LDA  (input) INTEGER
*     The leading dimension of the array A. LDA >=max(1,N).
*
*     EIG  (output) REAL array, dimension (N).
*     The eigenvalues in descending order.
*
*     V  (output) REAL array, dimension (LDA,N).
*     On exit, the columns of V are the  eigenvectors
*     of the original matrix A.
*
*     SWEEP (output) INTEGER
*     On exit, the number of required sweeps.
*
*     OFFA  (output) REAL
*     On exit, the Frobenius norm of the off-diagonal
*     entries of the updated matrix A.
*
*     NPROCS (input) INTEGER
*     The number of processors.
*
*     TEAM (input) INTEGER
*     The identification number of the team of the p_threads.
*
*     =========================================================
*
*     .. Parameters ..
```

```
*
*     Parameter NX must be set equal to the dimension
*     of the original matrix.
      INTEGER NX, NP, NB, NB2, PS
      PARAMETER (NX = 64, NP = 16, NB = NX/NP,
     &           NB2 = NB/2, PS = 1)
      REAL    ONE, ZERO
      PARAMETER ( ONE = 1., ZERO = 0.)
*     ..
*     .. Local Scalars ..
      INTEGER SET, MYNUM
      LOGICAL MODI
      REAL FC, TOL, RM, THRESH, OMEGA
*     ..
*     .. Local Arrays ..
*     ..
      INTEGER TOP(NP), BOT(NP), INDEX(NX),
     &        P(NP), Q(NP)
*     ..
      LOGICAL THR(NP)
*     ..
      REAL CT(NX,NX), PP(NX,NX), BL(NP,NB,NB), VC(NX)
*     ..
*     .. Arrays for Davies-Modi Scheme ..
*     ..
      REAL A1(NX,NX), V1(NX,NX), VA1(NX,NX), B(NX,NX),
     &     W(NX,NX),  X(NX,NX),  U(NX,NX),  X2(NX,NX)
*     ..
*     ..  External Subroutines ..
      EXTERNAL EVNORM, INITLS, UPDATE_COLUMNS,
     &         UPDATE_ROWS, MUSIC, SGEMM, ADXMAT,
     &         SORT01
*
*     .. External Functions ..
      REAL OFF, SLAMCH
      LOGICAL MODDAV
      EXTERNAL OFF, MODDAV, SLAMCH
*     ..
*     .. Intrinsic Functions ..
      INTRINSIC FLOAT
*     ..
*
*     .. Executable Statements ..
*
*     .. Compute the Frobenius norm of A ..
*
      CALL EVNORM(A,N,RM,TEAM)
*
*     .. Compute the tolerance ..
*
      TOL = FLOAT(N)*RM*SLAMCH('E')
*     .. Set U and V to  the N x N identity matrix ..
```

```
*
       CALL INITLS(V,N,TEAM)
       CALL INITLS(U,N,TEAM)
*
*     .. Initialize the vectors TOP and BOT ..
*
C*KSR* TILE ( I, TEAMID = TEAM )
       DO 4 I=1,NPROCS
         TOP(I) = I * 2 - 1
         BOT(I) = I * 2
    4  CONTINUE
C*KSR* END TILE
*
*      .. Compute the Frobenius norm of the off-diagonal
*         entries of A ..
*
       OFFA = OFF(A,N,TEAM)
*
*      .. Compute PR and AR for the threshold ..
       PR   = FLOAT(N*(N-1)/2)
       AR   = NB*(NB-1)/2
*
*      .. Initialize the counter SWEEP ..
       SWEEP = 0
*      .. Initialize the logical variable MODI ..
       MODI = .FALSE.
*
*      .. Keep iterating until the Frobenius norm of the
*         off-diagonal entries of A is less than TOL or
*         the condition for applying the Davies-Modi scheme
*         is satisfied ..
*
       DO 100 WHILE ((OFFA .GT. TOL) .AND. (.NOT. MODI))
*
         WRITE(*,1020) SWEEP, OFFA
 1020    FORMAT(1X,I5,E12.4)
*
       SWEEP = SWEEP + 1
*
*      .. Compute the threshold ..
*
       OMEGA  = (OFFA**2)/2
       THRESH = AR*(OMEGA/PR)
*
       DO 30 SET=1, 2*NPROCS-1
*      .. Initialize the logical vector THR ..
       DO I=1,NPROCS
         THR(I) = .TRUE.
       END DO
*
*
*      .. Update the block columns of A and V ..
```

```
*
C*KSR* PARALLEL REGION( TEAMID = TEAM, PRIVATE = (K, MYNUM ))
       MYNUM = IPR_MID()
         DO   K=1,NPROCS
           IF (MOD(K,NPROCS) .EQ. MYNUM)
     &      CALL UPDATE_COLUMNS(K,A,V,N,NB,NB2,NPROCS,TOP,BOT,
     &      THRESH,SWEEP,P, Q, BL, THR)
         END DO
C*KSR*  END PARALLEL REGION
*
*      .. Update the block rows of A ..
*
C*KSR* PARALLEL REGION( TEAMID = TEAM, PRIVATE = (K, MYNUM ))
       MYNUM = IPR_MID()
         DO   K=1,NPROCS
           IF ((MOD(K,NPROCS) .EQ. MYNUM) .AND. THR(K))
     &       CALL UPDATE_ROWS (K,A,N,NB,NB2,NPROCS,TOP,BOT,
     &                 P,Q,BL,THR)
         END DO
C*KSR*  END PARALLEL REGION
*
*     .. Compute the next Jacobi set using the
*        Tournament scheme ..
*
      CALL MUSIC (TOP,BOT,NPROCS,TEAM)
*
   30 CONTINUE
*
*     .. Compute the Frobenius norm of the off-diagonal entries ..
*
      OFFA = OFF(A,N,TEAM)
*
*     .. Check whether the condition for applying the Davies-Modi
*        method is satisfied ..
*
      IF (SWEEP .GE. 10) MODI = MODDAV(A,VC,N,TEAM)
*
 100  CONTINUE
*
*     .. Apply the Davies-Modi Method ..
*
      A1 = A
*
C*KSR* TILE(I, TEAMID = TEAM)
      DO I=1,N
         A1(I,I) = ZERO
      END DO
C*KSR* END TILE
*
C*KSR* TILE ( J, I, TEAMID = TEAM )
      DO J=1,N
         DO I=1,N
```

```
            IF (I .EQ. J) THEN
               V1(I,J) = ZERO
            ELSE
               V1(I,J) = A(I,J)/(A(I,I)-A(J,J))
            END IF
         END DO
      END DO
C*KSR* END TILE
*
      CALL SGEMM('N','N',N,N,N,1.,V1,N,A1,N,0.,VA1,N,CT,NPROCS,
     &            TEAM,PS)
*
      CALL ADXMAT(VA1,B,N,TEAM)
*
C*KSR* TILE ( J, I, TEAMID = TEAM )
      DO J=1,N
         DO I=1,N
            IF (I .EQ. J) THEN
               W(I,J) = ZERO
            ELSE
               W(I,J) = B(I,J)/(A(I,I)-A(J,J))
            END IF
         END DO
      END DO
C*KSR* END TILE
*
      CALL ADDMAT(V1,W,X,N,TEAM)
      CALL ADDMAT(U,X,U,N,TEAM)
*
      CALL SGEMM('N','N',N,N,N,0.5,X,N,X,N,0.,X2,N,CT,NPROCS,
     &            TEAM, PS)
*
      CALL ADDMAT(U,X2,U,N,TEAM)
*
      FG = 1./3.
*
      CALL SGEMM('N','N',N,N,N,FG,X2,N,X,N,0.,X,N,CT,NPROCS,
     &            TEAM, PS)
*
      CALL ADDMAT(U,X,U,N,TEAM)
*
      CALL SGEMM('N','N',N,N,N,1.,U,N,A,N,0.,X2,N,CT,NPROCS,
     &            TEAM,PS)
      CALL SGEMM('N','T',N,N,N,1.,X2,N,U,N,0.,A,N,CT,NPROCS,
     &            TEAM,PS)
      CALL SGEMM('N','T',N,N,N,1.,U,N,V,N,0.,V,N,CT,NPROCS,
     &            TEAM,PS)
*     .. Form the vector of the eigenvalues ..
C*KSR* TILE ( I , TEAMID = TEAM)
       DO 6 I=1,N
         EIG(I) = A(I,I)
    6    CONTINUE
```

```
C*KSR* END TILE
*
*      .. Sort the vectors of the eigenvalues in descending order ..
*
       CALL SORT01 (EIG,INDEX,N, TEAM)
*
*      .. Permute the columns of V in order to agree with EIG ..
*
C*KSR* TILE ( J,I, TEAMID = TEAM )
       DO  J=1,N
         DO  I=1,N
           PP(I,J) = V(I,J)
         END DO
       END DO
C*KSR* END TILE
*
C*KSR* TILE ( J,I, TEAMID = TEAM )
       DO  J=1,N
         DO  I=1,N
           V(I,J) = PP(I,INDEX(J))
         END DO
       END DO
C*KSR* END TILE
*
*     .. Compute the Frobenius norm of the off-diagonal entries
*        of the nearly diagonal matrix A ..
*
      OFFA = OFF(A,N,TEAM)
*
*     End of METHOD2
*
      END


      SUBROUTINE UPDATE_COLUMNS(K, A, V, N, NB, NB2, NPROCS,
     &            TOP, BOT,THRESH,SWEEP,P,Q,BL, THR)
*
*     .. Scalar Arguments ..
      INTEGER K, N, NB, NB2, NPROCS, SWEEP
      REAL THRESH
*     .. Array Arguments ..
      INTEGER TOP(NPROCS), BOT(NPROCS), P(NPROCS), Q(NPROCS)
      REAL A(N,N), V(N,N), BL(NPROCS,NB,NB)
      LOGICAL THR(NPROCS)
*
*     Purpose
*     ======
*
*     UPDATE_COLUMNS performs the following tasks:
*
*     1) Constructs the matrix block that corresponds to the
*        K-th index pair of the current Jacobi set.
```

```
*     2) Checks whether the threshold condition is satisfied
*        and if updates the logical variable THR(K).
*     3) If THR(K) is TRUE, computes the Schur Decomposition
*        of SH and updates the K-th block columns of A and V.
*
*     .. Parameters ..
*     Parameter NX must be set to the size of the original
*     matrix.
      INTEGER NX, NS, LWORK
      PARAMETER(NX = 64, NS = NX/16, LWORK = 3*NX-1)
      REAL ZERO
      PARAMETER (ZERO = 0.)
*     ..
*     .. Local Scalars ..
*
      INTEGER I, J, P1, P2, Q1, Q2
      REAL SS
*
*     .. Local Arrays ..
      REAL SH(NS,NS), AS(NX,NS), VS(NX,NS),
     &     W(NS), WORK(LWORK)
*
*     .. Executable Statements ..
*
      P(K) = (MIN (TOP(K), BOT(K))-1)*NB2
      Q(K) = (MAX (TOP(K), BOT(K))-1)*NB2
*
      P1 = P(K)+1
      P2 = P(K)+NB2
      Q1 = Q(K)+1
      Q2 = Q(K)+NB2
*
      BL(K,1:NB2,1:NB2)       = A(P1:P2,P1:P2)
      BL(K,1:NB2,NB2+1:NB)    = A(P1:P2,Q1:Q2)
      BL(K,NB2+1:NB,1:NB2)    = A(Q1:Q2,P1:P2)
      BL(K,NB2+1:NB,NB2+1:NB) = A(Q1:Q2,Q1:Q2)
*
      SS = ZERO
*
      DO J=1,NB-1
         DO I=J+1,NB
             SS = SS + BL(K,I,J)*BL(K,I,J)
         END DO
      END DO
*
      IF (SS .LT. THRESH) THEN
               THR(K) = .FALSE.
               RETURN
      END IF
*
      SH = BL(K,:,:)
*
```

```
       CALL SSYEV('V','U',NB,SH,NB,W,WORK,LWORK,INFO)
*
       BL(K,:,:) = SH
*
       AS(1:N,1:NB2)       =      A(1:N,P1:P2)
       AS(1:N,NB2+1:NB)    =      A(1:N,Q1:Q2)
       VS(1:N,1:NB2)       =      V(1:N,P1:P2)
       VS(1:N,NB2+1:NB)    =      V(1:N,Q1:Q2)
*
       CALL SGEMM('N','N',N,NB,NB,1.,AS,N,SH,
     &      NB,0.,AS,N)
*
       CALL SGEMM('N','N',N,NB,NB,1.,VS,N,SH,
     &      NB,0.,VS,N)
*
       A(1:N,P1:P2) = AS(1:N,1:NB2)
       A(1:N,Q1:Q2) = AS(1:N,NB2+1:NB)
       V(1:N,P1:P2) = VS(1:N,1:NB2)
       V(1:N,Q1:Q2) = VS(1:N,NB2+1:NB)
*
*      End of UPDATE_COLUMNS
*
       END


      SUBROUTINE UPDATE_ROWS(K, A,  N, NB, NB2, NPROCS,
     &                TOP, BOT, P, Q, BL, THR)
*
*     .. Scalar Arguments ..
      INTEGER K, N, NB, NB2, NPROCS
*     .. Array Arguments ..
      INTEGER TOP(NPROCS), BOT(NPROCS), P(NPROCS), Q(NPROCS)
      REAL A(N,N), BL(NPROCS,NB,NB)
      LOGICAL THR(NPROCS)
*
*     Purpose
*     =======
*
*     UPDATE_ROWS updates the K-th block row of A.
*
*     .. Parameters ..
*     Parameter NX must be set equal to the size of A.
      INTEGER NX, NS
      PARAMETER(NX = 64, NS = NX/16)
*     ..
*     .. Local Scalars ..
      INTEGER   P1, P2, Q1, Q2
*     .. Local Arrays ..
      REAL RG(NS,NX), BLCK(NS,NS)
*
*     .. Executable Statements ..
*
```

```
      P1 = P(K)+1
      P2 = P(K)+NB2
      Q1 = Q(K)+1
      Q2 = Q(K)+NB2
*
      RG(1:NB2,:)    = A(P1:P2,:)
      RG(NB2+1:NB,:) = A(Q1:Q2,:)
*
      BLCK = BL(K,:,:)
*
      CALL SGEMM('T','N',NB,N,NB,1.,BLCK,NB,RG,NB,0.,RG,NB)
*
      A(P1:P2,:) = RG(1:NB2,:)
      A(Q1:Q2,:) = RG(NB2+1:NB,:)
*
*     End of UPDATE_ROWS
*
      END
```

A.4 Auxiliary Routines

The following KSR Fortran routines are used by the codes given in Appendix A.1, Appendix A.2, and Appendix A.3.

```
      SUBROUTINE MUSIC (TOP, BOT, M, TEAM)
*
*     .. Scalar Arguments ..
      INTEGER M, TEAM
*     .. Array Arguments ..
      INTEGER TOP(M), BOT(M)
*
*     Purpose
*     =======
*
*     MUSIC computes the next Jacobi set using the
*     Tournament scheme.
*
*     .. Parameters ..
      INTEGER NP
      PARAMETER (NP = 16)
*
*     .. Local Scalars ..
      INTEGER I, K
*     .. Local Arrays ..
      INTEGER NEWTOP(NP), NEWBOT(NP)
*
*     .. Executable Statements ..
*
C*KSR* TILE ( K, TEAMID = TEAM )
       DO 2 K=1,M
```

```
*
         IF (K .EQ. 2) THEN
           NEWTOP(K) = BOT(1)
         ELSE IF (K .GT. 2) THEN
           NEWTOP(K) = TOP(K-1)
         END IF
*
         IF (K .EQ. M) THEN
           NEWBOT(K) = TOP(K)
         ELSE
           NEWBOT(K) = BOT(K+1)
         END IF
*
    2    CONTINUE
C*KSR* END TILE
*
C*KSR* TILE ( I , TEAMID = TEAM)
       DO 3 I=1,M
         TOP(I) = NEWTOP(I)
         BOT(I) = NEWBOT(I)
    3    CONTINUE
C*KSR* END TILE
*
      TOP(1) = 1
*
*     End of MUSIC
*
      END

      SUBROUTINE INITLS(A,N,TEAM)
*
*     .. Scalar Arguments ..
      INTEGER N, TEAM
*     .. Array Arguments ..
      REAL A(N,*)
*
*     Purpose
*     =======
*     INITLS creates the N x N identity matrix A
*
*     .. Parameters ..
      REAL ONE, ZERO
      PARAMETER(ONE = 1., ZERO = 0.)
*
*     .. Executable Statements ..
*
C*KSR* TILE ( J,I, TEAMID = TEAM )
       DO 3 J=1,N
         DO 3 I=1,N
           IF (I .EQ. J) THEN
             A(I,J) = ONE
```

```
          ELSE
            A(I,J) = ZERO

          END IF
    3  CONTINUE
C*KSR* END TILE
*
*     End of INITLS
*
      END


      SUBROUTINE SORT01 (A, INDEX, N, TEAM)
*
*     .. Scalar Arguments ..
*
      INTEGER N, TEAM
*     .. Array Arguments ..
*
      INTEGER INDEX(*)
      REAL A(*)
*
*     Purpose
*     =======
*
*     SORT01 sorts the elements of the vector A in
*     descending order. On exit, the vector INDEX
*     contains the positions of the elements of A
*     before sorting. For example, INDEX(I) = J
*     means that the I-th element of the sorted
*     vector was the J-th element of the original
*     vector.
*
C*KSR* TILE ( I , TEAMID = TEAM )
       DO 2 I=1,N
         INDEX(I) = I
    2    CONTINUE
C*KSR* END TILE
*
       DO 20 K=N,2,(-1)
         DO 10 I=1,K-1
           IF (A(I) .LT. A(I+1)) THEN
             TEMP = A(I)
             ITEMP = INDEX(I)
             A(I) = A(I+1)
             INDEX(I) = INDEX(I+1)
             A(I+1) = TEMP
             INDEX(I+1) = ITEMP
           END IF
   10      CONTINUE
   20    CONTINUE
*
```

```
*      End of SORT01
*
       END

       LOGICAL FUNCTION MODDAV(A,VECTOR,N,TEAM)
*
*      .. Scalar Arguments ..
*
       INTEGER N, TEAM
*      .. Array Arguments ..
       REAL A(N,*), VECTOR(*)
*
*      Purpose
*      =======
*
*      MODDAV checks whether the condition for
*      applying the Davies-Modi scheme is satisfied.
*
*      .. Intrinsic Functions ..
       INTRINSIC FLOAT, ABS
*      ..
*      .. Local Scalars ..
       REAL ALPHA, DELTA, EPSLON, E1, E2, E3
*      ..
*      .. Executable Statements ..
       MODDAV = .FALSE.
*
       ALPHA = ABS (A(2,1))
*
C*KSR* TILE ( J,PRIVATE=( I ),REDUCTION=( ALPHA ), TEAMID = TEAM )
       DO 2 J=1,N-1
         DO 2 I=J+1,N
           ALPHA = MAX (ALPHA, ABS (A(I,J)))
    2      CONTINUE
C*KSR* END TILE
*
C*KSR* TILE ( I , TEAMID = TEAM)
       DO 3 I=1,N
         VECTOR(I) = A(I,I)
    3    CONTINUE
C*KSR* END TILE
*
      CALL SORT02(VECTOR,N)
*
      DELTA = ABS (VECTOR(2) - VECTOR(1))
*
C*KSR* TILE ( I,REDUCTION=( DELTA ), TEAMID = TEAM )
       DO 4 I=2,N-1
         DELTA = MIN (DELTA, ABS (VECTOR(I+1) - VECTOR(I)))
    4    CONTINUE
C*KSR* END TILE
*
```

```
      E1 = FLOAT (N ** 3) * (ALPHA ** 4) / (DELTA ** 4)
      E2 = FLOAT (N ** 2) * (ALPHA ** 3) / (DELTA ** 2)
      E3 = FLOAT (N ** 2) * (ALPHA ** 4) / (DELTA ** 3)
*
      EPSLON = MAX (E1, E2, E3)
*
      IF (EPSLON .LT. 1.0E-03) MODDAV = .TRUE.
*
*     End of MODDAV
*
      END


      SUBROUTINE ADDMAT(A,B,C,N,TEAM)
*
*     .. Scalar Arguments ..
      INTEGER N, TEAM
*     .. Array Arguments ..
      REAL A(N,*), B(N,*), C(N,*)
*
*     Purpose
*     =======
*     ADDMAT adds two square matrices.
*
*     .. Local Scalars ..
      INTEGER I, J
*
C*KSR* TILE(J,I, TEAMID = TEAM)
       DO 10 J=1,N
         DO 10 I=1,N
            C(I,J) = A(I,J) + B(I,J)
 10   CONTINUE
C*KSR* END TILE
*
*     End of ADDMAT
*
      END


      SUBROUTINE ADXMAT(A,C,N,TEAM)
*
*     .. Scalar Arguments ..
      INTEGER N, TEAM
*     .. Array Arguments ..
      REAL A(N,*), C(N,*)
*
*     Purpose
*     =======
*
*     ADXMAT computes (1/2)( A + A').
*
*     .. Local Scalars ..
```

```
      INTEGER I, J
*
*      .. Executable Statements ..
*
C*KSR* TILE(J,I, TEAMID = TEAM)
      DO 10 J=1,N
         DO 10 I=1,N
            C(I,J) = 0.5*(A(I,J) + A(J,I))
 10   CONTINUE
C*KSR* END TILE
*
*     End of ADXMAT
*
      END


      SUBROUTINE SORT02 (A, N)
*
*     .. Scalar Arguments ..
      INTEGER N
*     .. Array Arguments ..
      REAL A(*)
*
*     Purpose
*     =======
*
*     SORT01 sorts the elements of the vector A in
*     descending order.
*
*     .. Local Scalars ..
*
      INTEGER I, J, K
      REAL TEMP
*     ..
*     .. Executable Statements ..
*
      DO 20 K=N,2,(-1)
        DO 10 I=1,K-1
           IF (A(I) .LT. A(I+1)) THEN
             TEMP = A(I)
             A(I) = A(I+1)
             A(I+1) = TEMP
           END IF
   10   CONTINUE
   20 CONTINUE
*
*     End of SORT02
*
      END


      SUBROUTINE EVNORM (A, N, NOR, TEAM)
```

```
*
*     .. Scalar Arguments ..
      INTEGER N, TEAM
      REAL NOR
*     .. Array Arguments ..
      REAL A(N,*)
*
*     Purpose
*     =======
*
*     EVNORM computes the Frobenious norm of a matrix.
*
*     .. Intrinsic Functions ..
      INTRINSIC SQRT
*     ..
*     .. local Scalars ..
      INTEGER I, J
*
      NOR = 0.
C*KSR* TILE ( J,I,REDUCTION=( NOR ),TEAMID = TEAM )
       DO 2 J=1,N
         DO 2 I=1,N
           NOR = NOR + A(I,J) ** 2
    2      CONTINUE
C*KSR* END TILE
*
      NOR = SQRT (NOR)
*
*     End of EVNORM
*
      END


      REAL FUNCTION OFF (A, N, TEAM)
*
*     .. Scalar Arguments ..
      INTEGER N, TEAM
*      .. Array Arguments ..
      REAL A(N,*)
*
*     OFF computes the Frobenius norm of
*     the off-diagonal entries of a symmetric
*     matrix A.
*
*     .. Parameters ..
      REAL ZERO
      PARAMETER (ZERO = .0)
*     ..
*     .. Intrinsic Functions ..
      INTRINSIC SQRT
*     ..
*     .. Local Scalars ..
```

```
      INTEGER I, J
      REAL SUM
*
*     .. Executable Statements ..
*
      SUM = ZERO
*
C*KSR* TILE ( J,PRIVATE=( I ),REDUCTION=( SUM ), TEAMID = TEAM )
       DO 2 J=1,N-1
         DO 2 I=J+1,N
           SUM  = SUM + A(I,J) ** 2
    2      CONTINUE
C*KSR* END TILE
*
      OFF = SQRT (2*SUM)
*
*     End of OFF
*
      END

      SUBROUTINE SYM2(SH)
*
*     .. Array Arguments ..
      REAL SH(2,2)
*
*     Purpose
*     =======
*
*     SYM2 computes the orthogonal factor of
*     the symmetric Schur decomposition
*     of the 2 x 2 matrix SH.
*
*     .. Intrinsic Functions ..
      INTRINSIC ATAN, COS, SIN, SQRT
*     ..
*     .. Local Scalars ..
      REAL P4, C, S, TAU, T, SIGNT
*
*     .. Executable Statements ..
*
      P4 = ATAN(1.0)
*
      IF (ABS(SH(2,2)-SH(1,1)) .LE. 1.1E-16) THEN
              C      =  SQRT(2.)/2.
              S      =  C
      ELSEIF (ABS(SH(1,2)) .GT. 1.1E-16)  THEN
         TAU    = (SH(2,2) - SH(1,1))/(2*SH(1,2))
         SIGNT  = TAU/ABS(TAU)
         T      = SIGNT/(ABS(TAU) + SQRT(1.0 + TAU**2))
         C      = 1.0/SQRT(1 + T**2)
         S      = T*C
      ELSE
```

```
          C      = 1.0
          S      = 0.0
       END IF
*
       SH(1,1) = C
       SH(2,2) = C
       SH(1,2) = S
       SH(2,1) = -S
*
*      End of SYM2
*
       END
```

```
       LOGICAL FUNCTION MODDAV(A,VECTOR,N,TEAM)
*
*      .. Scalar Arguments ..
*
       INTEGER N, TEAM
*      .. Array Arguments ..
       REAL A(N,*), VECTOR(*)
*
*      Purpose
*      =======
*
*      MODDAV checks whether the condition for
*      applying the Davies-Modi scheme is satisfied.
*
*      .. Intrinsic Functions ..
       INTRINSIC FLOAT, ABS
*      ..
*      .. Local Scalars ..
       REAL ALPHA, DELTA, EPSLON, E1, E2, E3
*      ..
*      .. Executable Statements ..
       MODDAV = .FALSE.
*
       ALPHA = ABS (A(2,1))
*
C*KSR* TILE ( J,PRIVATE=( I ),REDUCTION=( ALPHA ), TEAMID = TEAM )
       DO 2 J=1,N-1
         DO 2 I=J+1,N
           ALPHA = MAX (ALPHA, ABS (A(I,J)))
    2      CONTINUE
C*KSR* END TILE
*
C*KSR* TILE ( I , TEAMID = TEAM)
       DO 3 I=1,N
         VECTOR(I) = A(I,I)
    3    CONTINUE
C*KSR* END TILE
```

```
*
      CALL SORT02(VECTOR,N)
*
      DELTA = ABS (VECTOR(2) - VECTOR(1))
*
C*KSR* TILE ( I,REDUCTION=( DELTA ), TEAMID = TEAM )
       DO 4 I=2,N-1
         DELTA = MIN (DELTA, ABS (VECTOR(I+1) - VECTOR(I)))
    4    CONTINUE
C*KSR* END TILE
*
      E1 = FLOAT (N ** 3) * (ALPHA ** 4) / (DELTA ** 4)
      E2 = FLOAT (N ** 2) * (ALPHA ** 3) / (DELTA ** 2)
      E3 = FLOAT (N ** 2) * (ALPHA ** 4) / (DELTA ** 3)
*
      EPSLON = MAX (E1, E2, E3)
*
      IF (EPSLON .LT. 1.0E-03) MODDAV = .TRUE.
*
*     End of MODDAV
*
      END
```

Bibliography

[1] E. Anderson, Z. Bai, C. H. Bischof, J. W. Demmel, J. J. Dongarra, J. J. Du Croz, A. Greenbaum, S. J. Hammarling, A. McKenney, S. Ostrouchov, and D. C. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, 1992.

[2] H. C. Andrews and C. I. Patterson. Singular Value Decomposition and Digital Image Processing. *IEEE Trans. Accoustics, Speech and Signal Processing*, 24:26–53, 1976.

[3] K. S. Arun. A unitarily constrained total least squares problem in signal processing. *SIAM J. Matrix Anal. Appl.*, 13(3):729–745, 1992.

[4] L. Autonne. Sur les Groupes Linéaires, Réels et Orthogonaux. *Bull. Soc. Math. France*, 30:121–134, 1902.

[5] L. Autonne. Sur les Matrices Hypohermitiennes et sur les Matrices Unitaires. *Ann. Univ. Lyon*, Nouvelle Série I, Fasc. 38:1–77, 1915.

[6] G. Baker. Essentials on Padé approximants. *Academic Press*, New York, 1975.

[7] V. Bargmann, C. Montgomery and J. von Neumann. Solution of Linear Systems of High Order. Princeton: Institute for Advanced Study, 1946.

[8] I. Y. Bar-Itzhack. Iterative optimal orthogonalization of the strapdown matrix. *IEEE Trans. Aerospace and Electronic Systems*, AES-11(1):30–37, 1975.

[9] T. E. Bechrakis and E. Nicolaides. Statistical Analysis of Lexical Data. *National Centre for Social Research*, Athens, 1990. (In Greek).

[10] D. P. Bertsekas. The Auction Algorithm: A Distributed Relaxation Method for the Assignment Problem. *Ann. Oper. Res.*, 14:105–123, 1988.

[11] D. P. Bertsekas and D. A. Castãnon. Parallel Synchronous and Asynchronous Implementations of the Auction Algorithm. *Parallel Computing*, 17:707–732, 1991.

[12] C. H. Bischof and C. F. Van Loan. The WY Representation for Products of Householder Matrices. *SIAM J. Sci. Statist. Computing*, 8:2–13, 1987.

[13] Å. Björck and C. Bowie. An iterative algorithm for computing the best estimate of an orthogonal matrix. *SIAM J. Numer. Anal.*, 8(2):358–364, 1971.

[14] S. D. Blostein and T. S. Huang. Estimating 3–D motion from range data. *Proc. First Conference on Artificial Intelligence and Applications*, Denver, CO:246–250, 1984.

[15] E. Boman. Experiences on the KSR1 Computer. Report RNR-93-008, NAS Systems Division, Applied Research Branch, NASA Ames Research Center, Moffett Field, California, 1993.

[16] R. P. Brent and F. T. Luk. The Solution of Singular Value and Symmetric Eigenvalue Problems on Multiprocessor Arrays. *SIAM J. Sci. and Stat. Comp*, 6:69–84, 1985.

[17] K. W. Brodlie and M.J. D. Powell. On the Convergence of Cyclic Jacobi Methods. *J. Inst. Math. Appl.*, 15:279–87, 1975.

[18] D. S. Broomhead and D. Lowe. Multivariable Functional Interpolation and Adaptive Networks. *Complex Systems*, 2: 321–355, 1988.

[19] K. E. Bullen. An Introduction to the Theory of Seismology. Cambridge University Press, Cambridge, England, 1947.

[20] G. Carpaneto and P. Toth. Algorithm 548. Solution of the Assignment Problem. *ACM Transactions on Mathematical Software*, 6:104-111, 1980.

[21] R. B. Cattell. Factor Analysis. An Introduction and Manual for the Phychologists and Social Scientists. Harper & Bros, New York, 1952.

[22] R. B. Cattell. The scientific use of factor analysis in behavioral and life sciences. Plenum Press, New York, 1978.

[23] Graham Charters. Transform Techniques for Digital Image Compression. *M.Sc. Thesis*, Department of Mathematics, University of Manchester, 1991.

[24] C. W. Churchman, R. L. Ackoff, E. L. Arnoff. Introduction to Operations Research. John Wiley & Sons, Inc., 1957.

[25] C. H. Coombs. A Theory of Data. John Wiley, New York, 1964.

[26] D. Corneil. Eigenvalues and Orthogonal Eigenvectors of Real Symmetric Matrices. *Master's Thesis*, Dept. of Computer Science, University of Toronto, 1965.

[27] D. Cyganski and J. A. Orr. Applications of tensor theory to object recognition and orientation determination. *IEEE Trans. Pattern Anal. mach. Intelligence*, PAMI–7:663–673, 1985.

[28] R. O. Davies and J. J. Modi. A Direct Method for Completing Eigenproblem Solution on a Parallel Computer. *Linear Algebra and its Applications*, 77:61–74, 1986.

[29] J. W. Demmel and A. McKenney A Test Matrix Generation Suite. *LAPACK Working Note 9*, Courant Institute, New York, 1989.

[30] J. W. Demmel and K. Veselić. Jacobi's Method is More Acurate than QR. *SIAM J. Matrix Anal. Appl*, 13(4):1204–1245, 1992.

[31] J W. Demmel. Trading off parallelism and numerical stability. In Marc S. Moonen, Gene H. Golub, and Bart L. De Moor, editors, *Linear Algebra for Large Scale and Real-Time Applications*, volume 232 of *NATO ASI Series E*, pages 49–68. Kluwer Academic Publishers, Dordrecht, 1993.

[32] J. J. Dongara, I. S. Duff, D. C. Sorensen, and H. A. van der Vorst. Solving Linear Systems on Vector and Shared Memory Computers. Society for Industrial and Applied Mathematics, Philadelphia, 1991.

[33] H. E. Doran. Applied Regression in Econometrics. *STATISTICS: Textbooks and Monographs 102*, Marcel Dekker, Inc., New York, 1989.

[34] N. R. Draper and H. Smith. Applied Regression Analysis. 2nd Edition, John Wiley, New York, 1981.

[35] T. H. Dunigan. Kendall Square Multiprocessor: Early Experiences and Performance. Report ORNL/TM-12065, Oak Ridge National Laboratory, Oak Ridge, TN, 1992.

[36] P.S. Dwyer. The solution of the Hitchcock Transportation Problem with a method of Reduced Matrices. Statistical Laboratory, University of Michigan, 1955. (Privately circulated.)

[37] P. J. Eberlein. On Using the Jacobi Method on a Hypercube. *Hypercube Multiprocessors*, ed. M.T. Heath, SIAM Publications, Philadelphia, 1987.

[38] J. Egerváry. Matrixok Kombinatorius Tulajdonságairól. *Matematikai és Fisikai Lapok*, 38:16–28, 1931. Translated by H.W. Kuhn as “Combinatotial properties of matrices”, in *ONP Logistics Project*, Princeton University, 1953.

[39] K. Fan and A. J. Hoffman. Some metric inequalities in the space of matrices. *Proc. Amer. Math. Soc.*, 6:111–116, 1955.

[40] G. E. Forsythe and P. Henrici. The Cyclic Jacobi Method for Computing the Principal Values of a Complex Matrix. *Trans. Amer. Math. Soc.*, 94:1–23, 1960.

[41] M. M. Flood. On the Hitchcock Distribution Problem. *Pac. J. Math.*, 3(2):369–386, 1953.

[42] W. Gander. On Halley’s iteration method. *Amer. Math. Monthly*, 92:131–134, 1985.

[43] W. Gander. Algorithms for the polar decomposition. *SIAM J. Sci. Stat. Comput.*, 11(6):1102–1115, 1990.

[44] A. George, M.T. Heath, and J. Liu. Parallel Cholesky factorization on a shared memory multiprocessor. *Lin. Alg. and Its Applic.*, 77:165:187, 1986.

[45] J. A. Goldstein and M. Levy. Linear algebra and quantum chemistry. *Amer. Math. Monthly*, 98(8):710–718, 1991.

[46] G. H. Golub and C. Reinsch. Singular value decomposition and least squares solutions. *Numer. Math.*, 14:403–420, 1970.

[47] G. H. Golub and C. F. Van Loan. Matrix Computations. *Johns Hopkins University Press*, Baltimore, MD, 2nd ed., 1989.

[48] J. C. Gower. Multivariate analysis: Ordination, multidimensional scaling and allied topics. In E. H. Lloyd, editor, *Statistics*, volume VI of *Handbook of Applicable Mathematics,*, pages 727–781. John Wiley, Chichester, 1984.

[49] L. A. Guttman. A New Approach to Factor Analysis: The Radex. In *Mathematical Thinking in the Social Sciences*, ed. P.F. Lazarsfeld, Glencoe, Illinois: Free Press, 1954.

[50] M. J. Greenacre. Theory and Applications of Correspondence Analysis. *Academic Press*, 1984.

[51] M. G. Greenberg. A Method of Successive Cumulations for the Scaling of Pair Comparison Preference Judjements. *Phychometrica*, 30:441–448, 1965.

[52] R. F. Gunst and R. L. Mason. Regression Analysis and its Applications. *STATISTICS: Textbooks and Monographs 34*, Marcel Dekker, Inc., New York, 1980.

[53] E. Hagersten, A. Landin, and S. Haridi. DDM — A Cache-Only Memory Architecture. *IEEE Computer*, 25(9):45–54, 1992.

[54] P. R. Halmos. Bad Products of Good Matrices. *Linear and Multilinear Algebra*, 29:1–20, 1991.

[55] S. Hammarling. The Singular Value Decomposition in Multivariate Statistics. *ACM SIGNUM Newsletters*, 20(3):2–25, 1985.

[56] R. J. Hanson and M. J. Norris. Analysis of measurements based on the singular value decomposition. *SIAM J. Sci.Stat. Comput.*, 2(3):363–373, 1981.

[57] E. R. Hansen. On Cyclic Jacobi Methods. *SIAM J. Appl. Math.*, 11:448-59, 1963.

[58] C. T. Hare. Light Duty Emission Correction Factors for Ambient Conditions. Final Report to the Environment Protection Agency under contract No. 68–02–1777, 1977.

[59] P. Henrici. On the Speed of Convergence of Cyclic and Quasicyclic Jacobi Methods for Computing Eigenvalues of Hermitian Matrices. *SIAM J. Appl. Math.*, 6:146–62, 1958.

[60] J. Hertz, A. Krogh, and R. G. Palmer. Introduction to the Theory of Neural Computation. *Lecture notes Volume 1*, Santa Fe Institute Studies in the Sciences of Complexity, Addison-Wesley, 1992.

[61] N. J. Higham. Computing the polar decomposition—with applications. *SIAM J. Sci. Stat. Comput.*, 7(4):1160–1174, October 1986.

[62] N. J. Higham. Computing real square roots of a real matrix. *Linear Algebra and Applic.*, 88/89:504–430, 1987.

[63] N. J. Higham. The Symmetric Procrustes Problem. *BIT*, 28:133–143, 1988.

[64] N. J. Higham. The matrix sign decomposition and its relation to the polar decomposition. Numerical Analysis Report No. 225, University of Manchester, England, 1993. To appear in *Linear Algebra and its Applications.*

[65] N. J. Higham and P. Papadimitriou. Parallel Singular Value Decomposition via the Polar Decomposition. Technical Report No. 239, Department of Mathematics, University of Manchester, 1993.

[66] N. J. Higham and R. S. Schreiber. Fast polar decomposition of an arbitrary matrix. *SIAM J. Sci. Stat. Comput.*, 11(4):648–655, July 1990.

[67] R. A. Horn and C. R Johnson. Matrix Analysis. *Cambridge University Press*, 1985.

[68] R. A. Horn and C. R Johnson. Topics in Matrix Analysis. *Cambridge University Press*, 1991.

[69] J. R. Hurley and R. B. Cattell. Producing direct rotation to test a hypothesized factor structure. *Behavioral Science*, 7:258–262, 1962.

[70] K. Hwang. Advanced Computer Architecture: Parallelism, Scalability, Programmmability. McGraw-Hill Inc., 1993

[71] C. G. Jacobi. Über ein Leichtes Verfahren die in der Theorie der Säculärstörungen vorkommenden Gleichungen Numerisch Aufzulösen. *Crelle's J*, 30:51–94, 1846.

[72] S. M. Kay and S. L. Marple, Jr. Spectrum analysis — A modern perspective. *Proc. IEEE* 69:1380–1419, 1981.

[73] Kendall Square Research Corporation. *KSR Fortran Programming.* Waltham, MA, 1991.

[74] Kendall Square Research Corporation. *KSR OS User's Guide.* Waltham, MA, 1991.

[75] Kendall Square Research Corporation. *KSR Parallel Programming.* Waltham, MA, 1991.

[76] Kendall Square Research Corporation. *Technical Summary.* Waltham, MA, 1992.

[77] Kendall Square Research Corporation. *KSRlib/BLAS Library, Version 1.0, Installation Guide and Release Notes.* Waltham, MA, 1993.

[78] Kendall Square Research Corporation. *KSRlib/LAPACK Library, Version 1.0b BETA, Installation Guide and Release Notes.* Waltham, MA, 1993.

[79] C. Kenney and A. J. Laub. rational iterative methods for the matrix sign function. *SIAM J. Matrix Anal. Appl.*, 12(2):273–291, 1991.

[80] C. Kenney and A. J. Laub. On scaling Newton's method for polar decomposition and the matrix sign function. *SIAM J. Matrix Anal. Appl.*, 13(3):688–706, 1992.

[81] D. König. Theorie der Endlichen und Unendlichen Graphen. Chelsea Publishing Co., New York, 1950.

[82] H. W. Kuhn. The Hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2:83–97, 1955.

[83] D. J. Kuck and A. H. Sameh. Parallel Computation of Eigenvalues of Real Matrices. *Infomation Processing 1971*, North Holland, Amsterdam, 1266–1272, 1972.

[84] L. Lebart, A. Morineau and K. M. Warwick. Multivariate Descriptive Statistical Analysis. Correspondence Analysis and Related Techniques for Large Matrices. John Wiley and Sons, Inc., New York, 1984.

[85] H. J. Loether, D. G. McTavish, and P. M. Voxland. Statistical Analysis for Sociologies: A Student Manual. Allyn and Bacon Inc., Boston, 1974.

[86] Y. Makris. An investigation of the Classification Performance of RBFNs. M. Sc. Thesis, Department of Computer Science, University of Manchester, 1993.

[87] J. Mandel. Use of the Singular Value Decomposition in Regression Analysis. *The American Statistician*, 36(1):15–24, 1982.

[88] R. Mathias. Pertubation Bounds for the Polar Decomposition. Manuctript, 1991. To appear in *SIAM J. Matrix Anal. Appl.*

[89] S. Merrill. Draft Report on Housing Expenditures and Quality, Part III: Hedonic Indices as a Measure of Housing Quality. Abt Associates Inc., Cambridge, Mass., 1977.

[90] M. Metcalf and J. Reid. Fortran 90 explained. Oxford University Press, 1990.

[91] J. J. Modi and J. D. Pryce. Efficient Implementation of Jacobi's Diagonalisation Method on the DAP. *Numer. Math.*, 46:443–454, 1985.

[92] J. J. Modi. Parallel Algorithms and Matrix Computations. Clarendon Press, Oxford, England, 1988.

[93] M. T. Musavi, W. Ahmed, K. H. Chan, K. B. Faris and D. M. Hummels. On the Training of Radial Basis Function Classifiers. *Neural Networks*, 5:595–603, 1992.

[94] J. L. Myers. Fundamentals of Experimental Design. (3rd Edition), Allyn and Bacon, Boston, 1979.

[95] Numerical Algorithms Group Limited. *NAG Fortran Library Coincise Reference*. Oxford, UK, 1991.

[96] P. Pandey, C. Kenney and A. J. Laub. A Parallel Algorithm for the Matrix Sign Function. *Int. J. High Speed Computing*, 2(2):181–191, 1990.

[97] C. H. Papadimitriou and K. Steiglitz. Combinatorial Optimization. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1982.

[98] B. N. Parlett. The Symmetric Eigenvalue Problem. *Prentice–Hall*, Englewood Cliffs, N.J., 1980.

[99] U. Ramachandran, G. Shah, S. Ruvicumar, and J. Muthukumarasamy. Scalability Study of the KSR1. Technical Report GIT-CC 93/03, College of Computing, Georgia Institute of Technology, Atlanta, 1993.

[100] J. O. Rawlings. Applied Regression Analysis. Wadsworth and Brooks/Cole Advanced Books and Software, Belmond, CA, 1988.

[101] J. D. Roberts. Linear model reduction and solution of the algebraic Ricatti equation by use of the sign function. *Int. J. Control*, 32(4):677–687, 1980. First issued as report CUED/B-control/TR13, Department of Engineering, University of Cambridge, 1971.

[102] E. A. Robinson. Least Squares Regression Analysis in Terms of Linear Algebra. *Goose Pond Press*, Houston, Texas, 1981.

[103] H. Rutishauser. The Jacobi Method for Real Symmetric Matrices. *Numer. Math.*, 9:1–10, 1966.

[104] A. Sameh. On Jacobi and Jacobi-like Algorithms for Parallel Computers. *Math. Comp.*, 25:579–90, 1971.

[105] J. H. F. Schilderinck. Regression and Factor Analysis Applied in Ecomometrics. Martinus Nijhoff Social Sciences Division, Leiden 1977.

[106] P. H. Schönemann. A generalized Solution of the Orthogonal Procrustes problem. *Phychometrica*, 31:1–10, 1966.

[107] P. H. Schönemann. On two-sided orthogonal Procruste problems. *Phychometrica*, 33:19–33, 1968.

[108] A. Schönhage. On the Quadratic Convergence of the Jacobi Process. *Numer. Math.*, 6:410:12, 1964.

[109] R. Schreiber. Solving Eigenvalue and Singular Value Problems on an Undersized Systolic Array. *SIAM J. Sci. and Stat. Comp.*, 7:441–51, 1986.

[110] R. S. Schreiber and B. N. Parlett. Block reflectors: Theory and computation. *SIAM J. Numer. Anal.*, 25(1):189–205, 1988.

[111] H. W. Sewell, A. D. Haller, and G. W. Ohlendorf. The Educational and Early Occupational Attaintment Process: Peplication and Revision. *American Sociological Review*, 70:1014–27, 1970.

[112] R. I. Shrager. Optical Spectra from Chemical Titration: An Analysis by SVD. *SIAM J. Alg. Disc. Meth.*, 5(3):351–358, 1984.

[113] G. Shroff and R. Schreiber. Convergence of Block Jacobi Methods. Report 87–25, Comp. Sci. Dept., Rensselaer Polytechnic Institute, Troy, NY, 1987.

[114] J. P. Singh and J. L. Hennessy. An Empirical Investigation of the effectiveness and Limitations of Automatic Parallelization. Technical Report No CSL-TR-91-462, Computer System Laboratories, Stanford University, Stanford, CA, 1991.

[115] H. A. Taha. Operations Research. Macmillan Publishing Co., Inc., 1976.

[116] O. Tausky. The Role of Symmetric Matrices in the Study of General Matrices. *Linear Algebra and its Applications*, 5:147–154, 1972.

[117] M. Togai. An application of the Singular Value Decomposition to Manipulability and Sensitivity of Industrial Robots. *SIAM J. Alg. Disc. Meth.*, 7(2):315–320, 1986.

[118] R.C. Tryon. Identification of Social Areas by Cluster Analysis. *University of California Publications in Psychology*, 8(5), 1955.

[119] C. F. Van Loan. The Block Jacobi Method for Computing the Singular Value Decomposition. In *Computational and Combinatorial Methods in Systems Theory*, C. I. Byrnes and A. Lindquist eds., Elsevier Science Publishers B. V. (North Holland), 1986.

[120] D. F. Votaw and A. Orden. The Personnel Assignment Problem. In *Symposium on Linear Inequalities and Programming*, A. Orden and L. Goldstein (eds.), Project SCOOP, Headquarters, U.S. Air Forse, Washington:155–163, 1952.

[121] G. Wahba. Problem 65–1, A least squares Estimate of Satellite Attitude. *SIAM Review*, 8:384–385, 1966.

[122] J. H. Wilkinson. Note on the Quadratic Convergence of the Cyclic Jacobi Process. *Numer. Math.*, 6:269–300, 1962.

[123] J. H. Wilkinson. The Algebraic Eigenvalue Problem. *Clarendon Press*, Oxford, England, 1965.

[124] J. Williamson. A Polar Representation of Singular Matrices. *Bull. Amer. Math. Soc.*, 41:118–123, 1935.

[125] A. Winter and F. D. Murnaghan. On the Polar Respresentation of Non-singular Square Matrices. *Proc. National Academy of Sciences (U.S.)*, 17:676–678, 1931.