11

OF / DE

CANADIAN THESES ON MICROFICHE

THÈSES CANADIENNES SUR MICROFICHE

National Library of Canada
Collections Development Branch

Canadian Theses on
Microfiche Service

Ottawa, Canada
K1A 0N4

Bibliothèque nationale du Canada
Direction du développement des collections

Service des thèses canadiennes
sur microfiche

NOTICE

The quality of this microfiche is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30. Please read the authorization forms which accompany this thesis.

AVIS

La qualité de cette microfiche dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30. Veuillez prendre connaissance des formules d'autorisation qui accompagnent cette thèse.

THIS DISSERTATION
HAS BEEN MICROFILMED
EXACTLY AS RECEIVED

LA THÈSE A ÉTÉ
MICROFILMÉE TELLE QUE
NOUS L'AVONS REÇUE

Canada

0-315-19919-

National Library
of CanadaBibliothèque nationale
du Canada

Canadian Theses Division Division des thèses canadiennes

Ottawa, Canada
K1A 0N4

67826

PERMISSION TO MICROFILM — AUTORISATION DE MICROFILMER

• Please print or type — Écrire en lettres moulees ou dactylographier

Full Name of Author — Nom complet de l'auteur

SHANTHARAN, SHUBHA

Date of Birth — Date de naissance

16th Oct. 1955

Country of Birth — Lieu de naissance

INDIA.

Permanent Address — Résidence fixe

209, S. Oak # 130, Ames IA 50010, U.S.A.

Title of Thesis — Titre de la thèse

SOLVING SPARSE POSITIVE DEFINITE SYSTEMS
ON PIPELINE PROCESSORS

University — Université

CARLETON UNIVERSITY

Degree for which thesis was presented — Grade pour lequel cette thèse fut présentée

MASTER OF SCIENCE — INFORMATION AND SYSTEM SCIENCE

Year this degree conferred — Année d'obtention de ce grade

1985

Name of Supervisor — Nom du directeur de thèse

Dr. J. D. DIXON

Permission is hereby granted to the NATIONAL LIBRARY OF CANADA to microfilm this thesis and to lend or sell copies of the film.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

L'autorisation est, par la présente, accordée à la BIBLIOTHÈQUE NATIONALE DU CANADA de microfilmer cette thèse, de prêter ou de vendre des exemplaires du film.

L'auteur se réserve les autres droits de publication, ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés autrement reproduits sans l'autorisation écrite de l'auteur.

Date

17th April 1985

Signature

Shu

Shubha Shantharan

SOLVING SPARSE POSITIVE DEFINITE SYSTEMS ON PIPELINE PROCESSORS

by

SHUBHA SHANTHARAM, M.Sc.

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment of
the requirements for the degree of

Master of Science

Department of Mathematics and Statistics

Carleton University

Ottawa, Ontario

April 1, 1985

© copyright

1985, Shubha Shantharam

The undersigned hereby recommend to
the Faculty of Graduate Studies and Research
acceptance of the thesis, submitted by

SHUBHA SHANTHARAM, M.Sc.

in partial fulfilment of the requirements
for the degree of Master of Science.

B. M. Puttaswamaiah

B. M. Puttaswamaiah, Chairman
Department of Mathematics and Statistics

J. D. Dixon

J. D. Dixon, Thesis Supervisor

Carleton University

April 1985

All these structures have evolved over a span of 30 years. The first machine to use a kind of overlap is UNIVAC I, where program execution was overlapped with certain I/O activities. Partitioning of instruction execution units and using pipelining techniques were first introduced in the IBM machine called STRETCH and interleaving of memory was used by the IBM 7094-II to speed up memory access time. By 1967, the CDC-6600 had separate execution units for floating point operations and integer operations. With the demand for a higher execution rate there evolved VECTOR PROCESSORS in which a single function was executed successively on many individual data. (Such sets of data were called VECTORS.) Three vector processors which have internal pipelines are described in Chapter 2. Computer designers have used specialised pipelines (CRAY 1) as well as general purpose pipelines (CYBER 205) in these vector processors.

Another radical change in the computer architecture was to introduce replication of processors which resulted in the construction of the ILLIAC IV in the 1970's. Though ILLIAC IV is sometimes regarded as a failure, its influence has been profound. Many array processors like PEPE, BSP, ICLDAP have used the idea of providing an array of PE's (processing elements) and also have used the above mentioned overlapping ideas. Now the differences between pure pipelining, overlapping and pure parallelism have become quite indistinct. The machines like CRAY X-MP, CYBER 205 are the examples of the present trend.

ABSTRACT

Development of parallel and pipelined processors has initiated re-examination of the use of direct methods in solving large linear systems $Ax = b$, where A is an $n \times n$ sparse positive definite matrix. We consider those algorithms applicable to CRAY-like vector computers. Looking first at the conventional approaches to solve sparse systems, namely the band and envelope methods, we find that these are easily vectorizable and quite fast on CRAY-like computers. However general sparse methods tend to produce small vectors which do not use the CRAY-like computers to advantage. Different problems lead to different conclusions. We therefore look at methods like the minimum degree method, quotient tree ordering, and the nested dissection method which retain to some extent the sparseness of the original matrix in the factorization process and are compatible with a vectorized computation.

TABLE OF CONTENTS

	Page
Acceptance Sheet	ii
Abstract	iii
Acknowledgements	iv
Table of Contents	v
Introduction	1
Chapter 1: History and Application of Parallel Processors	2
1.0 Introduction	2
1.1 Evolution of Parallel Processors	2
1.2 Applications of Parallel Processing	4
Chapter 2: Pipelined and Parallel Processors	8
2.0 Introduction	8
2.1 Pipelined Processors	8
2.2 Vector Computers	12
2.3 Pipelined Computers	12
2.4 Processor Arrays	23
Chapter 3: Band and Envelope Methods for Sparse Symmetric Positive Definite Systems	29
3.0 Introduction	29
3.1 CRAY-like Processors	29
3.2 Matrix-vector Multiplication	31
3.3 Matrix Multiplication	33
3.3.1 Inner-product Method	33
3.3.2 Middle-product Method	34
3.3.3 Outer-product Method	34
3.4 Multiplication of Matrices by Diagonals	35
3.5 Recurrence	36
3.6 Cyclic Reduction Method	38
3.7 Linear Equations	40
3.8 Triangular Systems	45
3.9 Sparse Systems	47
3.9.1 Band Method	48
3.10 The Envelope Method	50
Chapter 4: Minimum Degree Methods	54
4.0 Introduction	54
4.1 Elimination Graph Model	54
4.2 Elimination by Reachable Sets	55
4.3 The Quotient Graph Model	55
4.4 Algorithms for Elimination Graphs	57
4.4.1 Elimination model and Reachable Set Model	57
4.4.2 Quotient Graph Model	58

TABLE OF CONTENTS (Continued)

4.5	Minimum Degree Algorithm	58
4.5.1	Minimum Degree Algorithm Using the Elimination Graph Method	59
4.5.2	Minimum Degree Algorithm Using the Reachable Sets	59
4.5.3	Minimum Degree Algorithm Using the Quotient Graph Model	61
Chapter 5:	Block Elimination	63
5.0	Introduction	63
5.1	Factorization of a Block Two By Two Matrix	63
5.2	Quotient Tree Partitioning	66
5.3	Nested Dissection Methods	70
Chapter 6:	Conclusions	73
References		75

INTRODUCTION

The technological revolution in the computer hardware industry has introduced new computers called vector processors. Use of vector processors to solve large sparse system of equations has attracted numerical analysts for a long time, but the existing algorithms to solve sparse systems fail to give the expected increase in their performance on these computers. The algorithms need to be redesigned with the specific architecture in mind.

In Chapter 1 we consider in brief the history of the development of these computers and a couple of examples where they are gainfully employed. In Chapter 2 we describe the salient features of the existing vector processors and how pipelined techniques are used to increase the performance. In Chapter 3 we consider methods for banded sparse positive definite systems. In Chapter 4 we examine the minimum degree method as suggested by George and Liu (1981) and in Chapter 5 the use of quotient tree ordering and nested dissection in the block elimination method.

History and Application of Parallel Processors

1.0 Introduction

In this chapter we consider in brief, the history of development, of the present day supercomputers. New architectural features used by different computers have enabled us to reach computational speeds up to millions of floating point operations per second. These supercomputers have become the necessary tools for science and engineering. In Section 1.2 we consider the use of these computers in two fields: in the study of seismic waves for the location of oil resources, and in basic research — to calculate the ground state energy of He_4 atoms in nuclear physics.

1.1 Evolution of Parallel Processors

There has been a tremendous improvement in the speed of executing arithmetic operations since the time that the UNIVAC I was produced commercially. This increased speed can be attributed mainly to improvement in hardware structure and parallelism. Parallelism refers to the ability to overlap or perform many tasks simultaneously. The principal ways of introducing parallelism into the architecture are

- (a) pipelining: using assembly line techniques to improve performance;
- (b) functional: providing several processing elements or processors all under a common control unit, performing the same operations;
- (c) multiprocessing: providing several processors each having its own memory and following its own instructions, but also communicating with the others via a common memory.

All these structures have evolved over a span of 30 years. The first machine to use a kind of overlap is UNIVAC I, where program execution was overlapped with certain I/O activities. Partitioning of instruction execution units and using pipelining techniques were first introduced in the IBM machine called STRETCH and interleaving of memory was used by the IBM 7094-II to speed up memory access time. By 1967, the CDC-6600 had separate execution units for floating point operations and integer operations. With the demand for a higher execution rate there evolved VECTOR PROCESSORS in which a single function was executed successively on many individual data. (Such sets of data were called VECTORS.) Three vector processors which have internal pipelines are described in Chapter 2. Computer designers have used specialised pipelines (CRAY 1) as well as general purpose pipelines (CYBER 205) in these vector processors.

Another radical change in the computer architecture was to introduce replication of processors which resulted in the construction of the ILLIAC IV in the 1970's. Though ILLIAC IV is sometimes regarded as a failure, its influence has been profound. Many array processors like PEPE, BSP, ICLDAP have used the idea of providing an array of PE's (processing elements) and also have used the above mentioned overlapping ideas. Now the differences between pure pipelining, overlapping and pure parallelism have become quite indistinct. The machines like CRAY X-MP, CYBER 205 are the examples of the present trend.

1.2 Applications of Parallel Processing

The demand for fast and efficient computing has increased many times in recent years, largely because of the requirements in scientific and engineering fields. A computer study and simulation of a physical model is often much more cost-effective and faster than physical experiments which would otherwise be necessary, and the practical constraints may be fewer than the ones where physical experiments are involved.

Some of the instance where such computer modelling is very useful are

- (i) simulation studies,
- (ii) engineering design,
- (iii) energy resources,
- (iv) medical and basic research.

Here we consider the application of these parallel processors in the study of energy resources and basic research.

(a) Energy Resources (Butschner, 1983)

The purpose of reflection seismology is the investigation of unknown subsurface structure like oil resources. Seismic waves are emitted by an energy source and the reflected signals are recorded by an array of receivers. Seismic migration procedures are based on the governing wave equation in order to locate the origins of the reflected signals more correctly. One can also state this as 'inverse scattering' problem.

To study these signals the elastic wave equations are used.

Neglecting shear waves one gets the acoustic wave equations

$$\left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2} \right) P = \frac{1}{v^2} \cdot \frac{\partial^2}{\partial t^2} P$$

where $P = P(x, y, z, t)$ stands for vertical pressure and $v = v(x, y, z)$ for acoustic velocity of the medium. Since the signals are forced to travel in the upward direction, the migration procedure consists of recursive downward continuation of the recorded pressure to a desired depth z . The parabolic one way equation is given (in the space-frequency domain) by

$$\frac{\partial}{\partial z} P = i \sqrt{\frac{w^2}{v^2} + \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}} P \quad (*)$$

where w is the frequency parameter. For actual applications the formal square root operator has to be represented by a Taylor series and for a computer implementation the differential operator has to be approximated by finite differences.

The forced decoupling of (*) gives a simplified system of equations

$$\frac{\partial}{\partial z} P = (O(x) + O(y)) P$$

In the space-time domain, by using approximation to the square root, the finite difference equation for $O(x)$ application is given by

$$T_1 \vec{P}_{t-\Delta t}^{z+\Delta z} = T_2 (\vec{P}_t^{z+\Delta z} - \vec{P}_t^z) + T_3 (\vec{P}_{t+\Delta t}^z - \vec{P}_{t-\Delta t}^z) + T_1 \vec{P}_{t+\Delta t}^z$$

$$\vec{P}_t^z = (P_t^z(x_1), \dots, P_t^z(x_n))^t$$

where the T_i are tridiagonal matrices. The time taken to compute a scalar solution when $n = 512$ has been found to be $1800 \mu s$ but using cyclic reduction method on 2 pipe CYBER 205 this can be reduced to $383 \mu s$. Using a multiple cyclic reduction method with $M = 64$ the time taken can be reduced to $201 \mu s$. One can see an order of time difference in the computational speeds.

A recent 3D-migration example from Prakla Seismes has (630, 450) gridpoints in (x, y) . the number of recorded echotime samples for node point is 1250. This results in a data volume of 350 million real numbers. About 5×10^{12} floating point operations have to be performed with a user CPU of 20 hours. Such seismic data processing requires considerable CPU power and I/O capabilities.

(b) Basic Research (Helmbrecht and Zabolitzky, 1983)

Here we consider the solution of the Schrödinger equation by Green-functions using Monte-Carlo methods. This technique of solution is applied to few body systems. Consider a collection of N He_4 atoms. The Hamiltonian of the system is given by

$$H = \sum_{i=1}^N \left(-\frac{\hbar^2}{2m} \nabla_i^2 \right) + \sum_{i < j} v(|r_i - r_j|).$$

The ground state energy E_0 is given by

$$H \psi(r_1, \dots, r_n) = E_0 \psi(r_1, \dots, r_n). \quad - (1)$$

The wave function is assumed to vanish for the atoms approaching infinity. Equation (1) is transformed by use of a Green's function operator of second order diff. equations into the integral equation:

$$\psi(R) = E_t \int G(R, R_1) \psi(R_1) dR_1 \quad - (2)$$

where $R = (r_1, \dots, r_n)$.

The full Green function is defined by

$$\left(\frac{-\hbar^2}{2m} \nabla_i^2 + v(R_1) \right) G(R, R_1) = \delta(R - R_1).$$

For a non-degenerate ground state, the iteration sequence

$$\phi^{(n+1)}(R) = E_t \int G(R, R_1) \phi^{(n)}(R_1) dR_1$$

will converge to the ground state wave equation for an arbitrary trial energy $E_t > 0$. The ground state energy is given by

$$E_0 = \lim_{n \rightarrow \infty} E_t \frac{\int \phi^{(n)}(R) dR}{\int \phi^{(n+1)}(R) dR};$$

Udo Helmbrecht et al have found that by using interpolation of functions for $N = 500$, the scalar mode on CYBER 205 took 32.00 milliseconds and on vector mode it took 1.49 milliseconds. Again vectorization of the problem reduces the computational time by a very significant extent.

Pipelined and Parallel Processors2.0 Introduction

In this chapter we consider the influence of "concurrency" on computer architecture, which has increased the computational speed to several million floating-point operations/sec. "Pipelining" and "Parallelism" are the two techniques, which have been employed by many of these Supercomputers. We also give a brief description of the computers which have employed the two techniques to achieve higher computational speed.

2.1 Pipelined Processors

In the past three to four decades computer architects have seen the basic technology change from relays and tubes to large scale integration chips. Regardless of these tremendous changes, there always has been a desire for a computer with super performance. To extend capabilities of computers, computer architects have been using several techniques that can be broadly classified under the term "concurrent operation". The two well-recognized techniques which come under this broad category are "parallelism" and "pipelining".

In "parallelism" concurrency is achieved by replicating the hardware structure, whereas "pipelining" takes the approach of assembly-line techniques. Thus, in "pipelining" a particular task is divided into subtasks and each subtask is allocated with a different hardware structure like "array structure". These subtasks are called 'stages'. The data or instruction flows through the different stages just as liquid flows in a physical pipeline. As a particular item flows

through the pipelining it will occupy a stage at a time and then another stage further down the pipeline. As time goes on, a stage vacated by one item is taken over by the one immediately following it. This simultaneous use of many different stages by different items at any given instant is called "overlap".

We now consider an example to illustrate how different architectures can increase the speed of execution of a task, for example performing arithmetic operations (Hockney and Jesshope, 1981). Consider the execution of a floating point addition in serial, pipelined and array architectures. Addition of floating point quantities requires four operations: comparing exponents, shifting, adding and renormalizing. The whole task of addition can be divided into four stages, each stage performing the operations mentioned above. Let x and y be two vectors of floating-point numbers,

$$x = (x_i) , \quad y = (y_i) \quad i = 1, 2, \dots, n$$

$$x_i = e \times 2^p \quad , \quad y_i = f \times 2^q$$

The sum vector z is given by

$$z = (z_i) = (x_i + y_i) , \quad i = 1, 2, \dots, n$$

$$x_i \quad y_i$$

The sequential calculation of the elements of the sum is illustrated in Figure 2.1(a), where a time axis runs from top to bottom.

Figure 2.1(a)

It is clear from the illustration that there is a potential of a four-fold increase in the speed in the pipelined process as compared to the serial process. And for array processors we have all the results at the same time.

There are two types of pipelining: static and dynamic. The above illustration of floating point addition is one of the classical examples of static pipelining. Each evaluation of the basic function is independent of previous evaluations. Every evaluation goes through the same subfunctions which are closely related. And the time to compute each subfunction is almost the same. Dynamic pipelining is nothing but the idea of overlapping. There may be dependencies between the evaluations which may require different subfunctions; the time for a stage may not be a constant; and subfunctions may be distinct in their purpose. One of the simplest examples of dynamic pipelining is the computer where all I/O (input/output) is handled by one processor and all computations by another. The communication between the processors is through a common memory module. Here the operations are less synchronized and the partitioning of the basic function is dynamically changing, so it is hard to predict the time per stage in advance.

Pipelines can be classified according to their capabilities: (Ramamoorthy and Li, 1977) a unifunctional and multifunctional. Unifunctional pipelines are those which perform the same kind of function evaluation on every set of data given to it without any variation. Multifunctional pipelines are capable of doing several kinds of functional evaluations.

2.2 Vector Computers

Program loops occur very frequently in scientific processing. In such cases the execution speed can be greatly increased by providing efficient hardware and software designs to handle these loops. Consider the addition of two vectors A and B. The typical loop for this is

```
For  I = 1  to  N  Do
    A(I)  =  A(I)  +  B(I)
end .
```

For most computers, the equivalent code in machine language is a sequence of instructions, that reads single elements of arrays A and B, adds them, stores the result in the array A, and then increments the loop control variable. These steps are repeated until the value of the variable reaches the limit. This repetition of instructions can be exploited by using instructions that read a series of elements of arrays A and B and another adds the elements, and another stores the series of sums to the A array. Such a series of numbers is called a VECTOR. The application of arithmetic and logical operations on these vectors is called the VECTOR PROCESSING. By using the idea of vector processing one can avoid overheads associated with the loop variable maintenance.

2.3 Pipelined Computers

Here we describe the salient features of the existing pipelined computers. The computers, which make use of pipelined aspects for increasing speed and are trend setters in their own ways, are CRAY 1, CDC CYBER 205, APS AP-120B and VP 100/200.

CRAY 1: The architecture of the CRAY 1 is shown in Figure 2.3(a) (Johnson, 1978, Hockney and Jesshope, 1981). It is both a scalar and a vector processor with separate registers and instructions for both applications. It has eight scalar registers (called S-registers) each having a single element. The eight vector registers (called V-registers) each have 64 elements. The operands lie in these registers thus reducing memory conflicts. The contents of vector length (VL) register determine the number of operations performed by the vector instruction. Eight 24-bit A registers are used as address registers for memory references and as index registers. Each of the A and S registers are supported by 64 rapid-access temporary registers called B and T registers. The memory consists of 1024-bit large scale integration (LSI) chips. The maximum size of the main memory on the CRAY 1 is 1-million, 64-bit words. It is divided into 16 memory banks that may operate concurrently. The clock period of the machine is 12.5 ns and the access and cycle-time of the memory is 50 ns, so a memory bank is busy for four clock periods after a request is accepted. Further reference cannot be made to the same bank while it is busy with a request. But references to other memory banks may be made at the interval of one clock period. Thus the main memory can deliver data at the rate of one 64-bit word from each of the 16 memory banks every 50 ns. The data bus between the registers and the main memory is only 64 bits wide, and so the maximum rate at which data is transferred is one 64-bit word per clock period, if the words are drawn from different memory banks. The minimum rate is one word per four clock periods when the words are drawn from the same memory bank. Successive memory addresses are allocated to successive

Figure 2.3(a)

The Architecture of CRAY 1

memory banks (this is called interleaving of the memory) so that the maximum rate can be maintained.

CRAY 1 has twelve specialized functional units all pipelined, in the central processing unit (CPU) to handle the arithmetic, logical and shift operations. Six of these specialized units are used strictly by the instruction processor for address calculation, scalar integer addition, logical and shifting operations and for scanning of bit vectors. Three of the remaining functional units are used exclusively for vector operations on integer or logical data, and the rest for performing floating point operations for both vector and scalar instructions.

The CRAY 1 has other unique features which enhances the performance of vector operations. The first is that two vector instructions that use distinct vector registers may execute nearly simultaneously. For example, the following instructions can be executed simultaneously.

$$v_2 \leftarrow v_0 + v_1$$

$$v_3 \leftarrow v_4 \otimes v_5$$

Secondly CHAINING can be used. When the result register for one vector instruction is the same as one of the input registers for an instruction which uses separate functional units, the hardware will start the second vector operation during the clock cycle when the first result is just leaving its functional unit. This is done by supplying a copy of the result to the second functional unit. This cycle is then repeated for every clock cycle thereafter. By proper programming several different vector operations can be 'chained' together in this way.

CDC CYBER 205: The main units are shown in Figure 2.3(b) (Hockney and Jesshope, 1981). The total memory is about 4 M words, which is divided into four 1 M word sections. And each 1 M word section is organised into 16 memory stacks. Each 1 M word section is connected to the memory interface by a 512-bit wide data path. This width is known as a 'superword' or 'sword'. This is the main unit of access to memory. The memory has a cycle and access time of 80 ns and the unit clock period is 20 ns. Hence each data path can read or write at the maximum rate of one sword per clock period which is about five times the access rate of the CRAY 1. Each stack of 1 M word has a 32-bit stack data path to the memory interface. Successive addresses are taken from successive stacks. Each stack is divided into eight independent banks, each holding 16k - 39 bit halfwords. The banks can share a single stack data path because only the last clock cycle of a memory access uses it. The memory interface unit organises memory requests at each 20 ns clock interval into swords, words, halfwords and then delivers or assembles this data via 128-bit wide paths to scalar and vector sections. The rest of the computer is communicated with through three read paths and two write paths and the memory interface unit has one sword buffer associated with each path. Each I/O section is connected to the memory interface by two 128-bit wide paths. Thus there are ten 128-bit paths connecting a memory interface unit to each vector, scalar and I/O section. Hence the maximum total transfer rate is about 1000 M words/sec. All data paths to the vector section pass through the scalar section, where single error correction and double error correction checking and priority determination for memory requests takes place. The scalar

section also contains the instructions issued to the pipeline. The instructions are decoded into scalar and vector instructions and the decoded vector instructions are then sent to the vector unit. The result from any unit of the scalar section can be passed on to any other unit by the process called shortstopping, which eliminates the time required to write results to a register and retrieve them for the next operation. Access to main memory is controlled by a load/store unit which acts as a pipeline which can accept one read from memory every clock period or write to memory every two clock periods. A randomly accessed word can be read from memory and loaded onto the registers in 300 ns provided the memory is not busy. If it is, then 80 ns is required.

On the other hand, all vector operations are main memory to memory operations since there are no vector registers. This increases the vector start up time. The vector unit consists of four floating point pipelines. Each of which consists of five pipelined functional units for addition, multiplication, shifting delay and logical operations. Each of these units is attached to a data interchange unit by three 128-bit data paths.

The addition operation in this computer is divided into seven stages. The backward connection is provided to allow the addition of the unnormalized portion of one element of a vector operation to the next element of the vector.

In this architecture, chaining is accomplished in a more restrictive sense. Since different units can exchange data between themselves

Figure 2.3(b)

Architectural Block Diagram of

CDC CYBER 205

through the data exchange unit, linking operations like [vector + vector $\times$ scalar], [(vector + scalar) $\times$ vector] can be chained.

FPS AP-120B / FPS 164 : In the late 1960's the high cost of super computers lead to the design intruction of floating point units which could be attached to minicomputers and which were much cheaper but still very fast. Such a unit was first marketed by Floating Point Systems in 1976 as the AP-120B (Charlesworth, 1981, GMU Technology Series, 1983). This machine was superseded by the FPS 164 which is very similar to AP-120B, but with some improvements like a 64-bit floating point arithmetics unit and a Cache memory.

The primary storage unit for data is main memory. It is also the secondary storage unit for instruction words. The maximum size of this memory is 1.5 M words of cycle time 500 ns. Memory mapping registers have also been added to accommodate multiple users. In addition to this, there is also an auxilliary memory which can be used by algorithms with high memory bandwidth requirements. The size of this memory ranges from 8000 to 32000 words. The X and Y data register units (each containing 32 38-bit registers) (Figure 2.3(c)) provide temporary storage for intermediate results and so reduce the traffic in and out of the main memory. For every CPU cycle one register location can be read and another written from both units which means four simultaneous accesses possible. The address adder unit is responsible for addition, subtraction and logical operations on memory addresses, loop counters and integer data contained in the address registers. The width of the address arithmetic is 32 bits. The 64 address registers, like index registers in conventional computers, provide space for the

separate hardware. Each stage does a portion of work in say, one clock period. At the end of each of clock period results are passed on to the next stage and results are accepted from the previous stage.

The CRAY-like machine has six major components:

1. Main Memory
2. Scalar Processor
3. Vector Controller
4. Vector Memory Address Generator and Memory Controller
5. Local Memory
6. Arithmetic Pipelining.

The main memory is similar to the main memories of conventional computers. Instructions and data in both vector and non-vector form resides in the memory. But the memory organization is heavily interleaved and designed to minimize the access time for vector operands. The scalar processor is responsible for non-vector operations. Once the vector instruction has been fetched and recognized, a separate vector controller takes over. The main functions of this unit are decoding the vector instruction, generating operand address parameters, allocating and setting up vector memory address generator and the arithmetic pipeline itself and monitoring the execution of vector instructions. The vector memory address generator and controller takes the address parameters and translates them into a series of main memory access requests and also makes sure that the arithmetic pipelining has the data it needs when it needs it. Finally the arithmetic pipeline is the hardware where the arithmetic functions associated with the vector instructions that are actually executed.

calculations needed to address array elements. The instruction memory keeps code fetches separate from data fetches; this enables the execution control to execute the two operations simultaneously without mutual interference. The branch control allows the processor to decide which instruction to execute next in parallel with completion of the current instructions.

The functional units are all pipelined. Most of the functional units like multiplier, main memory read pipeline have two or three internal stages. A new operation can be initiated every 167 ns .

Unlike the CRAY 1 computer, the scalar hardware is also used for computations and software loops using scalar operations used to program vector operations.

FACOM Vector Processor System: VP 100 / VP 200 : These models are manufactured by Fujitsu - Japan. Both the VP 100 and VP 200 employ pipeline architecture with multiple pipeline units which can operate concurrently (Kamiya, 1983).

The VP 200 (Figure 2.3(d)) has a main memory of 256 M bytes with a 256-way interleave and the VP 100 has a main memory of 128 M bytes with 128 interleave. There are separate scalar and vector units. The scalar unit which is responsible for fetching and decoding of instructions is provided with 16 general purpose registers, 8 floating point registers and 64 K bytes of Cache memory. The vector unit consists of six functional pipeline units, vector registers (64 K bytes capacity) and 256-vector mask registers. The pipelines for arithmetic operations consist of add/logical pipe, multiply pipe, divide pipe and any two of

Figure 2.3(d)

Major Components of
FACOM Vector Processors

these pipes can operate concurrently. There is also a mask pipe and two load/store pipes. The two load/store pipes are responsible for data transfer between the main storage and the vector registers. The vector operations are done on operands in vector registers (just as in CRAY 1) and operate with a band width of 32 bytes/15 ns. The mask pipe performs logical operations associated with mask vectors (that is, bit strings). The vector processor system also provides vector editing functions: compress/expand and indirect addressing. The processor also provides with dynamically reconfigurable vector registers. The length of the vector registers for the currently executed vector instruction can be altered by an instruction in the program.

The vector unit and the scalar unit can work concurrently and the vector operands can be chained as in the CRAY 1.

2.4 Processor Arrays

Another way of increasing the speed of computation is by replicating the processors (recall Figure 2.1(a)). Here we have N processors which are under the control of a common control unit. The control unit is responsible for decoding and transmitting the instructions and data to the processors. Hence, at any given instant of time, we have the processors doing the same operation on similar operands. The main reasons for this replication of processor elements are

- (i) it is cheaper to manufacture N processors than N computers,
- (ii) full utilization of communication bandwidth.

Here we consider two processor array systems: the BSP (Burroughs Scientific Processor) and ICL DAP (Distributed Array Processor). The

BSP is a miniversion of ILLIAC IV, and has been discontinued by Burroughs, but it is a well-designed array computer worth citing here. The ICL DAP computer is a design which balances computational power with inexpensive technology.

THE BSP: (Hwang, 1984; Hockney and Jesshope, 1981)

Figure 2.4(a)

The BSP is not a stand-alone computer. It is attached to a system manager as shown in Figure 2.4(a) — which is a host machine, usually a B-7800/B-7700 Burroughs processor. This frees the BSP from routine management and I/O functions. There are two data highways between the BSP and the system manager. The slow highway makes a direct contact between I/O units and the control unit of BSP. The fast data path is between I/O processor and the file memory controller.

The major components (Ref: Figure 2.4(b)) of the BSP are the control processor, the parallel processors, a file memory, parallel-memory models and two alignment networks (one for input data and the

Figure 2.4(b)

Major Components of BSP

other for output data). The control processor has 256 K words of memory where scalar and vector instructions and scalar data are stored and this processor provides supervisory interface to the system manager. There is a data pathway to the file memory which can operate at the full 12.5 M words/sec band width. The scalar processor processes all operating system and user program instructions stored in control processor memory. It has 16 registers and is clocked with a cycle of 80 ns. The scalar processor does some serial and scalar portions of user programs, but certain scalar instructions and all vector instructions are passed onto the vector processor. It is the responsibility of the scalar processor to check for vector hazards and to insert various data fields into vector instructions. The parallel processor or vector processor then validates these and transforms them into appropriate instructions. The control and maintenance unit is an interface between the system manager and different control processors. The parallel processors perform vector computations with a clock period of 160 ns. The memory-to-memory data operation forms a five stage pipeline. Data for the array operations is stored in 17 memory modules and is transferred to and from this memory through output and input alignment networks. Each memory module may contain from 32 K to 512 K words with 160 ns cycle time. Using a prime number (17) of memory modules assists access to rows, columns or diagonals without memory bank conflicts as long as the skip distance or stride is not a multiple of 17. The arithmetic element units operate at a clock frequency MHz and all of them at any instant of time will be executing the same instruction on different data sets. The file memory is a high speed secondary storage device loaded by the

system manager with BSP tasks and task files, and may also be used to store output files produced during the execution of the BSP program. A major advantage claimed by the BSP is that since file memory uses semiconductor memory, the reliability of the file memory is much greater than that of conventional devices like discs.

ICL DAP: The ICL DAP is a parallel machine using single instruction multiple data type parallelism. Just like the BSP, this machine also is a front-end, back-end kind of system. The host can be DAP 2900 system. The major components of the DAP system are as shown in Figure 2.4(c). The interface with the host system is provided by the DAP access control and a column highway which has one bit for each column of a DAP array. This column highway also provides a path between the master control unit registers and the rows of a DAP array. (The DAP array is a 64×64 matrix of slave processors.) Finally the highway also connects master control unit to the DAP instructions store. The row highway connects DAP array to master control unit (MCU) registers in the orthogonal direction. This highway is used exclusively for data transfers.

The DAP array is a 64×64 matrix of processing elements. The array is connected two dimensionally. Each processing element has four neighbours, and can communicate with each of them. Communication with edge processing elements takes a long time. The instruction may command periodic connection between rows and columns of the processor array. Each processor element consists of bit-serial units. The A register provides control (programmable) over the action of the processing element. The A register also has a latch input for rapid

combination of control masks. The Q register is the accumulator and the C-register is the carry register. The ICL DAP also has vector transmitting facilities which help in the replication of the vectors.

Figure 2.4(c)

Architecture of ICL DAP

CHAPTER 3

Band and Envelope Methods for
Sparse Symmetric Positive Definite Systems3.0 Introduction

In this chapter we consider the CRAY-like machine architecture and a few techniques for writing algorithms for such machines. We examine the basic numeric algorithm used to solve symmetric positive definite matrix problems known as Cholesky's method and several ways in which the computation can be performed. Though the methods are mathematically equivalent, they differ in the order in which the numbers are computed and used and are important with respect to the implementation in CRAY-like computers. When Cholesky's method is applied to sparse matrix, it generally suffers some fill-in. For some permutation matrix P , PAP^T will have the nonzeros near the main diagonal, so that Cholesky's factorization can be applied. We consider the simplest methods for solving sparse systems, the band scheme and the envelope or profile method, where in the RCM ordering method is used.

3.1 CRAY-like Processor

In Chapter 2, we considered a brief description of existing pipeline processors. Here we consider the basic features that are common to most of the pipeline processors and are helpful in implementing linear algebra algorithms. We consider an architecture that is CRAY-like in structure, where vector operations are performed on vector registers. Thus the hypothetical machine we are considering here typically provides for 'simultaneous' execution of a number of elementwise operations through pipelining. Pipelining takes the approach of breaking the function to be performed into smaller stages and each stage is allocated with a

separate hardware. Each stage does a portion of work in say, one clock period. At the end of each of clock period results are passed on to the next stage and results are accepted from the previous stage.

The CRAY-like machine has six major components:

1. Main Memory
2. Scalar Processor
3. Vector Controller
4. Vector Memory Address Generator and Memory Controller
5. Local Memory
6. Arithmetic Pipelining.

The main memory is similar to the main memories of conventional computers. Instructions and data in both vector and non-vector form resides in the memory. But the memory organization is heavily interleaved and designed to minimize the access time for vector operands. The scalar processor is responsible for non-vector operations. Once the vector instruction has been fetched and recognized, a separate vector controller takes over. The main functions of this unit are decoding the vector instruction, generating operand address parameters, allocating and setting up vector memory address generator and the arithmetic pipeline itself and monitoring the execution of vector instructions. The vector memory address generator and controller takes the address parameters and translates them into a series of main memory access requests and also makes sure that the arithmetic pipelining has the data it needs when it needs it. Finally the arithmetic pipeline is the hardware where the arithmetic functions associated with the vector instructions that are actually executed.

CRAY-like machine consists of vector registers and scalar registers where the operands reside. This reduces the memory to memory operations and also increases the execution rate of the program. Another feature that is used to achieve high rate of execution is chaining. Chaining is a technique wherein the output of register of one vector instruction is the same as input register for the next vector instruction. If the instructions use separate functional units, the hardware will start the second vector operation during the clock period and a copy of the result from the first operation is forwarded directly to the second unit when the result from the first operation is about to leave its functional unit. In CRAY-like machine we are considering, one can chain different arithmetic operations and vector loads from memory to register with other arithmetic operations.

In summary, the CRAY-like machine relies on a number of techniques to enhance its performance: fast cycle time, vector instructions to reduce the number of instructions interpreted, pipelining to utilize functional units chaining to overlap functional unit execution and overlapping to execute more than one independent vector instruction concurrently.

Now we consider programming this CRAY-like vector computer.

3.2 Matrix-Vector Multiplication

Consider the multiplication of a $m \times n$ matrix A by an n -vector x . In matrix notation this is written as

$$y = Ax \quad . \quad \quad \quad - (1)$$

The general algorithms for this are given by

```

Algorithm:      For i = 1 to m
3.2.1          For j = 1 to n
                  yi = yi + aijxj
                end;
            end;

```

```

Algorithm:      For j = 1 to n
3.2.2          For i = 1 to m
                  yi = yi + aijxj
                end;
            end;

```

Algorithm 3.2.1. uses an inner product approach, where each element of y is compiled using one row of A and all of x . The references to A are made by accessing the row entries. In a FORTRAN environment, the matrix entries are stored columnwise, and as the processor time is much less than the memory cycle time, there may exist a memory bank conflict. Algorithm 3.2.2 uses a column scaling method. In a column scaling method we add m vectors each of which is the product of a column of A and an element of x . More precisely,

$$\begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix} = x_1 \begin{bmatrix} a_{11} \\ \vdots \\ a_{m1} \end{bmatrix} + \dots + x_n \begin{bmatrix} a_{1n} \\ \vdots \\ a_{mn} \end{bmatrix}$$

The major advantage of this method is that y can reside in a

register until the entire operation is completed. However when $m \ll n$ the rows are much longer than the columns and then Algorithm 3.2.1 seems to be more appropriate (Dongarra et al, 1984).

3.3 Matrix Multiplication

Consider the multiplication of two $(n \times n)$ matrices A and B given by

$$C = AB \quad - (1)$$

The elements C_{ij} of the product matrix are given by

$$C_{ij} = \sum_{k=1}^n a_{ik} b_{kj} \quad - (2)$$

We can compute the matrix C using three methods, inner-product method, middle-product method and the outer-product method (Hockney and Jesshope, 1981, Kogge, 1981).

3.3.1 Inner-product Method:

The direct translation of equation (2) to algorithmic form is given by

```

Algorithm:      For i = 1 to N ,
3.3.1          For j = 1 to N ,
                  For k = 1 to N ,
                       $C_{ij} = C_{ij} + a_{ik} b_{kj}$       - (*)
                  end;
            end;
    end;
```

The assignment statement (*) in the algorithm is the inner product of i^{th} row of A and j^{th} column of B. This can be evaluated either sequentially or by a cascade sum method. We note here that Algorithm 3.3.1 involves n^2 inner products. The execution rate can be increased by calculating n of these inner products at a time or n^2 at a time instead of one at a time.

3.3.2 Middle-product Method:

By changing the order of the loops and by bringing the loop over rows i to the innermost position, we get the middle-product method.

```

Algorithm:      For j = 1 to N ,
3.3.2          For k = 1 to N ,
                  For i = 1 to N ,
                       $c_{ij} = c_{ij} + a_{ik} b_{kj}$     — (**)
                  end;
              end;
          end;

```

Every term in the loop over i can be computed in the parallel fashion. Here n inner products are computed at a time. In CRAY 1-like computers, super vector performance has been obtained using this method.

3.3.3 Outer-product Method:

This method is obtained by moving the loop over k to the outside.

```

Algorithm:      For k = 1 to n ,
3.3.3          For i = 1 to n ,
                For j = 1 to n ,
                     $c_{ij} = c_{ij} + a_{ik} b_{kj}$ 
                end;
            end;
        end;

```

The terms of the inner product can be obtained in parallel for all n^2 elements of C . This method is found to be suitable for a processor array that has the same dimension as the matrices.

3.4 Multiplication of Matrices by Diagonals

In previous cases we assumed that the matrices were dense. However if we have matrices where only certain diagonals are nonzero, direct methods to multiply such matrices will certainly be inefficient. An efficient vectorization of such problems suggested by Madsen (1976) uses the occurrence of natural vectors — the diagonals. The diagonals may be stored as vectors but since the i^{th} diagonal consists of only $n - i$ elements, these vectors will have different lengths.

Consider the simplest case of matrices with one diagonal as shown in the figure below.

(The main diagonal is taken as zero and lower and upper diagonals are counted away from the main diagonal.)

Diagonal vector representation:

The result of this product is the matrix with only one diagonal whose index is simply the sum of the indices two input diagonals. Further the entries are simple element-by-element multiples of sequential parts of each input. The generalization of this idea for matrices with more than one diagonal is quite straight forward, since we can write such matrices as sums of matrices each of which has only one diagonal.

3.5 Recurrence

Definition: A recurrence is a sequence of evaluations in which the value of the latest term depends on one or more previously evaluated terms (Hoekney and Jesshope, 1981). The recurrence is said to have order k if it depends on k previously evaluated terms.

Example 1: Consider the inner product of two vectors

$x = (x_1, \dots, x_n)^T$ and $y = (y_1, \dots, y_n)^T$. This can be written as a linear recurrence problem of order 1 with scalar product.

$$z_0 = 0$$

— (1)

$$z_k = z_{k-1} + x_k y_k ; \quad 1 \leq k \leq n$$

It appears that only one element can be evaluated at a time and that there is no scope for parallel evaluation, but by rephrasing the problem it is possible to evaluate recurrence in parallel.

This can be done as follows. The equations (1) can be written as

$$z_k = z_{k-1} + d_k \quad \text{where} \quad d_k = x_k y_k$$

FIGURE 3.5(a)

Figure 3.5(a) gives the routing diagram for the "cascade partial sum method" for $n = 8$. In this method, first a set of n accumulators are loaded with the elements to be summed. At level one, a copy of the accumulators is shifted one place to the right and added to the unshifted accumulators to form the sum of the data in adjacent pairs. At the next level the process is repeated with a shift of two places. In

```

Algorithm:      For k = 1 to n ,
3.7.2           $l_{kk} = (a_{kk})^{1/2}$ ;

                For j = k + 1 to n ,
                     $l_{jk} = a_{jk} / l_{kk}$  ;
                For i = k + 1 to n ,
                     $a_{ij} = (a_{ij} - a_{ik} a_{jk} / a_{kk})$  ;
                end;
            end;
        end;
```

Once the matrix is factorized the problem of solving $Ax = b$ reduces to solving the two triangular systems $Ly = b$ and $L^T x = y$.

3.8 Triangular Systems

Consider solving $L^T x = y$ (3.8.1) where L^T is a non-singular upper triangular matrix of order n . The last equation in this system is

$$\begin{aligned}
 l_{nn} x_n &= y_n \\
 x_n &= l_{nn}^{-1} y_n
 \end{aligned}
 \tag{3.8.2}$$

Substituting (2) in (1) we get

$$\begin{aligned} z_j &= a_j a_{j-1} z_{j-2} + a_j d_{j-1} + d_j \\ &= a_j^{(1)} z_{j-2} + d_j^{(1)} \end{aligned}$$

where

$$a_j^{(1)} = a_j a_{j-1} \quad \text{and} \quad d_j^{(1)} = a_j d_{j-1} + d_j \quad . \quad (3.6.4)$$

Repeated application of the above process may be summarised by the reduced equations for level ℓ , as

$$z_j = a_j^{(\ell)} z_{j-2^{\ell}} + d_j^{(\ell)} \quad \begin{array}{l} \ell = 1, \dots, r \quad \text{where} \quad r = \log_2 n \quad \text{and} \quad n = 2^r \\ j = 1, \dots, n \end{array} \quad (3.6.5a)$$

where

$$a_j^{(\ell)} = a_j^{(\ell-1)} a_{j-2^{\ell-1}}^{(\ell-1)} \quad (3.6.5b)$$

$$d_j^{(\ell)} = a_j^{(\ell-1)} d_{j-2^{\ell-1}}^{(\ell-1)} + d_j^{(\ell+1)} \quad (3.6.5c)$$

and initially

$$a_j^{(0)} = a_j \quad \text{and} \quad d_j^{(0)} = d_j \quad . \quad (3.6.5d)$$

Taking the values a_j , d_j or z_j as zero for values outside the range $1 \leq j \leq n$, the solution to recurrence (3.6.5a) is given by

$$z_j = d_j^{(\log_2 n)} \quad (3.6.5e)$$

The method is therefore to calculate successively the coefficients $a_j^{(\ell)}$ and $d_j^{(\ell)}$ defined by (3.6.5b) and (3.6.5c) until $d_j^{(\log_2 n)}$ is

found. In equation (3.6.5c) $d_{j-1}^{(\ell-1)}$ and $d_j^{(\ell-1)}$ have known values which were computed at the previous level and $d_j^{(\ell-1)}$ and d_j are distinct. Hence $\{d_j^{(\ell)}; j = 1, \dots, n\}$ may be evaluated by a single vector instruction. The above general recurrence reduces to a cascade sum when $a_j = 1$ for $2 \leq j \leq n$.

The algorithm for cyclic reduction is *

```

Algorithm:      z = d
3.6.1          For l = 1 to r
                z = a * Shift R(z, 2**l(l-1)) + z
                a = a * Shift R(a, 2**l(l-1))
            end;

```

3.7 Linear Equations

Consider a system of linear equations

$$Ax = b \quad \text{--- (1)}$$

where A is an $N \times N$ matrix and x and b are N -vectors.

The matrix A is said to be symmetric if $a_{ij} = a_{ji}$ for $i, j = 1 \dots N$. The symmetric matrix A is said to be positive definite if $x^t Ax > 0$ for all nonzero vectors x .

Theorem 1

If A is an $N \times N$ symmetric positive definite matrix, it has a unique factorization of the form LL^T where L is a lower triangular matrix with positive diagonal entries (George and Liu, 1981).

Proof

The proof is by induction of the order N of matrix A .

The result is trivially true for $N = 1$ (take $L = [\sqrt{a_{11}}]$) matrices since a_{11} is positive. Suppose the assertion is true for matrices of order $N - 1$. Since A is an $N \times N$ symmetric positive definite matrix it can be partitioned into the form

$$A = \begin{bmatrix} s & v^t \\ v & \bar{H} \end{bmatrix}$$

where s is a positive scalar since all the diagonal entries of A are > 0 and $\bar{H}$ is an $(N-1) \times (N-1)$ submatrix. Again A can be written as

$$A = \begin{bmatrix} \sqrt{s} & 0 \\ \frac{v}{\sqrt{s}} & I_{N-1} \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & \bar{H} \end{bmatrix} \begin{bmatrix} \sqrt{s} & v^t/\sqrt{s} \\ 0 & I_{N-1} \end{bmatrix}$$

where $H = \bar{H} - \frac{vv^t}{s}$. For any nonzero $(n-1)$ vector x ,

$$\begin{bmatrix} \frac{-x^t v}{s} \\ x^t \end{bmatrix} \begin{bmatrix} s & v^t \\ v & \bar{H} \end{bmatrix} \begin{bmatrix} \frac{-x^t v}{s} \\ x \end{bmatrix} = x^t \left(\bar{H} - \frac{vv^t}{s} \right) x = x^t H x$$

which implies $x^t H x > 0$. Hence H is also symmetric and is also positive definite. By the induction assumption H has a triangular factorization $L_H L_H^T$ with positive diagonals. Thus A can be written as

$$\begin{aligned}
 A &= \begin{bmatrix} \sqrt{s} & 0 \\ \frac{v}{\sqrt{s}} & I_{N-1} \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & L_H \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & L_H^T \end{bmatrix} \begin{bmatrix} \sqrt{s} & \frac{v^t}{\sqrt{s}} \\ 0 & I_{N-1} \end{bmatrix} \\
 &= \begin{bmatrix} \sqrt{s} & 0 \\ \frac{v}{\sqrt{s}} & L_H \end{bmatrix} \begin{bmatrix} \sqrt{s} & \frac{v^t}{\sqrt{s}} \\ 0 & L_H^T \end{bmatrix} = LL^T.
 \end{aligned}$$

Uniqueness of L :

Suppose A can be factored into $L_1 L_1^T$ where L_1 is again a lower triangular matrix with positive diagonal entries. Then we have

$$L_1 L_1^T = LL^T.$$

Multiplying on the left by L_1^{-1} ,

$$L_1^{-1} L_1 L_1^T = L_1^{-1} LL^T.$$

Again multiplying by $(L^T)^{-1}$ we have

$$(L_1^T) (L^T)^{-1} = L_1^{-1} L.$$

The product on the left hand side is upper triangular and the product on the right hand side is lower triangular. A matrix which is both lower triangular and upper triangular at the same time is a diagonal matrix. Hence we have

$$(L_1^T) (L^T)^{-1} = (L_1^{-1}) L = D \quad - (*)$$

and the diagonal entries of D are all > 0 since the diagonal entries

of L and L_1 are positive. Consider DD^T .

$$\begin{aligned} DD^T &= L_1^{-1} L (L_1^T (L^T)^{-1})^T \\ &= L_1^{-1} L L^{-1} L_1 \\ &= L_1^{-1} I_N L_1 \\ &= I_N. \end{aligned}$$

Since all diagonal entries of D are > 0 , this shows D is the unit matrix.

Then (*) becomes

$$L_1^{-1} L = I.$$

Multiplying on the left by L_1 we get

$$\begin{aligned} L_1 L_1^{-1} L &= L_1 \\ L &= L_1. \end{aligned}$$

Thus we have the uniqueness of factorization for A .

Algorithm for Cholesky's Decomposition:

An effective method of computing this decomposition is by comparing the entries in the matrix equation $A = LL^T$. For $i \geq k$, we get

$$a_{ik} = \sum_{p=1}^k l_{ip} l_{kp}.$$

Rearranging the above equation (Stewart, 1973) we get

$$l_{ik} = (a_{ik} - \sum_{p=1}^{k-1} l_{ip} l_{kp}) / l_{kk}$$

$$l_{kk} = (a_{kk} - \sum_{p=1}^{k-1} l_{kp}^2)^{1/2}$$

These equations can be used to compute the columns of L , one at a time, for $k = 1, 2, \dots, n$, and we can store L in A itself.

Algorithm: (Golub and Loan, 1983)
3.7.1

```

For k = 1 to n ,
    m = 0 ;
    For p = 1 to k - 1 ,
        m = m + akp2 ;
    end;
    akk = (akk - m)1/2 ;
    For i = k + 1 to n ,
        m = 0 ;
        For p = 1 to k - 1 ,
            m = m + aip akp ;
        end;
        aik = (aik - m) / akk
    end;
end;

```

The above algorithm involved inner products even though the columns of L are computed one at a time. The constructive proof of Theorem 1 suggests the outer product forms of the algorithm. The access to components of A during factorization can be depicted as

Algorithm: For $k = 1$ to n ,
3.7.2

$$l_{kk} = (a_{kk})^{1/2};$$

For $j = k + 1$ to n ,

$$l_{jk} = a_{jk} / l_{kk};$$

For $i = k + 1$ to n ,

$$a_{ij} = (a_{ij} - a_{ik} a_{jk} / a_{kk});$$

end;

end;

end;

Once the matrix is factorized the problem of solving $Ax = b$ reduces to solving the two triangular systems $Ly = b$ and $L^T x = y$.

3.8 Triangular Systems

Consider solving $L^T x = y$ (3.8.1) where L^T is a non-singular upper triangular matrix of order n . The last equation in this system is

$$\begin{aligned} l_{nn} x_n &= y_n \\ x_n &= l_{nn}^{-1} y_n \end{aligned} \quad (3.8.2)$$

In general if we know, $x_n, x_{n-1}, \dots, x_{i+1}$, we can solve the i^{th} equation of (3.8.1) which is

$$l_{ii} x_i + l_{i,i+1} x_{i+1} + \dots + l_{in} x_n = y_i \quad - (3.8.3)$$

to get

$$x_i = l_{ii}^{-1} \left(y_i - \sum_{j=i+1}^n l_{ij} x_j \right) \quad - (3.8.4)$$

These define an algorithm for the computation of the solution of (3.8.1) (George and Liu, 1981).

Algorithm: This algorithm computes the solution of the system 3.8.1 (3.8.1) by the method called the back substitution (George and Liu, 1981).

For $i = n$ to 1 ,

$$x_i = l_{ii}^{-1} \left(y_i - \sum_{j=i+1}^n l_{ij} x_j \right);$$

end;

We note here that the references are made by accessing the elements across the rows. In a FORTRAN environment we have columnwise storing of matrix elements. In a CRAY-like processor, the processor cycle is less than the memory cycle time. The above algorithm approach gives rise to memory bank conflicts. This can be avoided by making use exclusively of column operations. The basic operation in this approach is forming a scalar multiple of one vector and storing the result.

Algorithm: For $i = N$ to 1 ,
3.8.2

$$x_i = y_i / l_{ii}$$

$$\begin{bmatrix} y_1 \\ \vdots \\ y_i \\ \vdots \\ y_N \end{bmatrix} = \begin{bmatrix} y_1 \\ \vdots \\ y_i \\ \vdots \\ y_N \end{bmatrix} - x_i \begin{bmatrix} l_{1i} \\ \vdots \\ l_{ii} \\ \vdots \\ l_{Ni} \end{bmatrix}$$

end;

The algorithm (3.8.2) can be applied to lower triangular matrices with slight modifications.

Algorithm: For $i = 1$ to N ,
3.8.3

$$x_i = y_i / l_{ii}$$

$$\begin{bmatrix} y_1 \\ \vdots \\ y_i \\ \vdots \\ y_N \end{bmatrix} = \begin{bmatrix} y_1 \\ \vdots \\ y_i \\ \vdots \\ y_N \end{bmatrix} - x_i \begin{bmatrix} l_{1i} \\ \vdots \\ l_{ii} \\ \vdots \\ l_{Ni} \end{bmatrix}$$

end;

3.9 Sparse systems

Now we consider the case of sparse matrices. The usual direct method of factoring these matrices is the Cholesky's method as described above. However when A is sparse such factorization causes "fill-in" and the matrix L we obtain has more nonzeros than the lower triangular part of A . By reordering the matrix we can reduce the fill-in and hence the storage and computation requirements. Since for any $N \times N$

permutation matrix P , PAP^T is also symmetric and positive definite, we can apply Cholesky's method to the permuted matrix PAP^T . So we can solve an equivalent system

$$PAP^T x = b.$$

The choice of an appropriate P is called the ordering problem and algorithms for finding P are called ordering algorithms.

Now we consider the algorithms for the factorization of the sparse symmetric positive definite systems.

3.9.1 The Band Method

We often come across banded matrices which have a few nonzero diagonals. (Figure 3.9.1(a).) The band

$\beta(A)$ of a matrix $A = (a_{ij})$ is the maximum value of $|i - j|$ such that $a_{ij} \neq 0$, that is

$$\beta(A) = \max_{a_{ij} \neq 0} |i - j|.$$

The implicit nature of the band method is that it allows us to exploit the zeros outside the band, but not those within the band. The band is

filled almost completely during the factorization. Suppose we have matrix A (lower half) stored by columns. The columns of L are developed one at a time using appropriate linear combination of columns of A . When computing the k^{th} column of L , the following modification on the $(k+j)^{\text{th}}$ column of A is required (R.G. Voigt, 1977).

For $j = 1$ to $(n - k)$

$$\begin{bmatrix} a_{k+j, k+j} \\ a_{k+1+j, k+j} \\ \vdots \\ a_{n, k+j} \end{bmatrix} = \begin{bmatrix} a_{k+j, k+j} \\ a_{k+1+j, k+j} \\ \vdots \\ a_{n, k+j} \end{bmatrix} - a_{k+j, k} \begin{bmatrix} a_{k+j, k} \\ a_{k+1+j, k} \\ \vdots \\ a_{n, k} \end{bmatrix}$$

In case of banded matrices there are only β elements below the main diagonal. Hence one can change the limit to $k+\beta$ instead of n in the loops.

Algorithm: For $k = 1$ to $(\beta + 1)$,

3.9.1

$$l_{kk} = (a_{kk})^{1/2};$$

For $j = k + 1$ to $(k + \beta)$

$$l_{jk} = a_{jk} / l_{kk};$$

For $i = k + 1$ to $(k + \beta)$

$$a_{ij} = (a_{ij} - a_{ik} a_{jk} / a_{kk}); \quad (*)$$

end;

end;

end;

In CRAY-like computers, the operands are in the vector registers. The loop (*) can be vectorised. Suppose the column $(a_{k+1,j} \dots a_{\beta,j})$ is available in one of the vector registers. Then loading $(a_{k+1,k} \dots a_{\beta,k})$, multiplying and subtracting can be chained together. But then the arithmetic units will be idle while we are storing the result back to memory and getting the new operands for the next value of j , and moreover the multiplication does not begin with the first element of the vectors. The elements in the register have to be shifted. Another method (Fong and Jordan, 1976) is to reverse the vectors so that the computation begins with the first element.

If the band is too thin, the start up time for the arithmetic pipes will be more. It would be better to switch to scalar mode of operations in such cases.

3.10 The Envelope Method

An extension of the bandwidth method is the envelope or the profile method. The profile method assumes that all nonzero entries of our matrix are within an envelope of varying width about the main diagonal. The most widely used profile reduction ordering algorithm is a variant of the Cuthill-McKee ordering. The Cuthill-McKee method scheme may be regarded as a method which reduces the bandwidth of a matrix by local minimization of the β_i 's. This scheme can be used to minimize $\sum \beta_i$ of a matrix. A. George found that by reversing this ordering, a much superior ordering is obtained in terms of the profile reduction.

The Reverse-Cuthill-McKee Algorithm (RCM) (George and Liu, 1981) for a connected graph $G = (V, E)$ is as follows:

Algorithm: Step 1 - Determine a starting node r and assign
3.10.1

$v_1 \leftarrow r$; (See below for how to make a good
choice of starting node.)

Step 2 - For $i = 1, \dots, N$ find all the unnumbered
neighbours of the node v_i and number them
in increasing order of degree.

Step 3 - The reverse ordering is given by

$v'_1, \dots, v'_N$ where $v'_i = v_{N-i+1}$ for
 $i = 1, \dots, N$.

If the graph is disconnected, one can apply the above algorithm on
each connected component of the graph.

Let $d(v, w)$ be the length of the shortest path joining v, w
in a connected graph G . Then the eccentricity of a node v is
defined by

$$\ell(v) = \max \{d(v, w) \mid w \in V\}$$

and the diameter of G is given by

$$\begin{aligned} \delta(G) &= \max \{\ell(v) \mid v \in V\} \\ &= \max \{(d(v, w)) \mid v, w \in V\}. \end{aligned}$$

A node $v \in V$ is called a peripheral node if its eccentricity $\ell(v)$ is
equal to the diameter $\delta(G)$ of the graph G .

The peripheral nodes are good starting nodes for Step A in the
algorithm, but it is very difficult to find such a node. We have a
heuristic algorithm which finds nodes which have $\ell(r)$ near to $\delta(G)$,

and these nodes are good enough for the starting nodes. These nodes are called pseudo-peripheral nodes.

The key idea in the algorithm to find pseudo-peripheral nodes, is to construct a rooted level structure. Given a node $v \in V$, we consider a partitioning

$$L(v) = \{L_0(v), L_1(v), \dots, L_{\ell(v)}(v)\}$$

of V where $L_0(v) = \{v\}$, $L_1(v) = \text{Adj}(L_0(v))$ and

$$L_i(v) = (\text{Adj}(L_{i-1}(v)) - L_{i-2}(v)) \text{ for } i = 2, 3, \dots, \ell(v)$$

where $\ell(v)$ is the eccentricity of the node v (alternatively

$$L_i(v) = \{u \in V \mid \delta(u, v) = i\}.$$

Algorithm: Step 1 - Choose any $v \in V$, $r \leftarrow v$.

3.10.2

Step 2 - Construct the level structure rooted at r .

$$L(r) = \{L_0(r), L_1(r), \dots, L_{\ell(r)}(r)\}.$$

Step 3 - Choose a node $v \in L_{\ell(r)}(r)$ of minimum degree.

Step 4 - 1. Construct the level structure rooted at v .

$$L'(v) = \{L'_0(v), L'_1(v), \dots, L'_{\ell(v)}(v)\}.$$

2. If $\ell(v) > \ell(r)$ set $r \leftarrow v$ and go to

Step 3. Otherwise return r .

(The node v is a pseudo-peripheral node.)

Once the starting node has been found, the next step involved finding the degree of each unnumbered neighbour of the node. There are many established algorithms which may be used to find the degrees of the nodes and arrange them in increasing order of degrees.

All algorithms for bandwidth reduction are useful primarily for sparse matrices, which are not too large and irregularly patterned. These arise primarily in finite element approaches where the geometry is complicated. Numerical experiments on CRAY-like computers have shown that for $n = 1000$, RCM method gives much better performance over the other methods. For matrices of this order, the general methods may not make any use of effective use of the vector arithmetic.

CHAPTER 4

Minimum Degree Methods4.0 Introduction

Sparse matrices may suffer some fill-in when they are factored, so $L + L^T$ has more non zero entries than A and these appear in positions where zeros appear in A . The objective of many heuristics is to find an ordering or permutation of the rows and columns of A so that the fill-in is small. Some algorithms including the minimum degree algorithm (which is based on the Markowitz Method) require that the factorization be simulated as the ordering proceeds, because the ordering depends on the structure of partially factored matrix. There are three models by which positive definite symmetric factorization can be studied: the Elimination Graph Model; the Reachable Set Model; and the Quotient Graph Model. (George and Liu, 1978, 1981; George and McIntyre, 1978; Duff, 1981.)

4.1 Elimination Graph Model

Consider the proof of Theorem 1 of Chapter 3. The fill-in can be modelled by a sequence of graph transformations on the original graph G_A associated with the matrix A . The elimination graph G_v of a Graph $G = (V, E)$ by $v \in V$ is the graph obtained from G by deleting the node v and its incident edges and then adding edges to the remaining graph so that every pair of nodes in $\text{Adj}(v)$ are connected. The process of symmetric elimination on A can be modelled as a sequence of elimination graphs $G_0, G_1, \dots, G_{N-1}$, where $G_0 = G_A$ and $G_i = (G_{i-1})_{v_i} = (V_i, E_i)$ for $i = 1, \dots, N-1$. Here $V_i = \{v_{i+1}, \dots, v_N\}$ and each $G_i \equiv G_{H_i}$ where H_i is the part of the matrix which remains to be factored after

This implies that the test can be done by simply inspection of the adjacency sets of the nodes in the intersection of R_1 and R_2 .

(Lemma 5.3.6 and 5.3.7, George and Liu, 1981.)

4.5.3 Minimum Degree Algorithm Using Quotient Graph Model:

Algorithm Step 1 - $S = \phi$

4.5.3

$\text{Degree}(v) = |\text{Adj}(v)|$ for $v \in V$.

Step 2 - Select a node w such that $w \in V - S$ and

$\text{Degree}(w) = \min_{v \in V - S} (\text{Degree}(v))$.

Step 3 - Number the nodes in

$Y = \{v \in V - S \mid v \text{ is indistinguishable from } w\}$

next in the ordering.

Step 4 - For $u \in \text{Reach}(w, S) - Y$,

$\text{Degree}(u) = |\text{Reach}(u, S \cup Y)|$

and identify the set $\text{Reach}(w, S) - Y$.

Step 5 - Set $S = S \cup Y$. If $S = V$, then stop.

Else go to Step 2.

Minimum degree algorithm is very widely used in most of engineering problems. For CRAY-like computers the minimum degree algorithms by the quotient graph model may be suitable for two reasons, the storage requirement and the ordering procedure. The storage requirement is much less here and the factorization can be done "in-place". Whereas in the elimination method one is not aware of the storage requirements and this may create a problem for very large problems. Some experiments for finite element model of a geodesic dome with a fine subdivision which is mounted on a rectangular base with a coarse finite element mesh

The Quotient graph of G with respect to P is given by the graph $G/P = (P, \mathcal{E})$ where $\{Y_i, Y_j\} \in \mathcal{E}$ if and only if $Y_i \cap \text{Adj}(Y_j) \neq \emptyset$.

Let S be a subset of the node set V . Define the set

$$C(S) = \{C \subset S \mid G(C) \text{ is a connected component in the subgraph } G(S)\}$$

and the partitioning on V ,

$$\bar{C}(S) = \{\{y\} \mid y \in V - S\} \cup C(S).$$

Let $S_i = \{v_1, \dots, v_{i-1}\}$ for $1 \leq i \leq N$ and $S_0 = \emptyset$. For each i , the subset S_i induces the partitioning $\bar{C}(S_i)$ and the corresponding quotient graph

$$G_{Q_i} = G/\bar{C}(S_i) = (\bar{C}(S_i), E_{Q_i}).$$

Theorem 2

For $y \notin S_i$, $x \in \text{Reach}_G(y, S_i) \iff \{x\} \in \text{Reach}_{G_{Q_i}}(\{y\}, C(S_i))$.

Proof

Consider $x \in \text{Reach}_G(y, S_i)$. If x and y are adjacent in G , so are $\{x\}$ and $\{y\}$ in G_{Q_i} . Otherwise there is a path $y, s_1, \dots, s_t, x$ in G where $\{s_1, \dots, s_t\} \in S_i$. Let C be the component in $G(S_i)$ containing $\{s_1, \dots, s_t\}$. Then we have a path $\{\{y\}, C, \{x\}\}$ in G_{Q_i} , so that $\{x\} \in \text{Reach}_{G_{Q_i}}(\{y\}, C(S_i))$.

Conversely, consider any $\{x\} \in \text{Reach}_{G_{Q_i}}(\{y\}, C(S_i))$. Then there exists a path $\{y\}, C_1, \dots, C_t, \{x\}$ in G_{Q_i} , where each $C_j \in C(S_i)$. If $t = 0$ then x and y are adjacent in G . If $t > 0$ then by

the definition of $C(S_i)$, t cannot be greater than 1. The path is $\{y\}, C, \{x\}$. But $G(C)$ is a connected subgraph. Hence we can obtain a path y to x through $C \in S_i$ in G . Hence $x \in \text{Reach}_G(y, S_i)$ and the theorem is proved.

Thus we see the elimination graph G_i can be obtained from these quotient graphs G_{Q_i}

- (i) by removing the nodes in $C(S_i)$ and their incident edges from the quotient graph G_{Q_i} ;
- (ii) for each $C \in C(S)$, add edges to the quotient graph so that all adjacent nodes of C form a clique in the elimination graph.

4.4 Algorithms for Elimination Graphs

4.4.1 Elimination Model and Reachable Set Model:

Let v_i be the node eliminated from graph G_{i-1} to obtain the elimination graph G_i . The adjacency structure of G_i can be obtained as follows.

Algorithm Step 1 - Determine the adjacent set $\text{Adj}_{G_{i-1}}(v_i)$ in G_{i-1} .
4.4.1

Step 2 - Remove v_i and its adjacency list from the adjacency structure.

Step 3 - For each node $v_j \in \text{Adj}_{G_{i-1}}(v_i)$, merge the subsets $\text{Adj}_{G_{i-1}}(v_j) - \{v_i\}$ and $\text{Adj}_{G_{i-1}}(v_i) - \{v_j\}$ to get the adjacency set of v_j in G_i .

Flexible data structures like adjacency linked lists are suitable for explicit implementation. The major disadvantage with the explicit method is that, it requires significant overhead in storage and execution because of dynamic data structure and the maximum storage needed is not known until the end of the elimination process. The implicit execution by reachable set does not have this disadvantage but the amount of work required to determine reachable sets may be large if $|S_i|$ is large.

4.4.2 Quotient Graph Model:

Let $G = (V, E)$. Consider a subset $U \subset V$ and a node $v \notin U$ such that $G(U \cup \{v\})$ is connected. Consider the quotient graph generated by collapsing the subset $U \cup \{v\}$ to form a new node. The sequence of elimination graphs can be achieved "in-place".

Algorithm Step 1 - Determine the set $\text{Adj}(U \cup \{v\})$.

4.4.2

Step 2 - Use the node v as the representative of the new quotient member $U \cup \{v\}$.

Reset:

$$\text{Adj}(v) \leftarrow \text{Adj}(U \cup \{v\})$$

Step 3 - For $w \in \text{Adj}(U \cup \{v\})$, put

$$\text{Adj}(w) \leftarrow (\text{Adj}(w) - U) \cup \{v\}$$

The quotient graph $G_{Q_{i+1}}$ can be obtained from G_{Q_i} by the execution of the above transformation.

4.5 Minimum Degree Algorithm

The Minimum Degree Algorithm is a heuristic algorithm which is found to be very effective in finding a minimal fill-in ordering for sparse matrices. The key idea is, at each elimination step, to

permute the part of the matrix remaining to be factored so that a column with the fewest non zeros moves into the pivot position. This means that at each step we need to know the structure of the partially factored matrix, i.e., we must implement some model of the factorization process.

4.5.1 Minimum Degree Algorithm Using the Elimination Graph Model:

Algorithm: Let $G_0 = (V, E)$ be an unlabeled graph.
4.5.1

Step 1 - $i = 1$;

Step 2 - In the elimination graph $G_{i-1} = (V_{i-1}, E_{i-1})$,
choose v_i of minimum degree in G_{i-1} .

Step 3 - Form the new elimination graph $G_i = (V_i, E_i)$
by eliminating the node v_i from G_{i-1}
(using Algorithm 4.4.1).

Step 4 - $i = i + 1$. If $i > |V|$ stop . . Otherwise go
to Step 2 .

If there is more than one node with the minimum degree at a particular step, then break the ties arbitrarily. Different tie-breaking strategies give different versions of the minimum degree algorithm.

4.5.2 Minimum Degree Algorithm Using the Reachable Sets:

Algorithm Step 1 - $S = \phi$; Degree(v) = $|Adj(v)|$ for all $v \in V$.
4.5.2

Step 2 - Pick a node $y \in V - S$ whose degree is as small
as possible, i.e.

$$\text{Degree}(Y) = \min_{v \in V - S} (\text{Degree}(v)) .$$

Number the node y next and set

$$T = S \cup \{y\} .$$

Step 3 - $\text{Degree}(u) = |\text{Reach}(u, T)|$ for $u \in V - T$.

Step 4 - If $T = V$ then stop. Else go to Step 2.

As stated before this algorithm can be done "in-place". Step 3 can be modified to

$$\text{Degree}(u) = |\text{Reach}(u, T)|, \text{ for } u \in \text{Reach}(y, S).$$

Since the elimination of y affects only the degrees of the nodes in $\text{Reach}(y, S)$ some more improvements can be made in Step 2. When a node y of minimum degree is found at Step 2, we may find a subset of nodes that may be numbered automatically without carrying out any minimum degree search. Two nodes x and $y \in V - S$ are said to be indistinguishable with respect to elimination if,

$$\text{Reach}(x, S) \cup \{x\} = \text{Reach}(y, S) \cup \{y\}.$$

If we find two such nodes then they can be eliminated together, that is, they can be glued together and treated as a single node.

But identifying indistinguishable nodes is time consuming if we follow the definition. Suppose $G(C_1)$ and $G(C_2)$ are two connected components in the subgraph $G(S)$, where S is the set of eliminated nodes; $C_1, C_2 \in C(S)$. Let $R_1 = \text{Adj}(C_1)$ and $R_2 = \text{Adj}(C_2)$. If $y \in R_1 \cap R_2$ and $\text{Adj}(y) \subset R_1 \cup R_2 \cup C_1 \cup C_2$, then

$$\text{Reach}(y, S) \cup \{y\} = R_1 \cup R_2.$$

Then the set of indistinguishable nodes with respect to S is given by

$$Y = \{y \in R_1 \cap R_2 \mid \text{Adj}(y) \subset R_1 \cup R_2 \cup C_1 \cup C_2\}.$$

This implies that the test can be done by simply inspection of the adjacency sets of the nodes in the intersection of R_1 and R_2 .

(Lemma 5.3.6 and 5.3.7, George and Liu, 1981.)

4.5.3 Minimum Degree Algorithm Using Quotient Graph Model:

Algorithm Step 1 - $S = \phi$

4.5.3

$\text{Degree}(v) = |\text{Adj}(v)|$ for $v \in V$.

Step 2 - Select a node w such that $w \in V - S$ and

$\text{Degree}(w) = \min_{v \in V - S} (\text{Degree}(v))$.

Step 3 - Number the nodes in

$Y = \{v \in V - S \mid v \text{ is indistinguishable from } w\}$
next in the ordering.

Step 4 - For $u \in \text{Reach}(w, S) - Y$,

$\text{Degree}(u) = |\text{Reach}(u, S \cup Y)|$

and identify the set $\text{Reach}(w, S) - Y$.

Step 5 - Set $S = S \cup Y$. If $S = V$, then stop.

Else go to Step 2.

Minimum degree algorithm is very widely used in most of engineering problems. For CRAY-like computers the minimum degree algorithms by the quotient graph model may be suitable for two reasons, the storage requirement and the ordering procedure. The storage requirement is much less here and the factorization can be done "in-place". Whereas in the elimination method one is not aware of the storage requirements and this may create a problem for very large problems. Some experiments for finite element model of a geodesic dome with a fine subdivision which is mounted on a rectangular base with a coarse finite element mesh

Algorithm: Step 1 - Factor the matrix A_{11} into $L_{11} L_{11}^T$.
5.5.1

Step 1 - Solve the triangular system

$$L_{11} W = V \text{ for } W, \text{ where } V \text{ and } W \text{ are } r \times s \text{ matrices.}$$

Step 3 - Compute $A_{22} = \bar{A}_{22} - W^T W$.

Step 4 - Factor the matrix $A_{22} = L_{22} L_{22}^T$.

Now consider the modification matrix $V^T A_{11}^{-1} V$. This can be computed in two different ways

$$(V^T (L_{11}^{-1})^T) (L_{11}^{-1} V) = W^T W$$

or as

$$V^T ((L_{11}^{-1})^T (L_{11}^{-1} V)) = V^T ((L_{11}^{-1})^T W) = V^T \bar{W} \quad (5.1.1)$$

where

$$\bar{W} = (L_{11}^{-1})^T W$$

If V is available then we can avoid storing W if we compute A_{22} as implied by the equation (5.1.1). The columns of $V^T \bar{W}$ can be calculated one at a time and only one vector of length r is required for temporary storage. The algorithm 5.1.1 can be modified as follows.

Algorithm: Step 1 - Factor $A_{11} = L_{11} L_{11}^T$.
5.1.2

Step 2 - For each column v of V

(2.1) Solve $L_{11} w = v$.

(2.2) Solve $L_{11}^T \bar{w} = w$.

(2.3) Set each column of $A_{22} = A_{22*1}$ as

$$A_{22*1} = \bar{A}_{22*1} - V^T \bar{w}$$

Step 3 - Factor $A_{22} = L_{22} L_{22}^T$.

CHAPTER 5

Block Elimination5.0 Introduction

Finite element and finite difference methods are used for solving differential equations that arise in the study of structural analysis, heat flow, etc. The matrix associated with such problems is often symmetric and positive definite. If the matrix A is partitioned into p^2 submatrices A_{ij} ($1 \leq i, j \leq p$) and the x and b matrices are similarly partitioned, then, Cholesky's procedure can be used to solve the partitioned system. Here we consider the block methods for solving sparse linear systems and give two methods for solving finite element matrix problems.

5.1 Factorization of a Block Two By Two Matrix (Bunch, 1976)

Consider a block 2×2 linear system $Ax = b$ where A is a sparse symmetric positive definite matrix:

$$\begin{bmatrix} A_{11} & V \\ V^T & \bar{A}_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

where A_{11} is an $r \times r$ and $\bar{A}_{22}$ is an $s \times s$ submatrix. The block Cholesky's decomposition is given by

$$A = LL^T$$

where

$$L = \begin{bmatrix} L_{11} & 0 \\ W^T & L_{22} \end{bmatrix}$$

where L_{11} , L_{22} are Cholesky factors of A_{11} and $A_{22} = \bar{A}_{22} - V^T A_{11}^{-1} V$ respectively and $W = L_{11}^{-1} V$. Note that A_{11} is positive definite and so can be factored into $A_{11} = L_{11} L_{11}^T$ since it is a principal submatrix of a positive definite matrix A . A_{22} can also be factored into $L_{22} L_{22}^T$ for it is also symmetric and positive definite. Indeed for any vector y of length s , we have

$$\begin{aligned}
 & (-y^T V^T A_{11}^{-1}, y^T) \begin{bmatrix} A_{11} & V \\ V^T & \bar{A}_{22} \end{bmatrix} \begin{bmatrix} -A_{11}^{-1} V y \\ y \end{bmatrix} \\
 &= (-y^T V^T + y^T V^T, -y^T V^T A_{11}^{-1} V + y^T \bar{A}_{22}) \begin{bmatrix} -A_{11}^{-1} V y \\ y \end{bmatrix} \\
 &= -y^T V^T A_{11}^{-1} V y + y^T \bar{A}_{22} y \\
 &= y^T (\bar{A}_{22} - V^T A_{11}^{-1} V) y = y^T A_{22} y.
 \end{aligned}$$

Since $A = \begin{bmatrix} A_{11} & V \\ V^T & \bar{A}_{22} \end{bmatrix}$ is positive definite this shows that

$y^T A_{22} y > 0$ for every vector $y \neq 0$ and so A_{22} is positive definite.

Now

$$A_{22} = \bar{A}_{22} - V^T A_{11}^{-1} V$$

and

$$-V^T A_{11}^{-1} V = V^T (L_{11}^{-1})^T (L_{11}^{-1}) V = W^T W.$$

The computation of L is given by the following algorithm.

Algorithm: Step 1 - Factor the matrix A_{11} into $L_{11} L_{11}^T$.
5.5.1

Step 1 - Solve the triangular system ,

$L_{11} W = V$ for W , where V and W are
 $r \times s$ matrices.

Step 3 - Compute $A_{22} = \bar{A}_{22} - W^T W$.

Step 4 - Factor the matrix $A_{22} = L_{22} L_{22}^T$.

Now consider the modification matrix $V^T A_{11}^{-1} V$. This can be computed in two different ways

$$(V^T (L_{11}^{-1})^T) (L_{11}^{-1} V) = W^T W$$

or as

$$V^T ((L_{11}^{-1})^T (L_{11}^{-1} V)) = V^T ((L_{11}^{-1})^T W) = V^T \bar{W} \quad - (5.1.1)$$

where

$$\bar{W} = (L_{11}^{-1})^T W$$

If V is available then we can avoid storing W if we compute A_{22} as implied by the equation (5.1.1). The columns of $V^T \bar{W}$ can be calculated one at a time and only one vector of length r is required for temporary storage. The algorithm 5.1.1 can be modified as follows.

Algorithm: Step 1 - Factor $A_{11} = L_{11} L_{11}^T$.
5.1.2

Step 2 - For each column v of V

(2.1) Solve $L_{11} w = v$.

(2.2) Solve $L_{11}^T \bar{w} = w$.

(2.3) Set each column of $A_{22} = A_{22+i}$, as

$$A_{22+i} = \bar{A}_{22+i} - V^T \bar{w}$$

Step 3 - Factor $A_{22} = L_{22} L_{22}^T$.

Suppose the given matrix A is partitioned into p^2 submatrices A_{ij} , $(1 \leq i, j \leq p)$. The off-diagonal blocks will not have as simple a form as W and we cannot discard them since the cost of recomputing them would be very expensive. George and Liu (1978, 1981) have suggested a quotient tree partitioning method for A in order that the off-diagonal blocks of A will have a simple form.

5.2 Quotient Tree Partitioning

Let $G = (V, E)$ be a given unlabelled graph and $P = \{Y_1, \dots, Y_p\}$ be a partition on V such that, $V = \bigcup_{k=1}^p Y_k$ and $Y_i \cap Y_j \neq \emptyset$ for $i \neq j$. We define the quotient graph G/P to be the graph $(P, \mathcal{E})$, where $\{Y_i, Y_j\} \in \mathcal{E}$ if and only if $\text{Adj}(Y_i) \cap Y_j \neq \emptyset$.

An ordering α of G is said to be compatible with P , if each member Y_i of P is numbered consecutively by α . An ordering α which is compatible with P induces an ordering on G/P . The quotient graph G/P is said to be a monotonely ordered tree if it is a rooted tree in which the label of each node precedes the label of its father.

Lemma 1

Let the labelled quotient graph of the partitioned matrix $A = [A_{ij}]$, be a monotonely ordered tree. Let $A = LL^T$ be the Cholesky factorization of A and suppose L is partitioned compatibly with A . Then for each non-null submatrix L_{ij} in the lower triangular matrix L , we have $A_{ij} \neq 0$ and for $i > j$

$$L_{ij} = A_{ij}(L_{jj}^{-1})^T.$$

Proof

We prove the result by induction on $m = i + j$.

For $i = j$, it is enough to show that $A_{ii} \neq 0$, since it is the principal submatrix of a positive definite matrix and this settles the case $m = 2$.

For $i > j$, the blocks of L are given by

$$L_{ij}^T = L_{jj}^{-1} \left[A_{ji} - \sum_{k=1}^{j-1} (L_{ik} L_{jk}^T)^T \right].$$

Now we assume that $m > 2$ and that the lemma is true for all i, j with $i + j < m$.

Now we show that it is true for all i, j with $m = i + j$.

For $i = j$, we showed this above.

For $i > j$, it is enough to show that $L_{ik} L_{jk}^T$ is zero, for all k , $1 \leq k \leq j-1$. Suppose for a contradiction $L_{ik} L_{jk}^T \neq 0$, for some k . Then $L_{ik} \neq 0$, and $L_{jk}^T \neq 0$. Then by induction $A_{ik} \neq 0$ and $A_{jk} \neq 0$ since $i + k < m$ and $j + k < m$. $A_{ik} \neq 0$ and $A_{jk} \neq 0 \Rightarrow$ the nodes i and j are both fathers of the node k since the quotient tree is monotonely ordered - but each node has at most one father. So $i = j$ contrary to the assumption that $i > j$. Hence $L_{ik} L_{jk}^T = 0$ for all k , $1 \leq k \leq j-1$. Hence the lemma holds for all m .

Algorithm for Quotient Tree Partitioning

The algorithm suggested by George and Liu (1978, 1981) is closely related to the level structure presented for symmetric factorization in the envelope method (Chapter 3). Given a rooted level structure

$\mathcal{L} = (L_0, L_1, \dots, L_\ell)$, we attempt to refine the partition. By refining

on the levels of the level structure. Let $P = \{Y_1, \dots, Y_P\}$ be a partitioning obtained by subdividing each L_j as follows. Let B_j be the graph prescribed by

$$B_j = G\left(\bigcup_{i=j}^{\ell} L_i\right)$$

and each L_j be partitioned by

$$\{Y \mid Y = L_j \cap C, G(C) \text{ is a connected component of } B_j\}.$$

Let $\text{Span } Y = \{x \in V \mid \text{there is a path from } y \text{ to } x, \text{ for some } y \in Y\}.$

Algorithm: (We assume that we are given a root r for the rooted
5.2.1 level structure defined.)

Step 1 - $P \leftarrow \emptyset$

\ Empty the stack. Generate the level structure

$$\mathcal{L}(r) = \{L_0, L_1, \dots, L_{\ell(r)}\}.$$

Set $\ell = \ell(r)$ and choose $u \in L_{\ell(r)}$ and

let $S = \{u\}$;

Step 2 - If the node set T on the stack belongs to

L_{ℓ} , pop T and $S = S \cup T$;

Step 3 - Determine the set $Y = \text{Span}(S)$ in the subgraph

$G(L_{\ell})$. If $\ell < \ell(r)$ and some node in

$\text{Adj}(Y) \cap L_{\ell+1}$ has not been placed in a

partition member, then go to Step 6;

Step 4 - $P = P \cup \{Y\}$;

Step 5 - Determine $S = \text{Adj}(Y) \cap L_{\ell-1}$ and set $\ell = \ell - 1$.

If $\ell \geq 1$ go to Step 2, else stop.

Step 6 - Push S onto the stack.

Pick $y_{\ell+1} \in \text{Adj}(Y) \cap L_{\ell+1}$ and trace a path
 $y_{\ell+1}, \dots, y_{\ell+t}$ where $y_{\ell+t} \in L_{\ell+i}$ and
 $\text{Adj}(y_{\ell+t}) \cap L_{\ell+t+1} = \emptyset$. $S = \{y_{\ell+t}\}$.
 $\ell = \ell + t$ and then go to Step 3 (form
 new partition member).

One can choose the pseudo-peripheral node for the root. The diagonal blocks can be stored using the envelope structure. The nodes in each block are labelled arbitrarily. Since this is going to affect the storage scheme one can use the RCM algorithm on each block to reduce to the envelope structure within the block.

Algorithm: For $k = 1$ to p do
 5.2.2

(i) Determine the subset

$$U = \{y \in Y_k \mid \text{Adj}(y) \cap (Y_1 \cup \dots \cup Y_{k-1}) = \emptyset\}.$$

(ii) Apply the RCM algorithm on $G(U)$.

(iii) Number the nodes in $Y_k - U$ after U in
 an arbitrary order.

end;

p is given by $|P|$ where $P = \{Y_1, \dots, Y_p\}$.

If the quotient tree ordering produces a tridiagonal block structure on A , then using block structural-storing and treating each block as a full matrix considerable speed may be achieved in the factorization of A . The "hypermatrix scheme" (Noor and Voigt, 1975) is also a suitable method of storing A . In the hypermatrix scheme, the partitioned A may be stored as the elements of a hypermatrix, analogous to the way in

which the numbers are stored as elements of a matrix. An address or pointer matrix is used to identify the location of the non-zero submatrices of the hypermatrix. The decomposition of each submatrix and the other operations with the matrices can be speeded up treating the columns of each matrix as vectors.

5.3 Nested Dissection Methods

Consider an $m \times \ell$ grid having $n = m\ell$ nodes, $m \leq \ell$. The corresponding finite element matrix problem is given by $Ax = b$. We assume that, for some ordering of the equations from 1 to N , $a_{ij} \neq 0$ implies nodes i and j belong to the same small square. An artificial ordering of the grid points other than the natural ordering sometimes helps us to block diagonalize the matrix. The idea in the nested dissection method is to divide the grid by both horizontal and vertical separators as shown in the figure below. The regions 1-4 may again be separated in the usual way or by horizontal and vertical separators. This induces a block diagonal structure

(One step of the nested dissection ordering.)

bordered by block matrices. This is often called a block bordered diagonal factorization (BBDF) of the matrix.

Generalising the idea, we let V be the set of vertices of the $n \times n$ regular grid. Let S_0 be the set of vertices which lies on the

vertical line which separates V into two equal parts R_1 and R_2 . If we number the nodes of R_1 and R_2 by columns, followed by those in S_0 then we get the one level dissection ordering. We continue the dissection with the remaining two components. Choose vertex sets $S_j \subset R_j$, $j = 1, 2$, consisting of nodes lying on the mesh which as nearly as possible divides R_j into equal parts. The vertices in $R_j - S_j$ are numbered before those in the set S_j . This induces the same substructure in the two leading principal submatrices as that in the overall matrix. This process is repeated until no components are available for dissection.

We now consider the algorithm to induce the BBDF of the given graph $G = (V, E)$.

Algorithm: Step 1 - Let $R = V$, $N = |V|$.
5.3.1

Step 2 - Find a connected component $G(C)$ in $G(R)$ and construct a level structure of the component $G(C)$ rooted at a pseudo-peripheral node r

$$L(r) = \{L_0, L_1, \dots, L_\ell\}.$$

Step 3 - If $\ell \leq 2$, set $S = C$ and go to Step 4, else $j = \lfloor (\ell+1)/2 \rfloor$ and get $S \subset L_j$ where $S = \{y \in L_j \mid \text{Adj}(y) \cap L_{j+1} \neq \emptyset\}$.

Step 4 - Number the nodes in S from $N - |S| + 1$ to N
 $R = R - S$; $N = N - |S|$;
If $R \neq \emptyset$ go to Step 2.

The major disadvantage of using the nested dissection method in

CRAY-like computers is that the vectors are too short to take advantage of the vector processor. George, Poole and Voigt (1978) have suggested an incomplete nested dissection method, where the mesh subdivision is stopped before completion. At that stage we just use a band oriented ordering of nodes in each quadrant.

Suppose we stop the dissection process at, say, the ℓ^{th} stage and simply number the $n^2/2^{2\ell}$ ℓ -level node array grid column by grid column. The matrix A has $2^{(2\ell)}$ leading diagonal submatrices of size $(n/2^\ell)^2$ and band width $n/2^\ell$. The remaining diagonal submatrices correspond to the dissectors and these are full. George et al (1978) have showed (for CDC-Star computers-row wise dissection) that by stopping short 1 step, one may save as much as $(\frac{21}{4}n^2)$ locations for data management. The subsequent increase in the arithmetic operation count may be relatively small. However, it is difficult to say exactly when to stop the dissection. The start up time (usually of order 100 cycles) is negligible for CRAY-like computers, but there is always the nontrivial overhead associated with the sparse vector operations which may be viewed as additional start up time (Voigt, 1977). Furthermore experience with electrical network problems (Pottle, 1979) has shown that the speed of the solution of $Ax = b$ is increased in the vector-BBDF formulation, but not the factorization rate. The factorization may be speeded up if the blocks are treated as banded non-sparse matrices, but again the increased number of zero operations may offset this speed.

CHAPTER 6

Conclusions

To obtain the optimum performance from any computer it is necessary to tailor the computer program to suit the machine's architecture. The machine's effect is more pronounced if we consider vector computers for solving large and sparse problems. Vector computers tend to be more efficient if the vectors are long. Since most general sparse factorizations tend to produce short vectors, there is a biased opinion towards banded factorization in CRAY-like computers, but problems from different origins can lead to very different conclusions. The size and nature of the problem influences the choice of suitable methods for the solution of large sparse positive definite systems.

So far numerical tests (George and Liu, 1981) have been limited in scope and dealt only with finite element problems of relatively small size. Using the same set of numerical data these experiments have been replicated on a CRAY 1 (Lewis, 1982) and the results are presented in the figure below.

For small problems of $n \leq 1000$ experience has shown that sophisticated ordering methods simply do not pay and the RCM method described in Chapter 3 is preferable to minimum degree and nested dissection methods. For very large problems in electrical networks (Erisman, 1981) minimum degree methods are still used. Use of nested dissection methods to obtain BBDF looks more attractive since a minimum degree method introduces significant scattering in the matrix. It is unknown whether an incomplete nested dissection method would do a much better job on CRAY-like computers, but it has been shown theoretically (George et al., 1978) that it produces longer vectors and so should be more suitable for vector computers.

Regarding the storage requirements in these problems, it has been shown (Lewis, 1982) that only the refined quotient tree ordering is capable of factoring the matrix within the storage actually available. But in the hypermatrix structure of storage, the operations are done on blocks of matrices so that at any time only three matrices are needed in the central memory. The hypermatrix scheme is very adaptable to overlapping with I/O transfers.

Careful implementation of an algorithm in CRAY-machine language and use of chaining of different vector operations can increase the speed of execution. Overhead cost can be decreased to a very large extent in sparse matrix factorizations if one uses a bit compression/expansion method rather than index methods.

REFERENCES

- Bunch, J. R., 'Block Methods for Solving Sparse Linear Systems' in Sparse Matrix Computations, Bunch, J. R. and Rose, D. J., Eds., Academic Press, New York, 1976.
- Butschner, W., 'The Solution of the Seismic One-way Equation on Parallel Computers', in Parallel Computing 83, Feilmeier, M., Joubert, G., Schendel U., Eds., Elsevier Science Publishing Co., The Netherlands, 1984.
- Charlesworth, A. E., 'An Approach to Scientific Array Processing: The Architectural Design of the AP-120B / FPS-164 Family', Computer, September, 1981.
- Dongarra, J. J., Gustavson, F. G., and Karp, A., 'Algorithms for Dense Matrices on a Vector Pipeline Machine', SIAM Review, January, 1984.
- Duff, I. S., 'A Sparse Future' in Sparse Matrices and Their Uses, Duff, I. S., Ed., Academic Press, New York, 1981.
- Erisman, A. M., 'Sparse Matrix Problems in Electric Power Systems Analysis', in Sparse Matrices and Their Uses, Duff, I. S., Ed., Academic Press, New York, 1981.
- Fong, K. and Jordan, T., 'Some Linear Algebraic Algorithms and Their Performance on CRAY 1.' Report written under US Energy Research and Development Administration Contract No. W-7405-ENG-36 at Los Alamos Scientific Laboratory, Los Alamos, New Mexico, 1977.

George, A. and Liu, J. W-H., 'Algorithm for Matrix Partitioning and the Numerical Solution of Finite Element Systems', SIAM J. Numer. Anal. 15, 1978.

George A. and Liu, J. W-H., 'A Quotient Graph Model for Symmetric Factorization', in Sparse Matrix Proceedings 1978, Duff, I. S. and Stewart, G. W., SIAM, 1979.

George, A. and Liu, J. W-H., Computer Solution of Large Sparse Positive Definite Systems', Prentice Hall Inc., New Jersey, 1981.

George, A. and McIntyre, D. R., 'Analysis of Minimum Degree Algorithm to Finite Element Systems', SIAM J. Numer. Anal. 15, 1978, pp. 90-112.

George, A., Poole Jr., W. G. and Voigt, R. G., 'Incomplete Nested Dissection for Solving 'n by n Grid Problems', SIAM J. Numer. Anal. 15, 1978.

George, A., Poole Jr., W. G. and Voigt, R. G., 'Analysis of Dissection Algorithms for Vector Computers', J. Comp. and Maths. with Applics. 4, 1978, pp.287-304.

G. M. U. Technology Series, 'Array Processors, Competition for Super Computers', October 19, 1983.

Golub, G. H. and Van Loan, C. F., Matrix Computations, The Johns Hopkins University Press, Maryland, 1983.

Helmbrecht, V. and Zabolitzky, J. G., 'Greens-Function Monte Carlo on the CYBER 205', in Parallel Computing 83, Feilmeier, M., Joubert, G., Schendel, U., Eds., Elsevier Science Publishing Co., The Netherlands 1984.

Hockney, R. W. and Jesshope, C. R., Parallel Computers, J. W. Arrowsmith, Bristol, 1981.

Hwang, Kai, Ed., Tutorial Supercomputers: Design and Application, IEEE Comp. Society, 1984.

Johnson, P. M., 'An Introduction to Vector Processing', Computer Design, February, 1978, pp.89-92.

Kamiya, S., 'Practical Vectorization for the FACOM VP', Information Processing, 1983, pp. 389-394.

Kogge, P. M., The Architecture of Pipelined Computers, Academic Press, New York, 1981.

Lewis, J. G., 'Numerical Experiments with SPARSPAK', Technical Notes, Boeing Computer Services Co., Washington, 1982.

Madsen, N. K., Rodrique, G. H., and Karush, I. I., 'Matrix Multiplication by Diagonals on a Vector/Parallel Processor', Info. Processing Letter, Vol. 5, No. 2, June, 1976.

Noor, A. K. and Voigt, S. J., 'Hypermatrix Scheme for Finite Element Systems on CDC STAR-100 Computer', Computers and Structures, Vol. 5, 1975, pp. 287-296.

Pottle, C., 'Solution of Sparse Linear Equations Arising from Power System Simulation on Vector and Parallel Processors', ISA Transactions, Vol. 18, No. 3, 1979.

Ramamoorthy, C. V., 'Pipeline Architecture', Computing Surveys, Vol. 9,
No. 1, March 1977, pp. 61-102.

Stewart, G. W., Introduction to Matrix Computation, Academic Press,
New York, 1973.

Voigt, R. G., 'The Influence of Vector Computer Architecture on Numerical
Algorithms', in High Speed Computer and Algorithm Organization,
Kuck, D. J., Lawrie, D. H., and Sameh, A. H., Eds., Academic Press,
New York, 1977.

END

|2|4|0|2|8|6

FIN