PARALLELIZATION OF THE PRECONDITIONED CONJUGATE GRADIENT METHOD USING A PROCESSOR ARRAY

by

ROBERT A.M. ALLEN

A thesis
presented to the University of Manitoba
in partial fulfillment of the
requirements for the degree of
MASTER OF SCIENCE
in
ELECTRICAL ENGINEERING

Winnipeg, Manitoba, 1983

PARALLELIZATION OF THE PRECONDITIONED CONJUGATE GRADIENT METHOD USING A PROCESSOR ARRAY

BY

ROBERT A.M. ALLEN

A thesis submitted to the Faculty of Graduate Studies of
the University of Manitoba in partial fulfillment of the requirements
of the degree of

MASTER OF SCIENCE

ABSTRACT

This thesis presents a parallel implementation of the Polynomial Preconditioned Conjugate Gradient (PPCG) method for the solution of large, sparse, and symmetric sets of linear equations. The algorithm uses a truncated Neumann series expansion to obtain an approximate inverse of the system matrix for use as the preconditioning matrix. The PPCG algorithm incorporates a sparse matrix storage scheme so that large sparse systems can be handled with the maximum of efficiency.

The algorithm is specifically implemented on the International Computers Ltd. Distributed Array Processor. It is shown to be suitable for solving linear systems arising from both finite-difference and finite-element discretization of elliptic partial differential equations.

ACKNOWLEDGEMENTS

The author wishes to thank Dr. A. Wexler for his supervision and guidance during the evolution of this work.

The author also wishes to thank his colleagues in the Numerical Applications Group at the University of Manitoba for their helpful discussions. In particular, the author wishes to thank Mr. R. Nakonechny for his invaluable help.

In addition, the author is indebted to Mr. Dennis Jackson and Dr. David Fincham at Queen Mary College, to Dr. John Simkin and Dr. Bill Trowbridge at Rutherford Laboratories, and to other anonymous people in England. Without their help, the practical side of this thesis would not have come to fruition.

Finally, financial assistance from the University of Manitoba Fellowship Fund and from the Natural Sciences and Engineering Research Council is gratefully acknowledged.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF PRINCIPAL SYMBOLS

A	symmetric system matrix
$\underline{x}$	unknown vector of linear system
$\underline{b}$	source vector of linear system
$\underline{h}$	exact solution of linear system
$\underline{r}$	residual vector
$\underline{d}$	direction vector of exact line search
$F(\underline{x})$	quadratic energy functional
$g(\underline{x})$	gradient of $F(\underline{x})$
$E(\underline{x})$	error functional
α	optimum line search constant for $\underline{x}$ and $\underline{r}$
β	optimum line search constant for $\underline{d}$
$P_k(A)$	matrix polynomial of degree k in A
K	preconditioning matrix
L	lower triangular factor of K

A'	transformed system matrix
$\underline{x}'$	transformed unknown vector
$\underline{b}'$	transformed source vector
$\underline{d}'$	transformed direction vector
$\underline{r}'$	transformed residual vector
M	part of system matrix splitting
N	part of system matrix splitting
ρ	spectral radius
z	parameter governing accuracy of the approximate inverse of the system matrix
K_z^{-1}	approximate inverse of the system matrix
M^{-1}	splitting of A chosen to be $(\text{DIAGONAL}(A))^{-1}$

Chapter I
INTRODUCTION

The Conjugate Gradient (CG) method is, at first, a very appealing candidate for parallel implementation. In its unpreconditioned form, it involves only vector operations and is often said to be 'trivially vectorizable'.

Preconditioning has, however, become a necessary part of the CG method. One discovers that without preconditioning, there may be no convergence or very slow convergence. The affect of preconditioning is to reduce the spectral radius of the system matrix so that the conjugate gradient procedure is more stable (i.e. converges), and converges in far fewer iterations.

With the introduction of preconditioning, the romance between parallel processors and the CG algorithm quickly dissolves. It is very difficult to find a preconditioning method that is parallelizable. For example, consider the incomplete Cholesky preconditioning method which is highly successful on scalar computers. The preconditioning process involved in the Cholesky algorithm is highly recursive. Because recursive processes are inherently serial, the incomplete Cholesky algorithm cannot be put into a parallel form.

The problem is further aggravated if one wishes to implement a sparse storage scheme in conjunction with a preconditioning method. Since the choice for one may preclude a desirable choice for the other, the two must be carefully chosen so that there will be a minimum of compromise.

At the same time, whatever decisions are made, must be made with an intimate understanding of the architecture of the processor that is being used. The architecture is perhaps the most overriding consideration. Its capabilities will dictate exactly what options are available for a sparsity and preconditioning scheme.

The preconditioning scheme used in this work is the polynomial preconditioning method first proposed by Dubois et al. [1979,257-268]. It is discussed in Chapter 5 along with the conjugate gradient and sparsity algorithms. This choice for a preconditioning scheme has the advantage of making no special requirements on the sparsity scheme, save that it allows efficient matrix vector multiplication (i.e. it requires no special data structure for efficient implementation). With one less constraint to consider, the integration of the sparsity scheme with the parallel architecture is made much easier.

The parallel processor used in this thesis is the International Computers Ltd. (ICL) Distributed Array Processor (DAP). With 4096 processors, the DAP has a great potential for parallelism. Chapter 4 gives an overview of the DAP and the facilities associated with it. Chapters 2 and 3 give some background material on parallelism and parallel architectures so that the position of the DAP in the hierarchy of parallel processors can be better appreciated.

Chapter 6 presents results from the application of the Polynomial Preconditioned Conjugate Gradient (PPCG) algorithm to some field problems. Scalar and vector versions of the algorithm are compared with each other and with a scalar implementation of the Incomplete Cholesky Conjugate Gradient (ICCG) algorithm.

Chapter II

CONCEPTS IN PARALLEL PROCESSING

2.1 THE NEED FOR PARALLEL PROCESSING

The need for parallel processing is increasing for several reasons:

1. The absolute speed of computer hardware is limited. Even the promise of Josephson junction switching technology can only increase raw computer speed by a factor of 10 or so. Propagation delay is another factor limiting circuit speed. Its effects can be reduced by making circuits physically smaller using very large scale integration (VLSI). The advance of VLSI is, however, reaching fundamental limits that will halt further improvement.
2. Computational needs in scientific algorithms have reached levels where no foreseeable scalar computer will be adequate.

To illustrate the latter point, consider that the National Aeronautics and Space Administration (NASA) has contracted Control Data Corporation (CDC) and Burroughs Computer Corporation to produce processors capable of performing in excess of 10^9 floating point operations per second (flops) (Hockney and Jesshope [1981,19]). This performance is needed to run three-dimensional aerodynamic simulation codes. When these needs are compared to the 10 million flop performance that current scalar computers are capable of, the hopelessness of the scalar approach can be seen.

NASA also has heavy computational needs for the processing of satellite imagery. It has contracted Goodyear Aerospace to build a highly parallel computer called the Massively Parallel Processor (MPP) (Schafer and Fischer [1982,32]). The machine is configured as a 128 X 128 processor array (see Section 3.3), and will be capable of performing 6 X 10^9 8-bit additions per second and almost 2 X 10^9 8-bit multiplications.

2.2 PROBLEMS TO BE ADDRESSED

Up to this point in time, the development of numerical algorithms has been geared toward implementation on scalar processors. The conversion of scalar algorithms to parallel algorithms is not a straightforward process. There is no way of predicting whether or not an efficient parallel implementation of a scalar algorithm exists.

The issue is further complicated by the existence of many different parallel architectures. A parallel algorithm developed for one architecture may not be suitable for another. It may be that a totally new architecture will have to be developed to implement an algorithm. VLSI could allow the parallel programmer to design specific hardware to meet his needs.

2.3 <u>ROOTS OF PARALLELISM</u>

Parallelism means different things to different people. Each person has different applications to apply it to and his measure of performance is based upon different goals. For example, in time-sharing environments, the goal is to process separate jobs at as fast a rate as possible. The rate at which jobs are processed is referred to as <u>throughput</u>. Greater throughput can be achieved by adding multiprocessing capabilities to the system, but this does nothing to speed up the execution time of a single program. As a result, the system programmer and the system user will have conflicting opinions about the gains achieved with the multiprocessing system.

Throughput is of no interest here, as the goal is to achieve <u>speedup</u>. That is, to decrease the total execution time of a single task or program.

Parallelism, in a broad sense, can appear at many levels. The various levels of parallelism may be described as follows (Hockney and Jesshope [1981,25]):

1. Job Level
   a) Between jobs - This level describes the system level and is implemented using multiprocessors. Parallelism at this level improves system throughput, and does nothing to improve the execution time of a single program.

b) Between phases of a job - This level of parallelism refers to the overlapping of slow system I/O (such as disk access) with fast program execution. One program can execute while others (assuming more than one I/O channel) are performing slow I/O operations. Again, this level of parallelism serves to increase system throughput.

2. Program Level

   a) Between parts of a program - This level of parallelism does serve to decrease the total execution time of a single program. This scheme, implemented in a multiprocessing environment, allows the simultaneous execution of unrelated parts of a program. Special language constructs, such as those in concurrent Pascal, are used to signal which phases of a program are unrelated. Some specialized compilers are also able to perform some parallel (data flow) analysis automatically.

   b) Within do loops - If each execution of a loop is independent of the previous, each processor may execute the loop at the same time as the others.

3. Instruction Level

   a) Between phases of instruction execution - This level describes the ability to divide the execution of a process or instruction into a succession of independent steps. This allows a number of identical processes to be in various stages of execution at the same time and is referred to as pipelining.

b) Between elements of a vector operation - This level of parallelism reflects the fact that many processes perform identical and independent actions with each element of a vector. An example of such a process is the multiplication of a vector by a scalar. All of the elemental multiplications may be performed concurrently. This type of operation may be implemented by processor arrays (see Section 3.3), or by pipeline processors (see Section 3.1.2).

4. Arithmetic and bit level

   a) Within arithmetic logic circuits - It is possible to perform arithmetic operations on numbers by processing all bits in parallel or by processing each bit of the numbers internal representation sequentially. The advantages/disadvantages of the two approaches stem mainly from the desired complexity of the hardware needed to implement them. The bit serial method is usually opted for in the processor array designs as it leads to simple processors that use little silicon area, thereby enabling the assembly of larger arrays.

2.4 AN ARCHITECTURAL TAXONOMY

Many parallel processor designs have been suggested over the last two decades. There have been a number of attempts to group the various architectures into classes that share some common basis. Like the architectures they attempt to group, they have had varying degrees of success. Most notable are those due to Flynn [1972,949], Shore [1973,95-109], and Hockney and Jesshope [1981,31-47].

Flynn's classification scheme describes the interrelation between machine instructions and the data upon which the instructions operate. It does not attempt any description of the details of the hardware that an architecture is built from. The result is that broad groupings of architectures are lumped under the SIMD (single-instruction stream - multiple-data stream) and MIMD (multiple-instruction stream - multiple-data stream) classes. Individual members of these groupings are indistinguishable from each other, and it is possible for an architecture to belong to more than one group. Confusion also exists about the location of pipelines in the scheme.

Taxonomies based upon the architectural features themselves (Shore's and Hockney's), are more specific and descriptive, but at the same time, may be confusing. Since detailed semantics serve only esoteric purposes, Flynn's general taxonomy will serve the purpose here.

Flynn's taxonomy is the one most frequently seen in the literature. The following classifications are observed:

1. The Single-Instruction stream - Single-Data stream (SISD) - This describes the conventional scalar processor and the pipelined scalar processor. Depending on the point of view, this class may also be extended to include the pipelined vector processor.
2. The Single-Instruction stream - Multiple-Data stream (SIMD) - This group includes most processor arrays, associative processors, and pipelined vector processors. In this class, a single instruction stream is broadcast to control a number of processors, each operating in lockstep. The processors each perform their operations on local memory.
3. Multiple-Instruction stream - Single-Data stream (MISD) - Although somewhat limited in scope, this class is said by Flynn to describe specialized streaming operations where a single data stream is used to produce a number of result streams.
4. Multiple-Instruction stream - Multiple-Data stream (MIMD) - All multiprocessing systems are lumped under this heading. That is, a group of processors each executing a separate program with local memory, and sharing results via a common memory or a switched communications network.

There is some confusion about the exact placement of pipelined vector processors in this taxonomy. Flynn places them under the SIMD classification owing to the fact that they have specialized vector instructions that can mimic those possible with a processor array. The argument against that placement is that a pipelined vector processor only operates upon a single data stream and the pipeline itself only performs a single instruction upon that data stream.

The given version of Flynn's taxonomy is the one that is in common use, and will be used here.

Chapter III
SOME PARALLEL ARCHITECTURES

3.1 SPECIAL PURPOSE FUNCTION UNITS

This category includes both systolic arrays and pipelined execution units. They are similar in concept but differ mainly in the scale at which they address parallelism. Systolic arrays are designed to implement whole algorithms while pipelines are usually designed to implement a single instruction (vector or scalar). In some respects, though, the terms are interchangable.

Systolic Arrays

Systolic arrays derive their name from the way data moves through them. They consist of a group of processors connected in a rigid communication pattern, much like the circulatory system of man. A clock cycle, analogous to a heart systole, 'pumps' the data through the array at regular intervals. Therefore, the movement of data through the array is similar to the movement of blood through a circulatory system and hence the term systolic.

Systolic arrays are designed so that data need only be given to them once. Thereafter, internal communication paths shuttle the data to where it is needed. Thus, a good systolic design will realize two savings:

1. Computations are pipelined. This introduces parallelism within the algorithm.
2. Data is loaded into the array only once, reducing expensive memory references to a minimum.

The main disadvantage of the systolic approach is the inflexibility caused by the rigid internal communication paths. These paths are designed for the execution of a single algorithm, and a complete redesign is needed to implement different algorithms. It may, however, be possible to remove this difficulty with a configurable processor array in which the communication paths are redirectable (Snyder [1982,47-56]).

Because of their importance in light of recent advances in VLSI design, a detailed discussion of a systolic array follows. In addition, they are architecturally similar to the ICL DAP, which is the main thrust of this thesis.

Systolic - Banded Matrix-Vector Multiplication

Much of the pioneering work with systolic arrays was done by H.T. Kung. This example is taken from his work in Conway and Mead [1980,263-332].

The workhorse of Kung's systolic designs is the inner product step processor shown in its linear connection configuration in Figure 3.1. This basic processor performs the calculation Y = Y + A*X. In calculation, the operands (A and X) are passed through unchanged while Y is augmented by the addition of the product (A*X). The following internal structure can be assumed for the basic processor:

1. each processor contains three registers - RA, RX, and RY,
2. each register has an input and an output connection, and
3. the output lines are latched and the logic is clocked so that neighbouring processors in the array do not interact during a computation cycle.

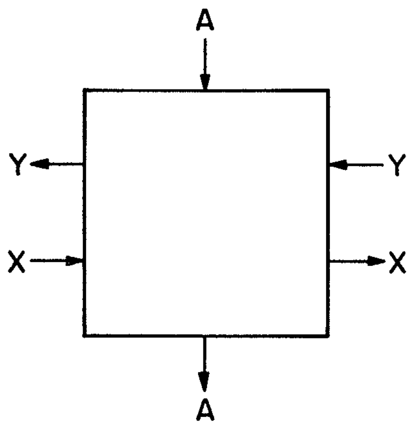

Figure 3.1: The inner product step processor.

In one cycle the inner product step processor performs the following operations:

1. shifts data off of the A, X, and Y lines into their respective registers,
2. computes RY = RY + RA*RX, and
3. puts contents of the registers onto their respective output lines.

with proper connections to its nearest neighbours, and perhaps the introduction of a few functionally different cells, the inner product step processor can be used to implement a large number of important numerical algorithms (Conway and Mead [1980,263-332], Kung [1982,37-46], and Snyder [1982,47-56]).

The banded matrix-vector multiplier can be seen in Figure 3.2. The figure depicts the data input into the array through seven cycles of the computation process along with the array states at the end of each cycle. The array multiplies a matrix with a bandwidth of 4 and order N with a vector of order N, where N is arbitrary.

The general geometry used in this example can be extended to matrices of larger bandwidth simply by adding more processors. In general, A matrix of bandwidth W and order N can be multiplied into a vector of order N using W processors. The computation is carried out in 2N+W time units, a much faster result then the WN time units needed for a scalar processor.

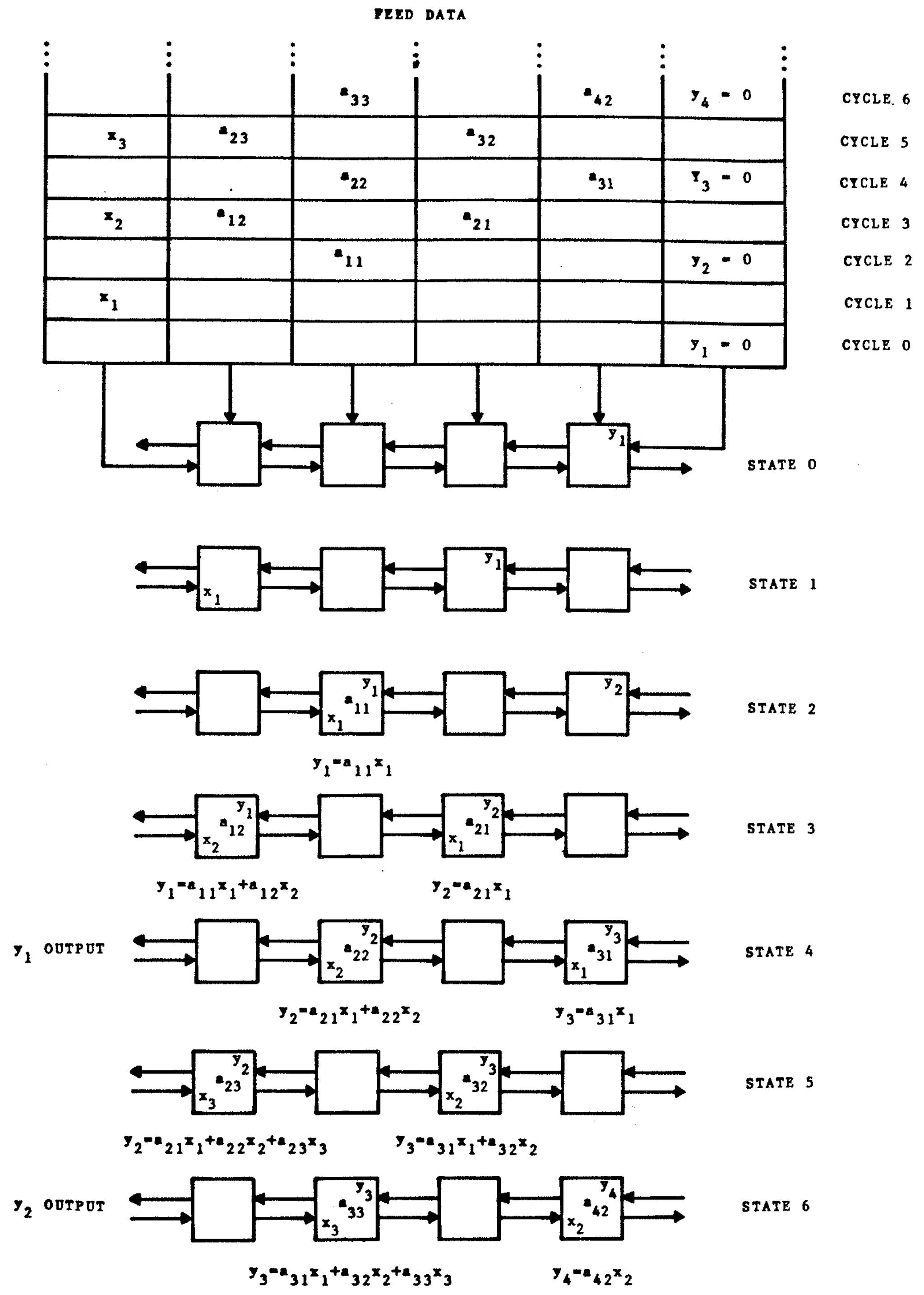

Figure 3.2: Banded matrix-vector multiplication.

Pipelined Processors

Pipelining refers to the disassembly of an instruction or process into a number of independent stages. These stages are cascaded so that each one, after processing its contained data, will pass data to the next element of the pipeline. Data enters one end of the pipeline and emerges from the other end altered by the complete operation that the pipeline implements. Data is shuttled from stage to stage once every clock period, which is defined as the longest execution time of the various stages in the pipeline. In this manner, a pipeline is able to produce a result once every clock period, which is generally much shorter than the time required for the whole instruction (sum of times for each of the stages).

A familiar example of pipelining is in the decoding and execution of instructions. The separate units allow the instruction stream to be processed in an overlapped fashion. When one instruction is in the execution unit, the operands for the subsequent instruction are being fetched etc.. In this fashion, instructions can be executed at a much faster rate then is possible with an unpipelined execution unit.

Other variations on the theme exist. Pipelines have been designed to implement addition, multiplication, and other special functions. Perhaps the most natural application, though, is to the processing of vectors of numbers. If two vectors have to be added, a pipeline can be used to implement the addition. Operands can be given to the pipe once every cycle, resulting in very efficient vector operations (especially for long vectors).

Most modern-day mainframes use pipelines somewhere in their architecture. Scalar computers such as the Amdahl 470V/6 and the IBM 360/195 have pipelined execution and arithmetic units, although they do not possess any vector instructions.

Vector pipelined processors are (arguably) the most powerful numeric processors in existence. The most notable of this breed are the CDC STAR 100, Texas Instruments TIASC, and the Cray Research CRAY 1.

An excellent review of pipeline architectures and principles is given in Ramamoorthy and Li [1977,61-102].

3.2 ASSOCIATIVE PROCESSORS

An associative processor can generally be described as a processor with the following two capabilities (Yau and Fung [1977,4-26]):

1. It is able to retrieve stored data based on their content or parts of their content (i.e. content addressing). This is very different from conventional computers where data is accessed via an absolute memory address.
2. It is able to perform both arithmetic and logical data transformations over many sets of data with a single instruction.

The first property places associative processors in a class by themselves, but because of the second property, they are generally grouped under the SIMD class. They differ from processor arrays (see Section 3.4) in that their addressing is based upon data content rather then on memory addressing.

The obvious application for associative processors is in data-base searching where they are able to do much of the work in parallel. With associative techniques, data-base machines can be made very efficient.

The two most important associative architectures are:

1. Fully Parallel
   a) Word Organized - Comparison logic is present at each bit of every word and the logical result is available at the output of every word.
   b) Distributed Logic - The comparison logic is associated with whole characters (groups of bits) or groups of characters.
2. Bit-Serial - One bit column (or bit-slice) of all words is operated on at one time. All words, then, are operated on in parallel.

An example of a fully parallel associative processor is the Parallel Element Processing Ensemble (PEPE) (Cornell [1976, 171-190]). The Goodyear Aerospace STARAN is an example of a bit-serial word-parallel associative processor (Meilander [1976,345-374]).

3.3 <u>PROCESSOR ARRAYS</u>

A processor array is generally a SIMD machine (an array of processors that perform identical operations in lockstep upon different data). This class should not be confused with the various 'array processors' that are on the market (so called because they are designed to process arrays of numbers). These special purpose function units are generally high-speed pipelined processors and not processor arrays.

Each processor in a processor array usually has nearest neighbour communication. To date, most arrays are arranged in a grid pattern with connections to their north, south, east, and west neighbours (orthogonally connected).

The processors used, such as those in the ICL DAP or the Goodyear MPP, are generally very simple bit-serial devices. The advantage of having simple processors can be seen by contrasting the ILLIAC IV (64 complicated processors) with the DAP (4096 simple processors) or the MPP (16384 simple processors). The use of a simple processor allows the assembly of much larger arrays and thus a potential for much more parallelism.

Another advantage of using simple bit-serial processors is that they may be implemented with VLSI very easily. Many processors could be put on one chip, simplifying the overall system design and reducing costs substantially.

The ICL DAP, a particular example of a processor array, is discussed more fully in Chapter 4.

3.4 DATA FLOW ARCHITECTURE

Data flow is as much a programming philosophy as it is an architecture. In fact, it is the philosophy that dictates what the hardware should do and the form it should take.

Data flow is based upon the principle that instructions contained within programs should be executed when their operands are available. If two or more instructions have their operands available simultaneously, then the instructions are executed concurrently. Programs executed in this manner are said to be 'data driven'.

Consider, for example, the following FORTRAN assignment statements:

1. A = X + Y,
2. B = X/A,
3. C = Y/A,
4. D = B + C, and
5. E = D/A.

Clearly, statements 2 and 3 can be executed concurrently, but only after the result of statement 1 is available. In a data flow machine, therefore, statement 1 would be dispatched to an execution unit first. When its result is ready, operations 2 and 3 would then be sent to separate execution units (along with their operands) and executed simultaneously. Each instruction carries information about where its result is to be sent, enabling the execution process to be done efficiently. This strategy requires, however, a large amount of preprocessing by the compiler to determine the data dependencies within a program.

The data flow concept, has no overhead associated with the synchronization of processors. Instructions are dispatched for execution when their operands are available, causing an automatic synchronization to occur. In addition, as long as there are enough execution units available to the program, its full parallelism can be exploited. Processor usage is also maximized to the point that the parallelism of the problem will allow.

It can be seen that the data flow approach (which gives a MIMD type of architecture) is very appealing. It offers the most efficient use of hardware that is possible. Implementation of a data flow machine is not an easy task, however, and there are other problems that need to be worked out before viable data flow machines will appear on the market (Gajski et al. [1982,58-69]).

For more information on data flow concepts see IEEE Computer [1982] which deals with the subject.

3.5 MULTIPROCESSING SYSTEMS

A multiprocessor is a MIMD device and has the basic definition given in the last chapter. Each processor stores its own program (perhaps a different program), and executes it independently from the other processors. The processors have both shared and local memory so that they may communicate with one another and share data.

Multiprocessor systems are more flexible than other parallel architectures (except perhaps data flow), but at the same time, their control is much more complex. One would expect that the speedup realized by using a multiprocessor would increase linearly with the number of processors, but overheads associated with the control and synchronization of the processors often reduce the speedup to far below linear.

Chapter IV
THE DISTRIBUTED ARRAY PROCESSOR

4.1 FACILITIES AT QUEEN MARY COLLEGE

Queen Mary College (QMC) forms part of the University of London, England. The QMC Computer Centre operates in affiliation with the University of London, which is also associated with a number of other installations.

The central processor used at QMC is an International Computers Ltd. (ICL) 2980. It provides the general services that one expects from a mainframe operation. Access to the 2980 is provided by an interactive terminal communications facility called Multi-Access Service (MAC) and a batch service.

Users external to QMC may communicate with the 2980 via three routes:

1. PSS - British Telecom's Packet Switch Stream network. This communications link also provides access to the IPSS international network.
2. SERC Network - A network operated by England's Science and Engineering Research Council.
3. METRONET - A network linking together the major computer centres affiliated with the University of London.

PSS/IPSS and SERC allow connection to MAC, while METRONET only provides access to the batch service.

MAC provides access to the CPU and filestore. A major limitation of the system is the rather small amount of disk space alloted each user: 226K. Any requirements beyond this amount require archival storage on catalogued tape. This process is made more or less automatic as files on tape can be referenced in a manner similar to that used for disk. Tape mounting is then done automatically by the operator.

User interaction with MAC is mediated by an ICL product called System Control Language (SCL). It allows the user to interact with his filestore and submit jobs.

The facilities at QMC are described in the QMC Computer Centre Handbook. The handbook consists of a number of separate documents, each describing various features of the operating system, hardware, and available software.

4.2 THE DISTRIBUTED ARRAY PROCESSOR (DAP)

DAP Hardware Overview

The DAP is installed at QMC as part of its 2980 service and forms an integral part of the computing environment. It is a SIMD processor array consisting of 4096 processors in a 64 X 64 configuration.

The DAP itself is configured as part of the main store of the ICL 2980 as shown in Figure 4.1. The DAP appears as a main store module of the 2980, but has

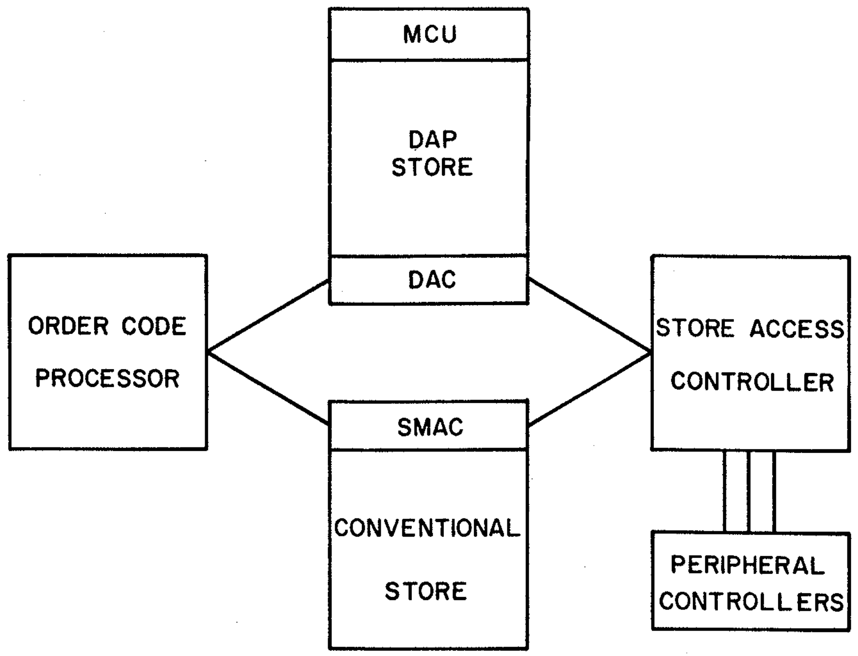

Figure 4.1: DAP - ICL 2980 system

(SMAC, Store Multiple Access Controller; DAC, DAP Access Controller; MCU, DAP Master Control Unit).

the additional capability of processing its contents with an integral processor array.

The major components of the DAP memory module (DAC - DAP STORE - MCU) are shown in Figure 4.2.

Two principal communication paths exist within the module. Both are 64 bit wide data paths, and provide communications between the 2980, DAP store module, and the MCU. The row highways have the specific task of feeding data into the MCU registers, while the column highways perform that task in addition to providing communications to the 2980 and fetching instructions for the execution unit in the MCU. Data may be transmitted to all processors simultaneously via the row or column highways, or each processor may 'AND' data onto the highways to allow global inquiries about the state of the processing elements.

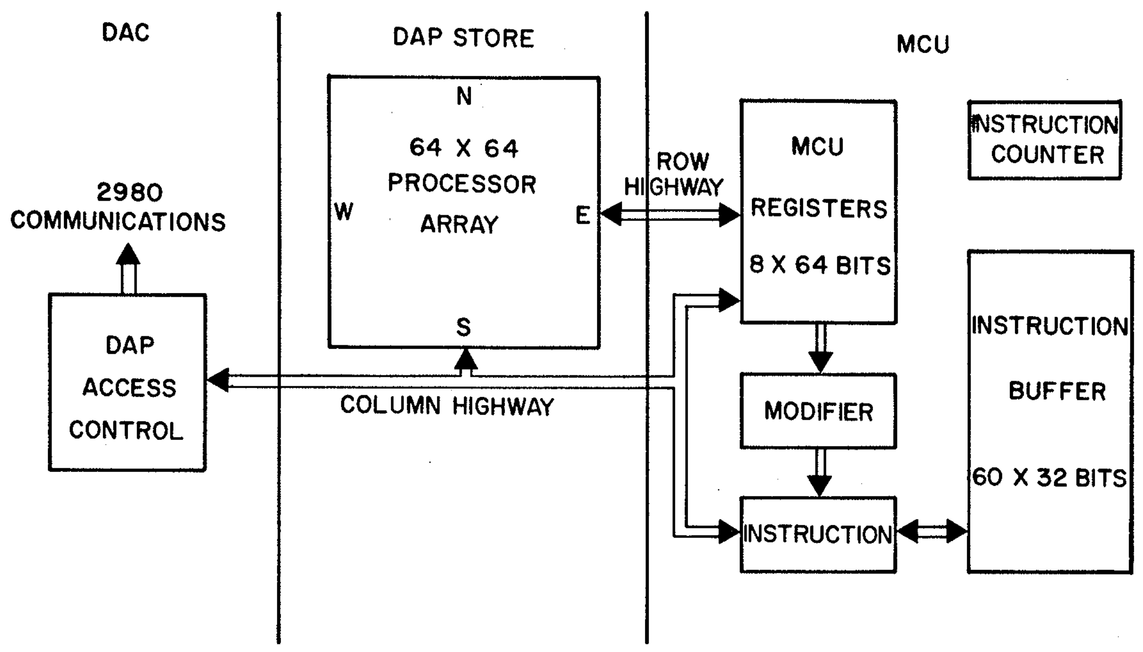

Figure 4.2: Internal organization of the DAP.

The MCU, as its name implies, is responsible for coordinating the operation of the DAP as a whole. It fetches instructions from DAP store, decodes them, and broadcasts appropriate commands to the processing elements (PEs) in the array. The MCU components, shown in Figure 4.2, have the following functions:

1. MCU registers - Used for data and/or instruction modification. Another use is to select (or transmit) data from (to) all processors in a row or column.
2. Modifier register - Used to hold operands in data and instruction modification, as well as to hold an address offset for instructions that reference memory.
3. Instruction register - Holds current instruction.
4. Instruction buffer - Buffers a sequence of 60 instructions for repeated execution in a hardware DO loop. The instructions contained in a loop are fetched only once, and can be executed repeatedly for up to 254 iterations.

For a detailed explanation of instruction execution on the DAP, the DAP APAL reference manual ICL [1979] can be consulted. APAL is the low-level assembly language in which the DAP is programmed.

The DAP store can be viewed as a 64 X 64 X 16K-bit cube, where each processor has a 16K-bit local store (see Figure 4.3). The 2980 is able to address DAP memory just as it does its normal memory. It sees each row of DAP memory (occupying a single store plane), as a 64-bit word. Higher addresses occupy first higher-numbered rows, and then deeper store planes.

The DAP processor array is orthogonally connected. Processors on the edge of the array may be configured to connect with the corresponding processor on the opposite edge

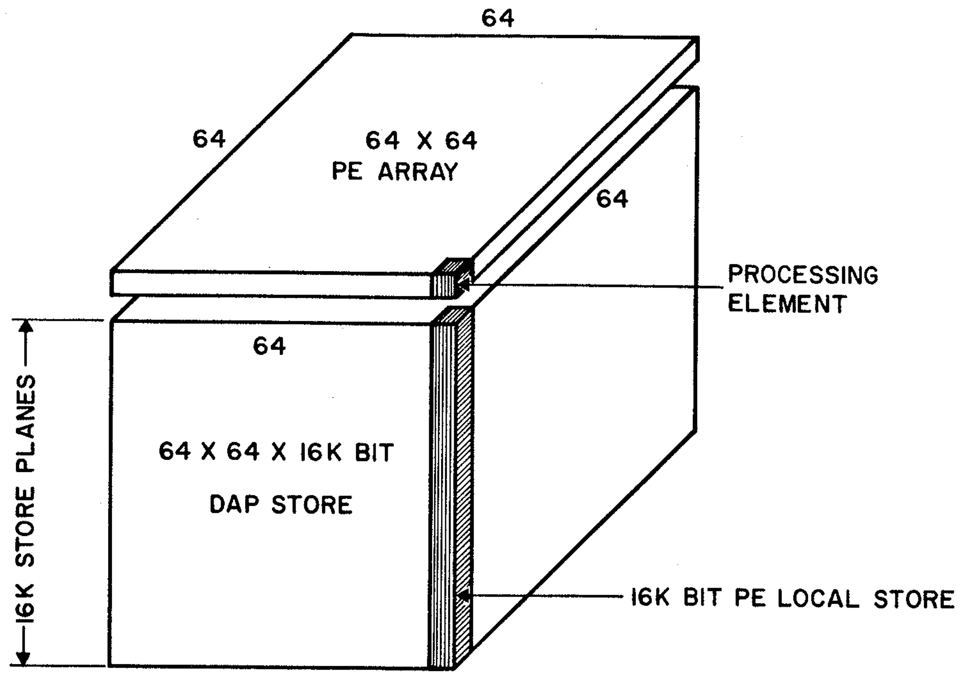

Figure 4.3: DAP - DAP store relationship.

of the array (cyclic geometry), or to receive zero for any communication with a nonexistent neighbour (planar geometry).

A schematic diagram of a DAP processor is shown in Figure 4.4. The processor contains three one-bit registers (A, Q, C) and a full adder that performs simple arithmetic on the contents of the registers or memory. The Q and C registers are generally used for the accumulation of sum and carry bits (respectively) generated by the full adder.

The A register, called the activity register, is of fundamental importance in the implementation of algorithms on the DAP. It can be used as a switch that enables or inhibits the execution of an instruction by a PE. This is necessary since it is a rare algorithm that will require the same operation to be performed by every PE always. Using the A register in this way is called MASKING.

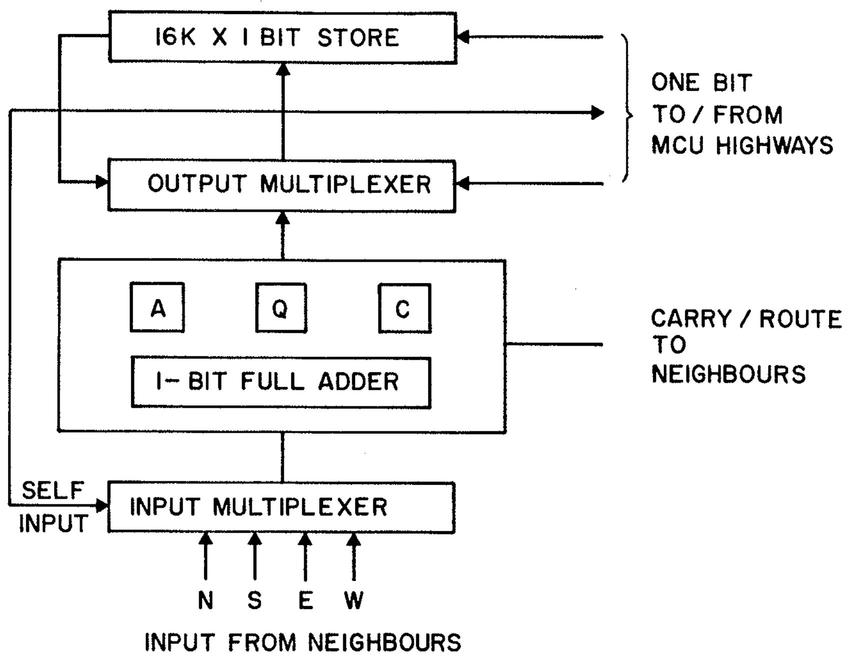

Figure 4.4: The DAP processing element.

DAP FORTRAN

DAP FORTRAN is the high-level programming language supported on the DAP. It is a fairly standard FORTRAN with specific extensions designed to allow the parallel processing capabilities of the DAP to be used easily. The language affords a user much more convenient access to the DAP than provided by APAL assembly language. It should be pointed out, however, that the use of APAL can be expected to yield more efficient programs.

DAP FORTRAN is fully described in two ICL documents: ICL [1980] and ICL [1981a]. The language is also described in Hockney and Jesshope [1981,242-246] and in Parkinson [1982,230-236].

Data Types

DAP FORTRAN is able to manipulate data objects consisting of vectors or matrices in much the same way that normal FORTRAN dialects manipulate their data objects (scalar variables). DAP FORTRAN'S data objects assume three forms or modes:

1. Scalar - Normal equivalent of FORTRAN variables.
2. Vector - An object consisting 64 independent elements. It is similar to a FORTRAN one dimensional vector but is limited in its length.
3. Matrix - A data object which contains a 64 X 64 array of elements. Besides the matrix representation, this object can also be used as a 'long vector' where successive columns are stacked underneath one another and referenced using a single index in the range [1,4096].

The three modes differ in the way that they are mapped onto the DAP store. The differences arise for reasons of efficiency when performing numerical calculations. In matrix mode, for example, the elements of a matrix (64 X 64) are stored one per PE. The operands for a calculation by a PE are then wholly located in its own store and the calculation proceeds bit-serially. In vector mode, on the other hand, elements of the vector are stored one bit per PE accross a row. Operations with vector elements are performed by a whole row of processors working in a cascaded fashion (scalars are also processed this way).

The data modes may be of type integer (1-8 bytes), real (3-8 bytes), double precision (8 bytes), character (1 byte), and logical (1 bit). Although integer and real variables are allowed to have different byte lengths, there is no storage efficiency to be gained by using the shorter lengths for vector and scalar variables. The only saving realized is in computation time, as arithmetic with shorter variables is faster. Table 4.1 shows some examples of variable declarations in DAP FORTRAN. Note the use of constrained dimensions (i.e. the null subscripts in '(,)') in the declaration statements. This type of declaration produces a vector or matrix of the maximum dimension allowed by the size of the DAP processor array (64 in this case).

TABLE 4.1

Variable Declarations in DAP FORTRAN

DECLARATION	RESULTING VARIABLE
INTEGER VI()	A 1X64 integer vector
REAL MA(,)	A 64X64 real array.
INTEGER VI(,5)	A 1X5 array of 1X64 integer vectors.
REAL MA(,,10)	A 1X10 array of 64X64 real matrices.
REAL*8 VI()	A real 1X64 vector containing 8 byte integers.
LOGICAL LB()	A 1X64 logical vector.
CHARACTER SK	A scalar character variable.

Numerical Operations

Arithmetic using DAP FORTRAN variables is essentially the same as that defined for scalar FORTRAN implementations. There are, however, some sensible extensions made to accommodate vector and matrix mode objects.

Assignment statements are exactly analogous to normal FORTRAN assignment statements. If vector or matrix mode variables are involved, the assignment is made via a parallel component assignment. That is, components on the left side of the assignment statement are made equal to the corresponding components on the right side of the assignment statement.

A natural restriction arising from this extension is that the quantities on either side of the equal sign be of the same mode, type, and length (numerical precision). Type incompatibilities are handled in a manner similar to that of scalar FORTRAN dialects. If possible, the type of the right side is changed to match that of the left side. Similarly, if the mode of the right side can be unambiguously 'expanded' so that its new mode matches that of the left side, an unlike mode assignment can be made. For example, consider a statement of the form:

```
VECTOR_VAR = SCALAR_VAR.
```

In this case, the scalar variable will be expanded into a vector with all of its components equal to the original scalar variable. On the other hand, the statement

```
MATRIX_VAR = VECTOR_VAR,
```

is not valid since there are two ways in which the vector to matrix expansion could be made (i.e. equal rows or equal columns). In this case, the desired expansion must be specified using built in DAP FORTRAN functions.

The unary operators '+', and '-', as well as the binary operators '+', '-', '*', '/', and '**', have extensions to allow use of nonscalar variables. The unary operations apply to all components of their argument, while the binary operations are performed between corresponding components of the two arguments. Thus, an expression of the form:

```
MATRIX_RESULT = MATRIX1*MATRIX2,
```

will produce a result matrix whose components are defined by

```
MATRIX_RESULT(I,J) = MATRIX1(I,J)*MATRIX2(I,J),
```

and not by the matrix multiplication formula.

A full complement of boolean unary and binary operators with vector and matrix extensions are provided. These operations are very useful in constructing logical masks for the indexing operations described in the next section. The mask can be used to inhibit operations for certain components of a vector or matrix variable.

In addition to the above, DAP FORTRAN provides many useful built-in functions. These functions can be divided into two groups:

1. COMPENENTIAL FUNCTIONS: examples of this group include the trigonometric and exponential functions. These functions operate on all modes in a parallel manner. For example,

   ```
   SIN(MATRIX_VAR),
   ```

   will produce 4096 simultaneous results.
2. AGGREGATE FUNCTIONS: these functions perform basic manipulations on vector and matrix mode objects. Examples of this class are shifting and expansion operations.

These two groups are fully described in the manuals ICL [1980], and ICL [1981a].

Indexing Techniques

DAP FORTRAN has very powerful indexing constructs that can be applied to vector and matrix mode variables. Arrays, vectors, or scalars can be selected from both declared variables and the results of functions and numerical expressions.

Indexing may be applied to both the right and left sides of an assignment statement. Indexing on the left selects those elements of a variable that are to be altered by the assignment, and indexing on the right selects the elements in a variable that are to be used in computation.

Indexing on the right can be used to make the following selections:

1. a scalar from a vector, vector array, matrix, or matrix array, and
2. a vector from a vector array, matrix, or matrix array, and
3. a matrix from a matrix array.

The mode of the value selected by an indexing expression is determined by the number of null subscripts in the expression. If no null subscripts exist, a scalar is selected. if one null subscript exists, a vector is selected, and if two exist, a matrix is selected.

A constrained subscript position (defined as the first of second index position), in an indexing expression may contain any of the following:

1. a null subscript,
2. an integer scalar value/expression in range 1 to 64,
3. an integer vector expression with component values in range 1 to 64,
4. a logical vector expression,
5. a logical matrix expression, and
6. a '+' or a '-' for shift indexing.

The shift indexing mentioned in point 6 makes use of the nearest neighbour communications that exist in the DAP architecture. A PE is allowed to share data via the row and column highways with its neighbours to the north, south, east, and west. The GEOMETRY statement controls how processors on the edge of the array are treated. Setting geometry to 'cyclic' gives an edge processor data from the corresponding processor on the opposite edge of the array, while setting it to 'planar' always gives a processor a zero value. The geometry is separately switchable for the N-S and E-W edges via the statement

```
GEOMETRY(N-S,E-W),
```

where the words 'plane' and 'cyclic' are placed as arguments to select the proper geometry.

Tables 4.2 and 4.3 show some examples of indexing techniques. The following declarations are assumed:

```
INTEGER V(), VARRAY(,5,5), M(,), IV(), MARRAY(,,5)
LOGICAL LV(), LM(,).
```

TABLE 4.2

Examples of Left Side Indexing

ASSIGNMENT	ACTION
M(,3)=V	Copies V into selected column of M
V1(2)=V2	Assigns selected component of V1 the corresponding component of V2.
M(,IV)=V	Assigns the selected component of vector M(,IV) the corresponding component of V.
V1(LV)=V2	Assigns V2(i) to V1(i) if and only if LV(i) is true. All other components of V1 are unchanged.
M1(LM)=M2	Similar to previous.
M(LV,3)=S	Assigns the scalar S to all components in the third column of M which correspond to a true element of LV.

Control Statements

DAP FORTRAN supports all the control statements common to most standard FORTRANS: IF, GOTO, DO, CONTINUE, CALL, STOP, and RETURN. These structures are enhanced somewhat to allow the use of vector and matrix mode variables in logical and loop limit calculations. In addition some debugging aids are provided by the TRACE and ERROR statements (ICL [1981a] and ICL [1981b]).

TABLE 4.3

Examples of Right Side Indexing

EXPRESSION	ACTION	RESULT
V(3)	Selects third component of vector V	Scalar
V(LV)	If a single component of LV is true, the corresponding component of V is selected.	Scalar
V()	Selects entire vector.	Vector
VARRAY(,3,2)	Selects a single column from the array.	Vector
M(2,3)	Selects single component of matrix variable.	Scalar
M(LM)	If only one component of LV is true, a single component of M is selected.	Vector
M(2,)	Selects a single row from M.	Vector
M(LV,)	When a single component of LV is true, the row corresponding to the non-zero element is selected.	Vector
M(IV,)	Selects a vector V where V(i)=M(IV(i),i)	Vector
M(LM,)	When LM has only one true component per column, a vector is selected whose components come from corresponding components of the columns of M.	Vector
M(,)	Selects entire matrix	Matrix
M(+,)	Selects (or forms) a matrix whose components are shifted one row position down from M's. The top row will be all zeros or equal to the bottom row if the geometry is plane or cyclic respectively.	Matrix

Program Structure

All DAP programs consist of two sections - a DAP FORTRAN or APAL section and a 2900 FORTRAN 'host' section.

The host section is needed to provide a call to the DAP section of the program and to provide all input/output (I/O) that may be needed for the job. This is a result of the fact that DAP FORTRAN has no I/O facilities.

Communication between the DAP and host sections of a program is performed via named common blocks. Owing to the fact that the two parts of the program use different data formats in memory, special DAP FORTRAN subroutines are provided to perform data mapping conversion. The ENTRY subroutine must convert relevant data from 2980 FORTRAN format to DAP FORTRAN format when called, and then back again when returning to the host program for data output.

The DAP FORTRAN section may be made up of subroutines, function subprograms, or block data subprograms. DAP FORTRAN subroutines can be declared in three ways:

1. SUBROUTINE name,
2. SUBROUTINE name(dummy arguments), and
3. ENTRY SUBROUTINE name.

The first two forms of declaration correspond to normal FORTRAN constructs with the natural extension to allowing vector and matrix mode parameters. The latter declaration denotes a subroutine that is to be called by the host section of the program. ENTRY subroutines provide the only access to the DAP facilities from the host FORTRAN section of the program. There may be more than one ENTRY subroutine if the user wishes.

DAP FORTRAN function subprogram declarations differ from standard FORTRAN declarations in the addition of a mode designator to the basic syntax:

type*length mode FUNCTION name(dummy arguments).

Thus DAP FORTRAN functions can return vector or matrix mode results.

Chapter V

THE CONJUGATE GRADIENT ALGORITHM

The conjugate gradient method (algorithm) will be derived from an optimization point of view. An iterative method will be developed that seeks a solution to a set of linear equations by requiring that each iterant minimize an error functional. The error functional is designed to give a measure of the current iterative solutions 'closeness' to the exact solution, and as such, the solution vector that minimizes the error functional will be the solution to the system of linear equations.

The derivation of the conjugate gradient method presented here follows that given by Axelsson [1977].

5.1 THE CLASSICAL CONJUGATE GRADIENT ALGORITHM

The solution to the system of equations

$$A\underline{x} = \underline{b} \tag{5.1}$$

is sought, where A is a symmetric positive definite N X N matrix, and $\underline{x}$ and $\underline{b}$ are respectively the unknown and forcing vectors (length N). Let the exact solution to the equations (5.1) be denoted by

$$\underline{h} = A^{-1}\underline{b}. \tag{5.2}$$

Given an estimate $\underline{x}$ of the solution vector, define the residual to be

$$\underline{r} = \underline{b} - A\underline{x}. \tag{5.3}$$

With the above definitions, consider the quadratic functional (Wexler [1980,5-21], Axelsson [1977,5-6])

$$F(\underline{x}) = \frac{1}{2}\langle\underline{x}, A\underline{x}\rangle - \langle\underline{b}, \underline{x}\rangle, \tag{5.4}$$

which is a so-called energy functional. The solution which minimizes (5.4) is the solution of minimum energy. Note that '< , >' denotes the standard inner product, which is assumed valid for real spaces.

As the name of the CG method implies, information about the gradient of the functional (5.4) is used to determine a path to its minimum. The gradient of (5.4) is given by

$$\underline{g}(\underline{x}) = \text{GRAD}(F(\underline{x})) = A\underline{x} - \underline{b}. \tag{5.5}$$

Noting the definition of the residual, (5.5) can be rewritten as

$$\underline{g}(\underline{x}) = -\underline{r}. \tag{5.6}$$

Observe here, that in following a path to the minimum of the functional, the negative of (5.6) is used since it is in the direction of the minimum.

Rewriting (5.4) in the form

$$F(\underline{x}) = \frac{1}{2}\langle(\underline{h}-\underline{x}),A(\underline{h}-\underline{x})\rangle - \frac{1}{2}\langle\underline{h},A\underline{h}\rangle, \qquad (5.7)$$

and using the fact that the last term is constant, it can be seen that minimizing (5.4) is equivalent to minimizing

$$E(\underline{x}) = \frac{1}{2}\langle(\underline{h}-\underline{x}),A(\underline{h}-\underline{x})\rangle, \qquad (5.8)$$

which shall be called the error functional. Two alternate forms of (5.8) are

$$E(\underline{x}) = \frac{1}{2}\langle\underline{r},A^{-1}\underline{r}\rangle, \qquad (5.9)$$

and

$$E(\underline{x}) = \frac{1}{2}\langle\underline{g}(\underline{x}),A^{-1}\underline{g}(\underline{x})\rangle, \qquad (5.10)$$

where $g(\underline{x})$ is given by (5.6). Note, that the gradients of (5.4), (5.8), (5.9), and (5.10) are equal.

In a CG iteration, one constructs a path through the space of solution vectors such that (5.10) is minimized and a solution is obtained on the Nth step. Each iterative step may be considered as an exact line search of the form

$$\underline{x}^{k+1} = \underline{x}^{k} + \alpha_k\underline{d}^{k}. \qquad (5.11)$$

That is, in proceeding from the current solution vector to the next, one travels along a direction $\underline{d}^k$ a distance α_k. The direction vector is chosen with some idea of the gradient, and the parameter α_k is chosen so that $\underline{x}^{k+1}$ will be located at the minimum of (5.10) along the line $\underline{p}^k$. The requirement that $E(\underline{x})$ be minimized successively by each step of the CG algorithm enables the value of α_k to be determined.

Observe that

$$E(\underline{x}^k+\alpha_k\underline{d}^k) = \langle(\underline{b}-A(\underline{x}^k+\alpha_k\underline{d}^k)),A^{-1}(\underline{b}-A(\underline{x}^k+\alpha_k\underline{d}^k))\rangle, \qquad (5.12)$$

can be written as

$$E(\underline{x}^k+\alpha_k\underline{d}^k) = -2\alpha_k\langle\underline{r}^k,\underline{d}^k\rangle + \alpha_k^2\langle\underline{d}^k,A\underline{d}^k\rangle. \qquad (5.13)$$

Setting the derivative with respect to α_k of (5.13) equal to zero gives the minimization requirement that

$$\alpha_k = \frac{\langle\underline{r}^k,\underline{d}^k\rangle}{\langle\underline{d}^k,A\underline{d}^k\rangle}. \qquad (5.14)$$

Now consider the calculation of the residual vectors for each CG iteration. While they may be calculated from (5.3), the matrix multiplication involved is not helpful. Using (5.3) and (5.11), the following recursive definition for the residual is obtained:

$$\underline{r}^{k+1} = \underline{r}^k - \alpha_k A\underline{d}^k. \qquad (5.15)$$

The matrix product in this formula can be used elsewhere in the algorithm, giving greater efficiency.

At this juncture, recursive definitions for both $\underline{x}$ and $\underline{r}$ have been determined. All that is needed to complete the algorithm is a definition for $\underline{d}$. It is the choice made for $\underline{d}$ that separates the CG method from the more general conjugate direction (CD) method. The conjugate direction method makes no specification on how the direction vectors are to be derived, save that they be A-orthogonal (i.e. $\langle \underline{d}, A\underline{d} \rangle = 0$). The conjugate gradient method on the other hand, requires that the direction vectors be constructed via A-orthogonalization of the residual vectors generated by (5.15).

The orthogonalization could be realized by a Gram-Schmidt process (Lang [1972,138-139]), but it is undesirable to store each $\underline{r}$ vector that is generated. Instead, the following iterative procedure is used:

$$\underline{d}^0 = \underline{r}^0, \tag{5.16}$$

followed by

$$\underline{d}^{k+1} = \underline{r}^{k+1} + \beta_k \underline{d}^k. \tag{5.17}$$

To prove the validity of this process, the orthogonality of the residual vectors must be demonstrated. With that fact, the A-orthogonality of the direction vectors can be proved, and finally the value of β_k determined.

Using (5.16), (5.15) can be rewritten as

$$\underline{r}^k = (I + C_1 A + C_2 A^2 + \; . \; . \; . \; . \; + C_k A^k)\underline{r}^0, \tag{5.18}$$

or

$$\underline{r}^k = (I + P_k(A))\underline{r}^0, \qquad (5.19)$$

where $P_k(A)$ is a polynomial of degree k in A with no constant term. Substituting (5.19) into the error functional (5.9) produces

$$E(\underline{x}^k) = \langle (I+P_k(A))\underline{r}^0, A^{-1}(I+P_k(A))\underline{r}^0 \rangle. \qquad (5.20)$$

Interpreting (5.20) as defining the square of a norm with respect to the matrix A^{-1}, the minimization of (5.20) is equivalent to requiring that $-P_k(A)\underline{r}^0$ be an approximation to $\underline{r}^0$ (so their sum will be zero). The best approximation to $\underline{r}^0$ will be when the error is orthogonal (in the A^{-1} norm) to the basis of approximating vectors (Davis [1975,176]).

It is therefore required that

$$\langle (I+P_k(A))\underline{r}^0, A^{-1}P_j(A)\underline{r}^0 \rangle = 0 \qquad (j \leq k). \qquad (5.21)$$

But

$$P_j(A) = A + AP_{j-1}(A). \qquad (5.22)$$

Substituting (5.22) into (5.21) gives

$$\langle (I+P_k(A))\underline{r}^0, (I+P_j(A))\underline{r}^0 \rangle = 0 \qquad (j \leq k-1), \qquad (5.23)$$

which (using (5.19)) demonstrates the orthogonality of the residual vectors.

With the result in (5.23), the A-orthogonality of the direction vectors can be proved. Let j be less than k, then (using (5.15) and (5.17))

$$\langle \underline{d}^k, A\underline{d}^j \rangle = \frac{\beta_{j-1}}{\alpha_k} \langle (\underline{r}^k - \underline{r}^{k+1}), \underline{d}^{j-1} \rangle. \tag{5.24}$$

Extending (5.24) by induction leaves

$$\langle \underline{d}^k, A\underline{d}^j \rangle = \frac{\beta_{j-1}\beta_{j-2} \cdot \cdot \beta_0 \langle (\underline{r}^k - \underline{r}^{k+1}), \underline{d}^0 \rangle.}{\alpha_k} \tag{5.25}$$

Using (5.16) and the orthogonality of the residual vectors shows that (5.25) equals zero, which proves the A-orthogonality of the direction vectors.

It remains for the value of β_k to be determined. The A-orthogonality of the direction vectors gives

$$\langle \underline{r}^{k+1} + \beta_k \underline{d}^k, A\underline{d}^k \rangle = 0 \,. \tag{5.26}$$

Solving for β_k produces

$$\beta_k = \frac{-\langle \underline{r}^{k+1}, A\underline{d}^k \rangle}{\langle \underline{d}^k, A\underline{d}^k \rangle}. \tag{5.27}$$

It is profitable to put (5.14) and (5.27) into more computationally efficient forms. Using (5.17) and the orthogonality of the residual vectors gives

$$\alpha_k = \frac{\langle \underline{r}^k, \underline{r}^k \rangle}{\langle \underline{d}^k, A\underline{d}^k \rangle}. \tag{5.28}$$

Substituting (5.15) and then (5.28) into (5.27) yields

$$\beta_k = \frac{\langle \underline{r}^{k+1}, \underline{r}^{k+1} \rangle}{\langle \underline{r}^k, \underline{r}^k \rangle}. \tag{5.29}$$

The derivation of the classical conjugate gradient algorithm is now complete. Using equations (5.3), (5.11), (5.15), (5.16), (5.17), (5.28), and (5.29), the algorithm can be summarized as follows:

$$\underline{x}^0 = \text{ARBITRARY}, \tag{5.30}$$

$$\underline{r}^0 = \underline{b} - A\underline{x}^0, \tag{5.31}$$

$$\underline{d}^0 = \underline{r}^0, \tag{5.32}$$

$$\alpha_k = \frac{\langle \underline{r}^k, \underline{r}^k \rangle}{\langle \underline{d}^k, A\underline{d}^k \rangle}, \tag{5.33}$$

$$\underline{x}^{k+1} = \underline{x}^k + \alpha_k \underline{d}^k, \tag{5.34}$$

$$\underline{r}^{k+1} = \underline{r}^{k} - \alpha_k A\underline{d}^{k}, \quad (5.35)$$

$$\beta_k = \frac{\langle \underline{r}^{k+1}, \underline{r}^{k+1} \rangle}{\langle \underline{r}^{k}, \underline{r}^{k} \rangle}, \quad (5.36)$$

and

$$\underline{d}^{k+1} = \underline{r}^{k+1} + \beta_k \underline{d}^{k}. \quad (5.37)$$

where k= 0, 1, 2, ... and the iteration terminates when the Euclidean norm of the residual vector is less than some prescribed value.

The listing in Appendix A includes the implementation of the above equations in DAP FORTRAN.

5.2 THE PRECONDITIONED CONJUGATE GRADIENT ALGORITHM

Theoretically, the CG method should terminate in a finite number of steps (less than or equal to N - the dimension of the linear system). If round-off error occurs, or if the system matrix has a large spectral condition number (defined as the ratio of the largest to the smallest eigenvalue), however, convergence may never occur or may take considerably more than N iterations. Also, for large N, even a well-conditioned system will require a large amount of execution time owing to the fact that each CG iteration is fairly expensive.

The slow convergence rate of the CG algorithm can be improved by performing a preconditioning process on the system matrix. The effect is to reduce the spectral condition number of the system matrix which in turn improves the convergence rate (Kershaw [1978,46], Axelsson [1977,17-23]).

Preconditioning is realized by multiplying the original system (5.1) by a matrix K^{-1} which is an approximate inverse to the system matrix. The ultimate preconditioning matrix would be the precise inverse of the system matrix, as multiplying by such a matrix would solve the system exactly in one iteration. The effect of preconditioning, then, is to put the linear system 'closer' to its solution.

Since our system matrix is symmetric, the matrix K will also be symmetric, and can be written as

$$K = (LL^T). \tag{5.38}$$

Multiplying (5.1) by K^{-1} gives the new system

$$(LL^T)^{-1}A\underline{x} = (LL^T)^{-1}\underline{b}. \tag{5.39}$$

For the present purpose, it is necessary to rewrite (5.1) as

$$(L^{-1}AL^{-T})(L^T\underline{x}) = (L^{-1}\underline{b}), \tag{5.40}$$

or (to define the primed quantities),

$$A'\underline{x}' = \underline{b}'. \tag{5.41}$$

The convergence properties of (5.41) and (5.39) will be identical since $(LL^T)^{-1}A$ and $(L^{-1}AL^{-T})$ are similar matrices and have the same eigenvalues. The CG method, now called the Preconditioned Conjugate Gradient (PCG) method, can be applied to the system (5.41) in a manner identical to its application to (5.1). This results in a set of equations 'identical' to equations (5.30)-(5.37), but with primed quantities replacing the normal quantities.

A', $\underline{b}'$, and $\underline{x}'$ are as defined in (5.4). As for $\underline{r}'$, re-writing (5.9) in the form

$$E(\underline{x}^k) = \frac{1}{2}\langle(L^{-1}\underline{r}^k),(L^{-1}AL^{-T})^{-1}(L^{-1}\underline{r}^k)\rangle, \tag{5.42}$$

allows the definition of $\underline{r}'$ as

$$E(\underline{x}^k) = \frac{1}{2}\langle\underline{r}'^k,A'^{-1}\underline{r}'^k\rangle, \tag{5.43}$$

giving finally

$$\underline{r}'^k = L^{-1}\underline{r}^k. \tag{5.44}$$

Similarly to (5.16),

$$\underline{d}'^0 = \underline{r}'^0. \tag{5.45}$$

The relationship between $\underline{d}$ and $\underline{d}'$ is somewhat arbitrary. The choice

$$\underline{d}'^k = L^T\underline{d}^k \tag{5.46}$$

is made as it results in a considerable simplification in the equations defining the PCG method.

With the above definitions for the primed variables, it is possible to transform the equations back to normal variables. The resulting algorithm is given in the following equations:

$$\underline{x}^0 = \text{ARBITRARY}, \tag{5.47}$$

$$\underline{r}^0 = \underline{b} - A\underline{x}^0, \tag{5.48}$$

$$\underline{d}^0 = K^{-1}\underline{r}^0, \tag{5.49}$$

$$\alpha_k = \frac{\langle \underline{r}^k, K^{-1}\underline{r}^k \rangle}{\langle \underline{d}^k, A\underline{d}^k \rangle}, \tag{5.50}$$

$$\underline{x}^{k+1} = \underline{x}^k + \alpha_k \underline{d}^k, \tag{5.51}$$

$$\underline{r}^{k+1} = \underline{r}^k - \alpha_k A\underline{d}^k, \tag{5.52}$$

$$\beta_k = \frac{\langle \underline{r}^{k+1}, K^{-1}\underline{r}^{k+1} \rangle}{\langle \underline{r}^k, K^{-1}\underline{r}^k \rangle}, \tag{5.53}$$

and

$$\underline{d}^{k+1} = K^{-1}\underline{r}^{k+1} + \beta_k \underline{d}^k. \tag{5.54}$$

where k=0, 1, 2, ... and the iteration terminates when the Euclidean norm of the residual is less then some prescribed value.

The PCG algorithm presented in (5.47) - (5.54) is a special case of the generalized CG method first presented by Hestenes [1956,83-102]. In his derivation, Hestenes places no requirements on the properties of the matrix K . As a result, the choice for K is not entirely obvious. The derivation presented here has the advantage of indicating exactly what properties K should possess.

The implementation of the PPCG algorithm in DAP FORTRAN is included in Appendix A.

5.3 CHOICE OF THE PRECONDITIONING MATRIX - K

At this point, the only constraint put on the matrix K^{-1} is that it be an approximate inverse of the system matrix. The method by which K^{-1} is obtained has not been specified.

For scalar processors, the most efficient way to obtain an approximate inverse seems to be the incomplete Cholesky factorization method put forth by Meijerink and van der Vorst [1977,148-162] (An implementation of which is discussed by Kershaw [1978,43-65]). Also, Nakonechny [1983] has shown that the the Incomplete Cholesky Conjugate Gradient (ICCG) method has the advantage of allowing efficient implementation of a linked-list sparsity scheme.

Despite its advantages, ICCG is not suitable for implementation on a parallel computer (Webb et al. [1982,325-329]). The incomplete Cholesky decomposition is inherently a recursive process that does not lend itself to parallel implementation. Successive column eliminations must proceed serially. While some parallelism can be extracted from a column elimination when a sparsity scheme is not used, with a sparsity scheme, parallel implementation is hopeless (especially on the DAP).

The goal here, then, is to arrive at an algorithm that:

1. incorporates preconditioning in its framework,
2. allows sparse storage of generally sparse system matrices, and
3. is efficiently implementable on the DAP.

The above requirements are met by combining the basic PCG algorithm with a class of polynomial preconditioners discussed by Dubois et al. [1979,257-268] and Johnson et al. [1983,362-376].

The Polynomial Preconditioned Conjugate Gradient (PPCG) algorithm approximates the inverse of the system matrix A by a truncated Neumann series expansion. This approximate inverse is then used as K^{-1} in the PCG algorithm ((5.47) - (5.54)). Consider the splitting of the system matrix

$$A = (M - N) = M(I - M^{-1}N). \qquad (5.55)$$

In exact analogy with the theory of scalar series (as opposed to matrix series), A^{-1} can be represented exactly by

$$A^{-1} = (I - M^{-1}N)^{-1}M^{-1}, \tag{5.56}$$

which can be written as

$$A^{-1} = \left(\sum_{i=0}^{\infty} (M^{-1}N)^{i}\right)M^{-1}, \tag{5.57}$$

or

$$A^{-1} = \left(\sum_{i=0}^{\infty} (I - M^{-1}A)^{i}\right)M^{-1}, \tag{5.58}$$

provided that (Mirsky [1982,332])

$$\rho(M^{-1}N) = \rho(I - M^{-1}A) < 1, \tag{5.59}$$

where ρ is the spectral radius of its matrix argument. The latter point (5.59), follows from the fact that a matrix raised to higher and higher powers will approach zero only if its spectral radius is less than one (Mirsky [1982,328]).

Owing to the fact that the calculation of (5.58) is intractable, an approximation to the inverse of the system matrix can be constructed by truncating the series (5.58) after a few terms (typically 1-4). Let the truncated inverse be defined by

$$K_z^{-1} = \left(\sum_{i=0}^{z-1} (I - M^{-1}A)^{i}\right)M^{-1}, \tag{5.60}$$

where the possible z values (one to infinity) determine the degree to which A^{-1} is approximated. The PPCG algorithm is therefore parameterized by the quantity z and shall, hereafter, be denoted as PPCG(z).

The matrix K^{-1} need not be explicitly calculated and stored. It is needed only for the matrix vector product in (5.47) - (5.54) of the form

$$\underline{c} = K_z^{-1}\,\underline{r}. \tag{5.61}$$

This product can be evaluated whenever it is needed, saving great storage costs (K^{-1} will be denser than A itself). This is a great advantage over the ICCG method which requires additional storage equal to the storage required for the A matrix. PPCG(z) requires only the storage of an additional vector of length N over the basic CG algorithm.

The value of z should be user-specifiable since the matrix vector multiplication required to evaluate (5.61) is very expensive. For increasing values of z, there is a definite trade-off between:

1. the decrease in total execution time resulting from the decrease in the number of iterations needed to achieve a specified accuracy, and
2. the increase in total execution time resulting from the increased execution time of a single iteration for higher z values.

The choice for M^{-1} in (5.60) is of fundamental importance. It must be such that (5.59) holds. An exceptional choice is to take

$$M^{-1} = (\text{DIAGONAL}(A))^{-1}. \qquad (5.62)$$

With M^{-1} of this form, (5.59) is guaranteed to hold if A is strictly or irreducibly diagonally dominant (Varga [1965,73]). In particular, if A is a real N X N matrix, and $(a_{ij}) \leq 0$ for all $i \neq j$, then $M^{-1}N$ is nonnegative, irreducible, and convergent if (Varga [1965,84]):

1. A is nonsingular and A^{-1} is > 0, or
2. the diagonal entries of A are positive real numbers.

Matrices of this type arise in many cases of interest. Varga [1965,161-208] demonstrates that matrices with the above properties arise naturally from the finite-difference solution of elliptic partial differential equations.

An additional advantage of (5.62) is that any matrix-vector products involving M^{-1} in the evaluation of (5.61) can be replaced by a vector-vector product. The latter is especially efficient on the DAP.

It can be seen that the preconditioning algorithm presented here is only as good as the matrix-vector multiplication routine used to implement it. It is important that the sparsity scheme chosen allows a very efficient routine to be coded. This point is especially important with parallel processors since the architecture will often limit the usable sparsity schemes with a resulting limitation in the options available for matrix-vector multiplication routines.

The implementation of PPCG(z) on parallel processors will involve compromises between sparsity schemes and multiplication routines. These considerations as applied to the ICL DAP are discussed in the following two sections.

5.4 SPARSE MATRIX STORAGE SCHEME

The use of sparse matrix storage in numerical analysis is an absolute necessity. In general, the matrices arising from finite-element and finite-difference analysis are very large and very sparse. Dense storage of such matrices is impossible, and as such, schemes must be devised that store only the nonzeros of a given matrix.

There are special problems associated with storing sparse matrices on the DAP. It is of fundamental importance that the data structure chosen allow the full parallelism of the DAP (4096 simultaneous operations) to be exploited. This goal can be achieved only if numerical operations are performed using matrix mode data objects.

A problem arises, however, when it becomes necessary to reorder data so that matrix arithmetic can be used. The data reordering process can consume considerable amounts of time. Indeed, the matrix - vector multiplication algorithm presented in the next section spends most of its time performing data reorganization and a very minimal amount of time doing actual addition and multiplication.

It is here that a deficiency in the DAP design comes to light. The limited processor interconnection pathways do not allow for efficient data reorganization. If it were possible to implement a permutation network along with the DAP, its power would be greatly enhanced.

The storage scheme and its associated matrix - vector multiplication algorithm used here, are adaptations of those suggested by Parkinson [1981]. The system matrix is stored so that the nonzero coefficients are stored one row per processor. That is, the contents of a row are entirely contained within the local store of a single processor. The row number that a processor stores is given by its long vector order number defined in Section 4.2.2.1. Each coefficient is stored along with an integer number that indicates which column it belongs to.

Thus, the system matrix is entirely described by the following three quantities:

1. the coefficient values stored as a matrix array,
2. the long vector position that the coefficients occupy, and
3. an integer matrix array whose entries give the column position of the corresponding element in the coefficient array.

Figure 5.1 shows an example of this data structure for a hypothetical 2 X 2 DAP storing a 4 X 4 matrix. Note that the symmetry of the system matrix is ignored in this scheme.

$$\begin{array}{cccc}
 & \multicolumn{2}{c}{\text{STORAGE SCHEME}} & \\
\text{SYSTEM MATRIX} & \text{COEFFICIENT VALUE} & \text{COLUMN POINTER} & \\
\left[\begin{array}{cccc} 2 & 0 & 0 & 4 \\ 0 & 2 & -1 & 1 \\ 0 & -1 & 3 & 0 \\ 4 & 1 & 0 & 2 \end{array}\right] &
\begin{array}{c} \begin{bmatrix} 2 & -1 \\ 2 & 4 \end{bmatrix} \\ \begin{bmatrix} 4 & 3 \\ -1 & 1 \end{bmatrix} \\ \begin{bmatrix} 0 & 0 \\ 1 & 2 \end{bmatrix} \end{array} &
\begin{array}{c} \begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix} \\ \begin{bmatrix} 4 & 3 \\ 3 & 2 \end{bmatrix} \\ \begin{bmatrix} 1 & 1 \\ 4 & 4 \end{bmatrix} \end{array} &
\begin{array}{c} \text{STORE PLANE 1} \\ \\ \text{STORE PLANE 2} \\ \\ \text{STORE PLANE 3} \end{array}
\end{array}$$

Figure 5.1: Example of matrix storage scheme.

One small point should be mentioned here. Since it is probable that not every row will have the same number of nonzero coefficients, some zero entries will need to be stored in the coefficient matrix array. This happens in the store planes (defined in Figure 5.1) subsequent to the one that the last nonzero of a row is stored in. The column pointer corresponding to these zeros is arbitrarily set to 1. This is done with a view to using a standard system subroutine in the matrix - vector multiplication algorithm which requires that the column pointer values be in the range [1,4096].

5.5 MATRIX-VECTOR MULTIPLICATION ALGORITHM

A matrix vector product of the form

$$\underline{y} = A\underline{x} \, , \tag{5.63}$$

is to be evaluated. To this end, the A - matrix is assumed to be stored in the manner described in the previous section. Let the problem be of order N, where N is in the range [1,4096]. In addition, let processor k contain the kth component of $\underline{x}$ and $\underline{y}$.

The matrix - vector multiplication process can, in general, be viewed as N independent row vector - column vector scalar products. Since all of these scalar products are independent, they may be evaluated in parallel. This is done by processing the matrix store one plane at a time. Each plane contains one nonzero multiplier from a term in each of the N scalar products. All that is needed is to generate a matrix containing the proper multiplicand in each position so that the DAP FORTRAN matrix mode multiplication can be used, generating 4096 scalar product terms. This matrix of product terms can then be summed into the $\underline{y}$ vector to accumulate all the scalar products simultaneously.

The problem is to generate a matrix of multiplicands for every plane of matrix store. This matrix is best defined by the following formula:

MULTIPLICAND(I) = X(COLUMNS(I))

where the matrix COLUMNS indicates to what column the multiplier of MULTIPLICAND(I) belongs and is equal to the present plane of the column pointer matrix array.

There are two convenient ways to generate the above transformation.

The first is by using an existing subroutine in the DAP subroutine library at QMC (Liddel and Bowgen [1982]). This routine, called M01_PERMUTE1, performs precisely the required transformation. The main disadvantage to this approach is that the subroutine takes a considerable amount of time to execute (11 ms).

The second way is by use the DAP FORTRAN broadcast facility. The transformed matrix is built by testing the values contained in COLUMNS sequentially, and broadcasting the value X(TESTED VALUE) to all matrix positions in the MULTIPLICAND array which correspond to an occurrence of the tested value in the COLUMNS array. The assignments (broadcasts) in each step are done in parallel and are implemented using the logical mask indexing facilities of DAP FORTRAN. If fewer than about 500 broadcasts are needed for a particular plane of the matrix store, this approach will be faster than using M01_PERMUTE1.

The scheme used for a particular store plane is determined by counting the number of broadcasts needed for that plane. Since this analysis is quite expensive, it is not really necessary if only one multiplication is being done.

If there are many multiplications using the same A matrix, however, the analysis can be done once and used for all subsequent multiplications, making the extra costs incurred negligible.

The code implementing this multiplication algorithm is listed in Appendix B. The listing is heavily documented and should help to explain some of the finer points of the algorithm.

Chapter VI
TEST PROBLEM RESULTS

The examples presented in this chapter serve two purposes:

1. to study the properties of the PPCG(z) algorithm, and
2. to study the efficiency of the PPCG(z) algorithm on the DAP.

Parallel and serial versions of the PPCG(z) and CG algorithms are compared and contrasted with each other, and with a serial implementation of the ICCG algorithm for two electric field problems.

The first, a Dirichlet finite-difference problem, was implemented on both an ICL DAP and an Amdahl 5850. This example provides comparisons of the CG, ICCG, and PPCG(z) algorithms, as well as giving a measure of the performance of the ICL DAP.

The second example is a finite-element problem. It was included to show that, while not applicable in theory, the PPCG(z) method provides a viable solution technique for such problems. This problem was implemented on the Amdahl 5850 only, and as such, serves only to characterize the PPCG(z) algorithm in its own right, rather than the parallelization of it.

The serial conjugate gradient routines used here are derived from those of Nakonechny [1983]. These routines use a linked-list technique as a sparse storage scheme. This method is ideally suited to scalar processors, as it minimizes the searching time needed to implement a matrix-vector multiplication algorithm. The serial processor, then, is not penalized by asking it to implement the sparse storage and multiplication algorithms described in Chapter 5.

6.1 THE FINITE-DIFFERENCE PROBLEM

Consider the Dirichlet field problem shown in Figure 6.1. The solution of Laplace's equation for the potential ϕ in the interior region is sought (under the prescribed boundary conditions). The finite-difference analysis of such a problem produces a linear system whose coefficient matrix obeys precisely the properties required by the PPCG(z) method (see Section 5.3).

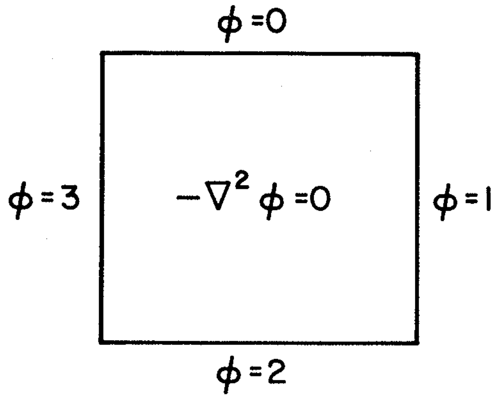

Figure 6.1: Finite-difference problem geometry.

In order to cater to the DAP processor parallelism, the region in Figure 6.1 was discretized into a 64 X 64 grid. This yielded 4096 unknowns, which precisely matches the number of processors in the DAP. It was not necessary to run smaller problems than this, as the DAP PPCG(z) algorithm is such that it will solve a 64 X 64 problem just as fast as it will a smaller problem.

The resulting linear system (with a 4096 X 4096 coefficient matrix) was solved using CG and PPCG(z) on the DAP, and using CG, ICCG, and PPCG(z) on the Amdahl 5850. Figures 6.2 and 6.3 show the iteration count and execution time properties of the various methods (excluding the Amdahl PPCG(z) results). As can be seen from the figures, parallel PPCG(z) competes very favourably with the scalar ICCG algorithm. While ICCG is the winner in reducing the iteration count (except for PPCG(4)), PPCG(2) is the clear winner when execution time is considered.

An interesting property of the PPCG method can be seen in Figure 6.2. A comparison of the PPCG(2) and PPCG(3) curves shows that higher preconditioning orders (higher z values) do not necessarily give lower iteration counts. This phenomenon can be explained via analytic convergence estimates given by Dubois [1979,264]. It is analogous to truncating an alternating scalar series at an undesirable term. For example, the Nth partial sum may actually be closer to the limit than the (N+1)th partial sum. Therefore, by analogy,

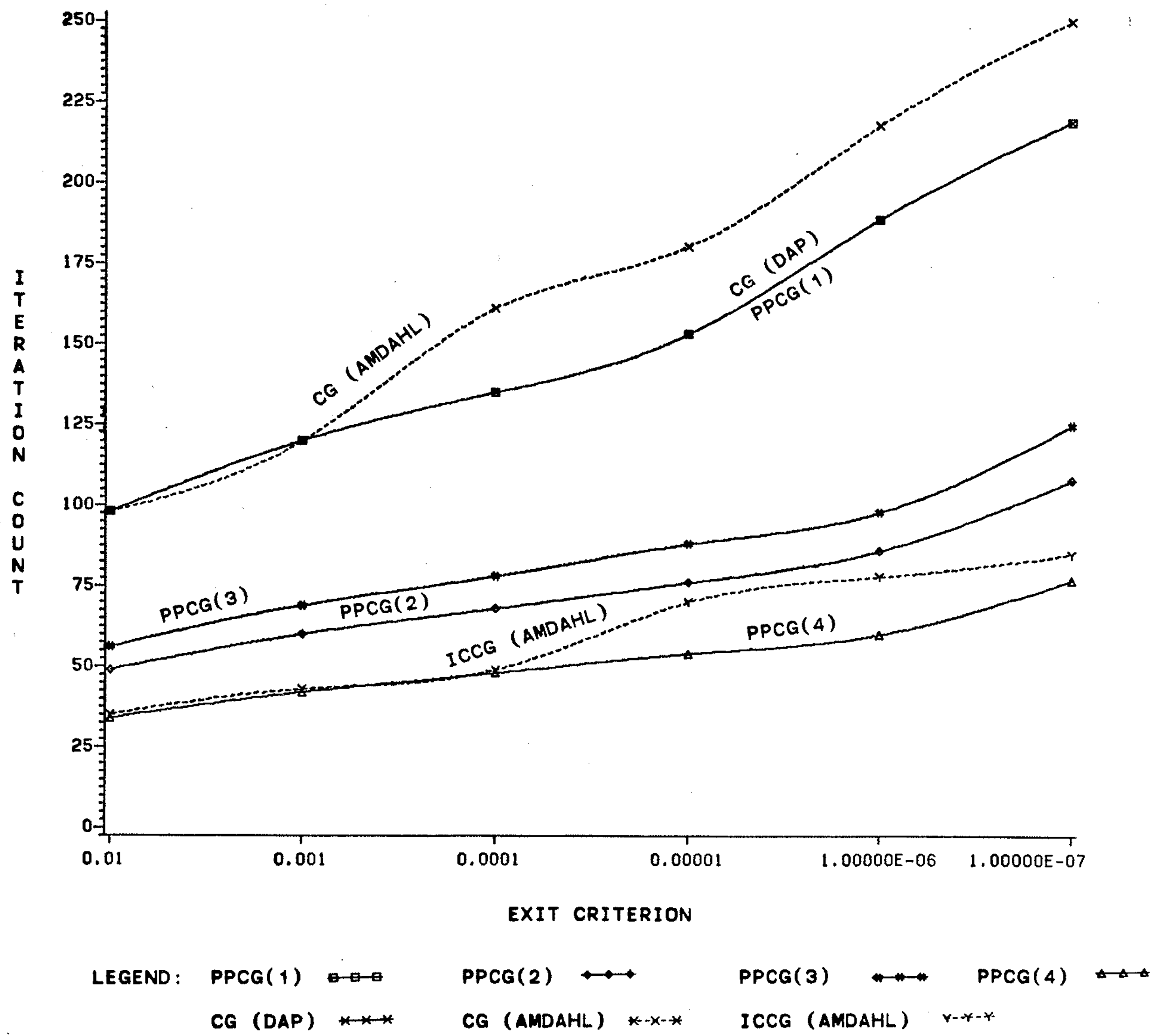

Figure 6.2: Iteration count vs. exit criterion for the finite-difference problem.

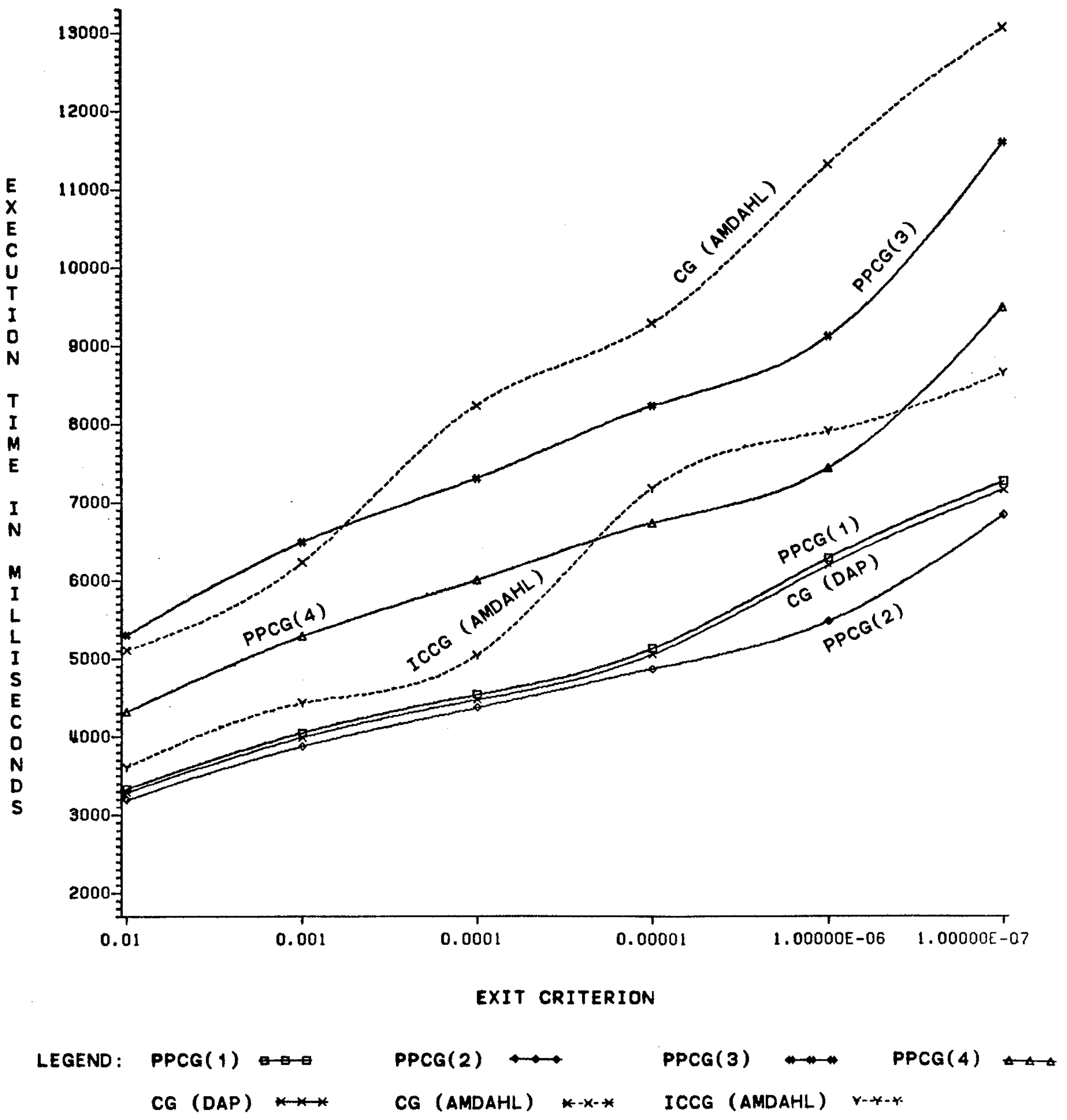

Figure 6.3: Execution time vs. exit criterion for the finite-difference problem.

the Nth estimate of the inverse of the system matrix may be closer to the actual inverse than is the (N+1)th estimate.

As mentioned in the previous chapter, the expense of single PPCG(z) iteration is highly dependent on the value chosen for z. The interaction of this property with the reduction in iteration count can be observed in Figure 6.3. While the iteration count for PPCG(4) is lower than that of PPCG(2), the greater expense of each iteration of the former causes it to do worse in terms of total execution time.

The expense of appending successively more terms onto the inverse series is also evident in Table 6.1. Each increment of z increases the execution time for a PPCG(z) iteration by 30 ms on the DAP and by 36 ms on the Amdahl. For the z values of interest (4 or less), this increase is very significant.

TABLE 6.1

Execution Time per Iteration for the DAP and Amdahl

	PPCG(1)	PPCG(2)	PPCG(3)	PPCG(4)	CG	ICCG
DAP	34	64	94	125	33	. . .
AMDAHL	54	90	125	162	52	103

Table 6.1 also gives performance comparisons between the DAP and the Amdahl 5850 (an example of a fast scalar processor). When comparing CG and PPCG(z) results, it can be seen that in all cases the DAP executes a single iteration faster than the Amdahl.

When comparing iteration counts for equivalent CG and PPCG(z) algorithms on the DAP and the Amdahl, an interesting observation comes to light. Comparing the curves for CG(DAP) and CG(Amdahl) in Figure 6.2, and corresponding curves for the PPCG(z) algorithm in Figure 6.4, one observes that the algorithms on the Amdahl require more iterations to satisfy a specified convergence requirement. Since the algorithms are identical in function, one can only conclude that the DAP algorithms produce less round-off error than do the Amdahl algorithms.

This phenomenon is due to the fact that the parallel algorithm is able to conserve precision by using a number of accumulators when evaluating the scalar products. Adding a small number to a large number occurs less often, and as a result, less rounding occurs.

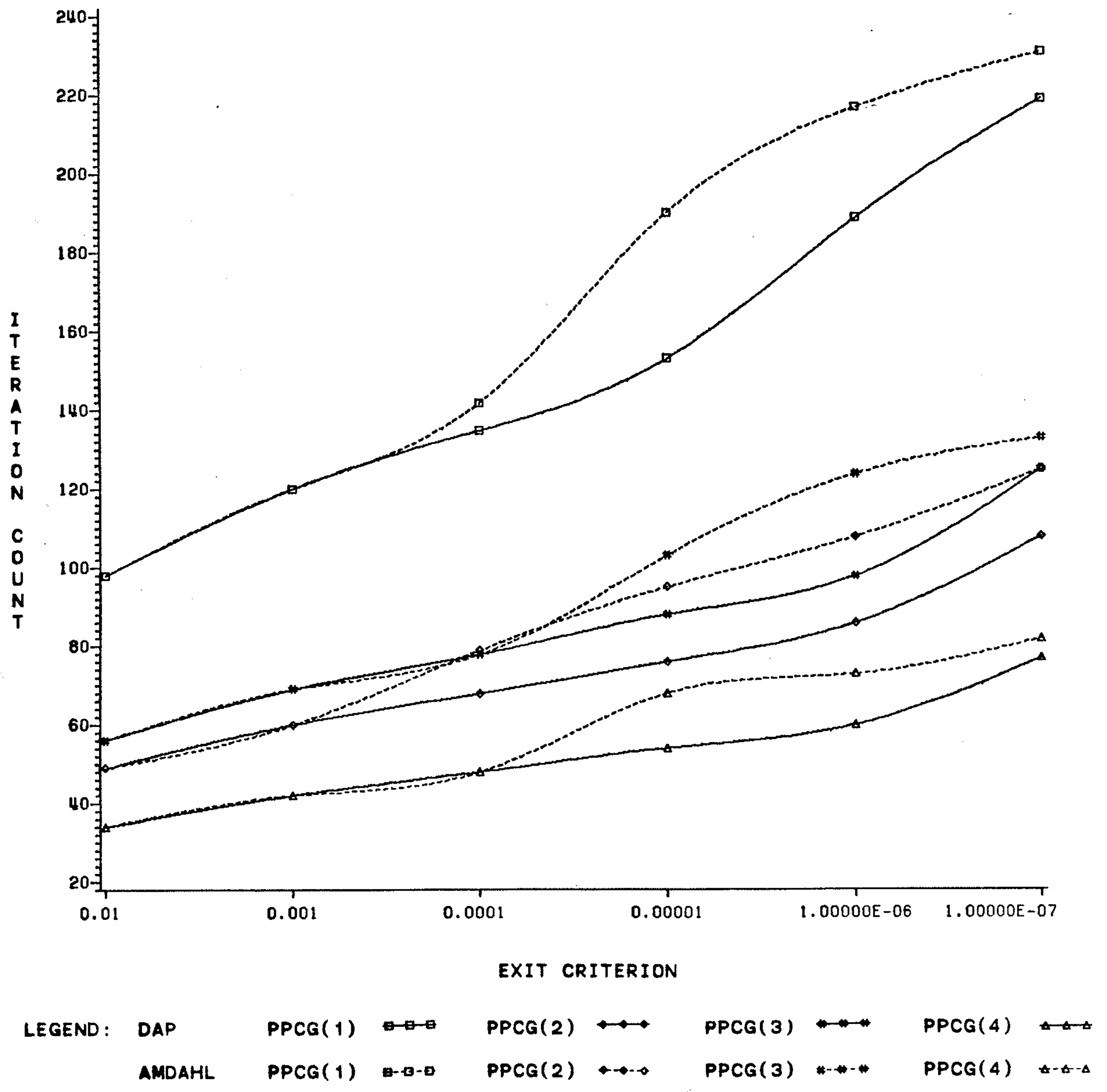

Figure 6.4: Convergence of DAP and Amdahl PPCG(z) algorithms.

6.2 THE FINITE-ELEMENT PROBLEM

As a second example, consider the geometry shown in Figure 6.5. In this field problem, the z-component of the magnetic vector potential (A_z) is to be determined (under the boundary conditions indicated). The region itself is divided into a number of areas of different magnetic permeability μ.

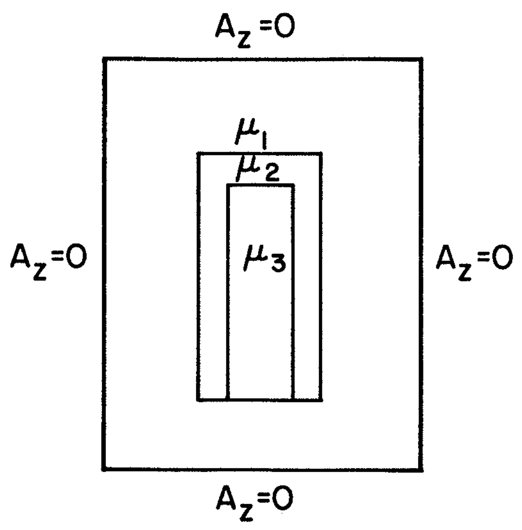

Figure 6.5: Finite-Element Problem Geometry

The finite-element solution of this problem produces a positive definite system matrix that does not meet the requirements of the PPCG(z) method (see Section 5.3). Despite this, the results presented here indicate that PPCG(z) is useful for solving finite-element problems.

Figures 6.6 and 6.7 show the iteration count and execution time results for the CG, ICCG, and PPCG(z) algorithms running on an Amdahl 5850 scalar processor. Note that a solution of this problem was not attempted on the DAP.

ICCG again excels in its ability to reduce the iteration count. Unlike the previous example, no PPCG algorithm of interest exhibits a lower iteration count than ICCG.

Total execution time is the telling factor, however. The execution time of PPCG(1) is much lower than that of ICCG for the full range of exit criteria considered, while PPCG(2) and PPCG(3) have lower execution times for those ranges of exit criteria where round-off error is not prevalent. The round-off error for tighter exit criteria indicated in figures 6.6 and 6.7 should not occur for the parallel PPCG(z) solution of this problem.

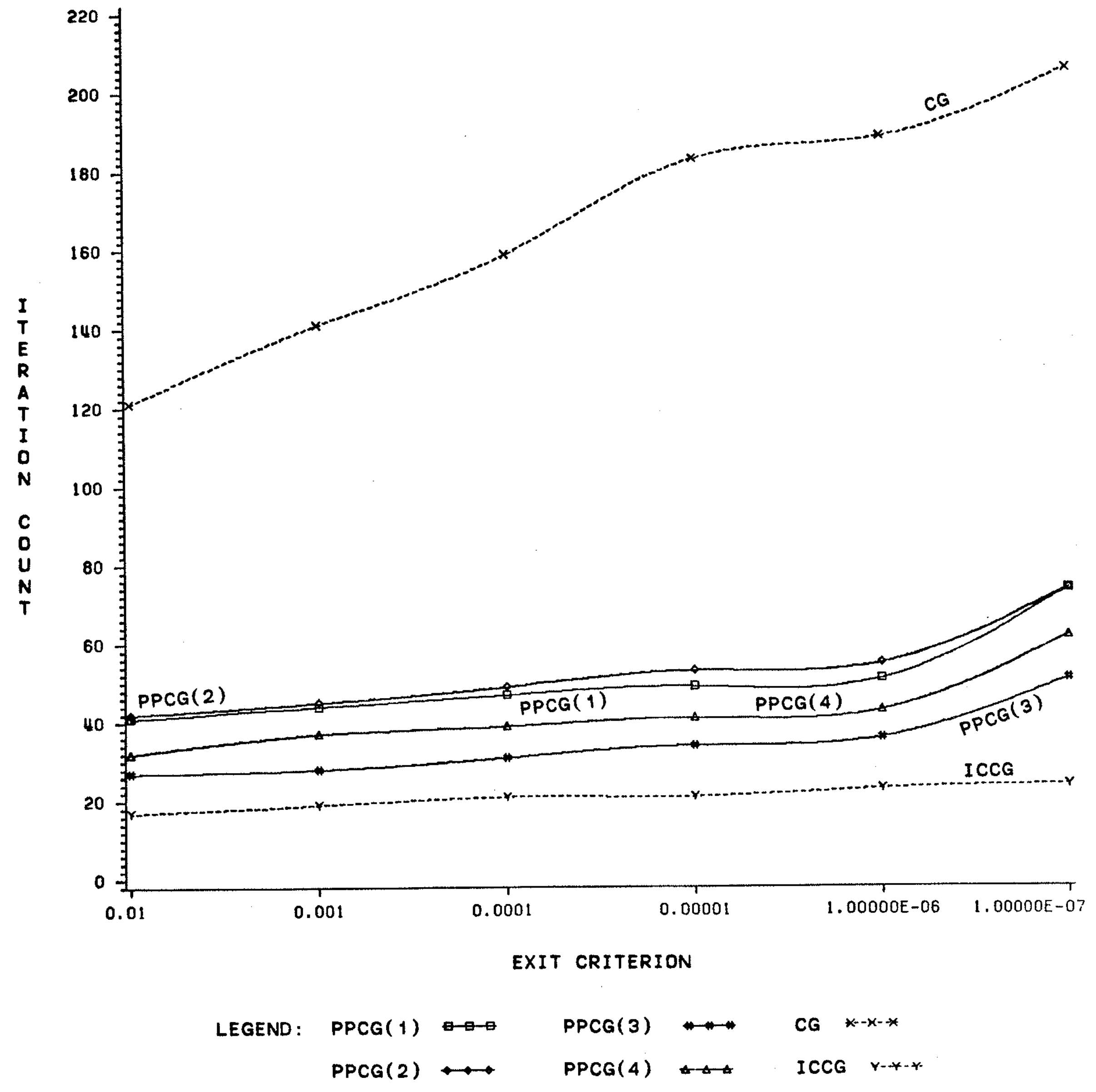

Figure 6.6: Iteration count vs. exit criterion for the finite-element problem.

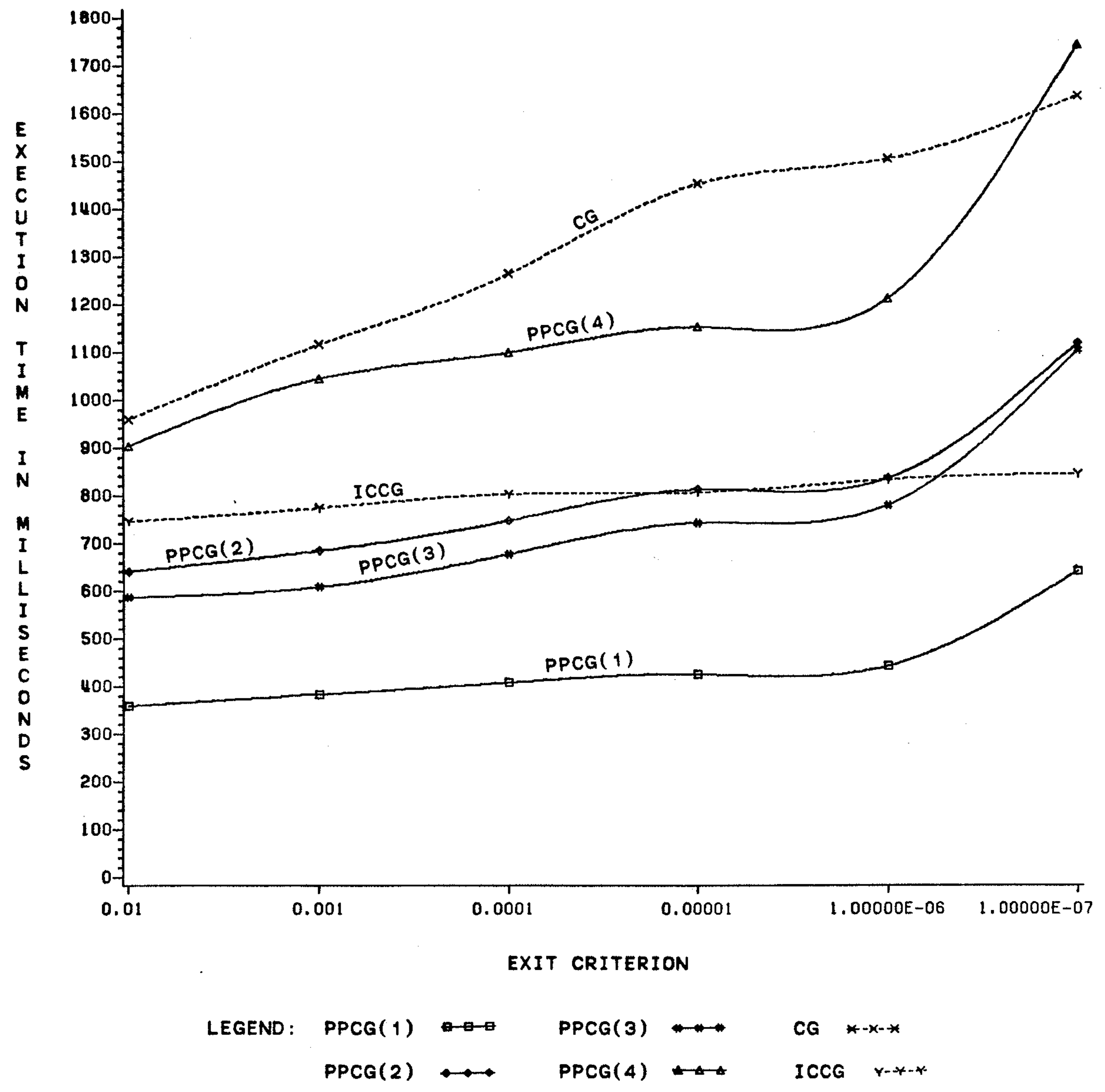

Figure 6.7: Execution time vs. exit criterion for the finite-element problem.

Chapter VII
CONCLUSION

This thesis has shown that polynomial preconditioning allows efficient parallelization of the preconditioned conjugate gradient algorithm on a processor array. The parallel implementation of PPCG(z) on the DAP attains faster solution times than an implementation of the ICCG algorithm on a fast scalar processor. The parallel PPCG(z) performance ranges from 1-2 times that of scalar ICCG. Also, both the PPCG(z) and ICCG algorithms are applicable to the same classes of matrices so that the two methods should be interchangeable.

Dubois performed polynomial preconditioning on a CDC STAR 100, which is a pipelined vector processor. His algorithm incorporated a sparsity scheme that was matrix dependent. The DAP PPCG(z) algorithm, however, was implemented with a general sparse matrix storage scheme. This feature is important as many of the solution techniques used in numerical analysis produce large, generally sparse matrices.

The parallel CG and PPCG(z) algorithms were shown to be more resistant to round-off error. The ability to easily accumulate into a number of locations when performing scalar products allows round-off effects to be reduced.

The PPCG(z) algorithm's performance is highly dependent upon the efficiency of the matrix-vector multiplication algorithm. If a more efficient multiplication algorithm could be found, the DAP could offer even greater speed advantages over a scalar processor running ICCG. This would, in turn, require the formulation of a different sparsity scheme.

One option is to adapt the sparsity scheme to the topology of the problem at hand. In the case of the finite-difference example, one could store only nonzero diagonals. The matrix-vector multiplication would then take the form of the algorithm suggested by Madsen et al. [1976]. This approach, however, has the disadvantage of decreasing the generality of the PPCG(z) algorithm.

Since the permutation operation is the bottleneck in the present matrix-vector multiplication algorithm, greater performance could be achieved by streamlining this operation. Software options to achieve this are limited. It would have to be realized with hardware additions to the DAP. A switching network like that used in the Burroughs FMP (Gottlieb and Schwartz [1982,30]) could be used to interconnect the DAP processing elements, resulting in faster long vector permutations. Such a hardware addition to the DAP would make it a much more powerful processor.

The incentive to either improve the DAP or the PPCG(z) algorithm increases if one considers the performance of the present algorithm when the number of unknowns exceeds the parallelism of the DAP.

Consider, for example, a problem with 8192 unknowns. The matrix for such a problem would be stored in two blocks, each identical to the storage scheme described in Chapter 5, one representing the first 4096 rows, and the second representing the last 4096 rows (low and high order respectively). Since the column pointer array for both the high and low order blocks may contain values in the range [1,8192] (i.e. from both row blocks), a permutation done for a plane in the low order matrix store must be accompanied by one using the corresponding plane in the high order matrix store (and vice versa). Thus, two permutations are required to process each plane in each matrix storage block. Since there are twice the number of matrix storage blocks, the number of permutations in the 8192 unknown problem exceeds that in the 4096 unknown problem by a factor of 2^2 (assuming that the number of nonzeros per row is the same for both problems).

Generalizing the above result, then, if the number of unknowns exceeds the number of DAP processors by a factor of j, the number of permutations needed to perform a matrix-vector multiplication increases by a factor of j^2 . Exceeding the DAP parallelism by two or three may be acceptable, but any larger a problem size would require reworking the algorithm.

The most natural way to extend the present PPCG(z) algorithm to larger problem sizes is to increase the size of the

DAP. A 128 X 128 DAP could solve matrices of order 16384 in the same time as the present DAP solves matrices of order 4096. A DAP of this size is entirely feasible (indeed, the 128 X 128 MPP processor array has already been built), and with the advances in VLSI technology of late, has a good probability of being built.

The major conclusion that can be reached from this thesis, in conjunction with Dubois' work, is that the preconditioned conjugate gradient algorithm does have a future in the world of parallel processing. It has been argued that, because of the inherent recursiveness of preconditioning algorithms, it would never be competitive with other direct or iterative methods. The incorporation of a general sparsity scheme into the parallel algorithm can only serve to strengthen this conclusion. The performance of the PPCG(z) algorithm on the DAP, while not earthshaking, indicates a potential for it to be a serious parallel linear-equation solver.

Appendix A

CONJUGATE GRADIENT LISTING

```
      SUBROUTINE CG(SN,SP,SMAXPLANES,MA,MCOLUMNS,LVX,LVB,
     &             SEXIT,SNUMITER,SPRECON_ORDER)
C
C*********************************************************************
C
C   PERFORMS CONJUGATE GRADIENT SOLUTION OF (MA)*(LVX) = (LVB)
C   WITH OR WITHOUT THE POLYNOMIAL PRECONDITIONING SCHEME
C
C*********************************************************************
C
C                         PARAMETER DICTIONARY
C
C*********************************************************************
C
C SN - DIMENSION LINEAR SYSTEM.
C SP - NUMBER OF PLANES OCCUPIED BY MA.
C SMAXPLANES - NUMBER OF STORAGE PLANES TAKEN BY MA
C            MAXIMUM VALUE FOR SP.
C MA - MATRIX OF COEFFICIENTS STORRED ONE ROW PER PROCESSOR.
C MCOLUMNS - GIVES COLUMN NUMBER OF CORRESPONDING MA ELEMENT.
C LVX  - UNKNOWN (RESULT VECTOR).
C LVB  - SOURCE VECTOR.
C SEXIT - EXIT CRITERION. WHEN THE EUCLIDEAN NORM OF THE RESIDUAL
C        IS LESS THAN THIS VALUE, THE SUBROUTINE EXITS.
C SNUMITER - THE NUMBER OF ITERATIONS TAKEN FOR CONVERGENCE.
C SPRECON_ORDER - INDICATES TO WHICH ORDER THE PRECONDITIONING
C                 POLYNOMIAL IS EVALUATED. ZERO INDICATES NO
C                 PRECONDITIONING.
C
C*********************************************************************
C
      INTEGER SN,SP,SMAXPLANES,SNUMITER,SLOOPLIM
      INTEGER MCOLUMNS(,,SMAXPLANES), MTEMP(,)
      REAL MA(,,SMAXPLANES)
      INTEGER I,J,K,VCOUNT()
      REAL LVX(,), LVB(,)
      REAL SALPHA,SBETA,S_OLD_PROD,S_NEW_PROD
      REAL LVANS(,), LVR(,), LVD(,)
      REAL SEXIT
      INTEGER SPRECON_ORDER
      LOGICAL SFIRST
      REAL LVDIAG_INV(,), LVFIRST_TERM(,), LVRESULT(,)
      INTEGER MLONG_INDEX(,),VTEMP(),PLACE
C
C*********************************************************************
C
C                        VARIABLE DICTIONARY
C
C*********************************************************************
C
C LVANS - RETURNS RESULT FROM SPARSE MULTIPLY.
C SALPHA - EXACT LINE SEARCH CONSTANT FOR NEW X VECTOR CALCULATION.
C SBETA - EXACT LINE SEARCH CONSTANT FOR NEW DIRECTION VECTOR
C         CALCULATION.
C LVD - DIRCTION VECTOR.
C LVR - RESIDUAL VECTOR.
```

```
C LVDIAG_INV - VECTOR HOLDING THE INVERSE OF THE DIAGONAL
C              OF THE SYSTEM MATRIX.
C LVFIRST_TERM - THE FIRST TERM IN THE POLYNOMIAL EXPANSION.
C LVRESULT - ACCUMULATOR FOR THE POLYNOMIAL EXPANSION.
C S_OLD_PROD, S_NEW_PROD - TEMPORARIES FOR SCALAR PRODUCTS USED
C                          IN THE SALPHA AND SBETA CALCULATIONS.
C MLONG_INDEX - HOLDS THE LONG VECTOR ORDERING OF THE PROCESSOR
C               ARRAY. THIS ORDERING ASSIGNS EACH PROCESSOR
C               A NUMBER IN THE RANGE [1..4096], WITH THE FIRST
C               COLUMN GETTING VALUES [1..64], THE SECOND COLUMN
C               VALUES [65..128] ETC.
C PLACE - USED TO GENERATE MLONG_INDEX.
C VTEMP - USED TO GENERATE MLONG_INDEX.
C SFIRST - INDICATES FIRST CALL TO SPARSE_MULTIPLY.
C VCOUNT - HOLDS NUMBER OF BROADCASTS NEEDED FOR EACH LAYER
C          OF THE SYSTEM MATRIX.
C
C*********************************************************************
C
C                          EXTERNAL SUBROUTINE
C
      EXTERNAL SPARSE_MULTIPLY
C
C     WHICH ROUTINE SHOULD BE USED?
C
      IF (SPRECON_ORDER.NE.0) GOTO 1000
C
C*********************************************************************
C
C     CONJUGATE GRADIENT ROUTINE (NO PRECONDITIONING)
C
C*********************************************************************
C
C     INITIALIZE VARIABLES.
C     SET UP SFIRST FOR FIRST CALL TO SPARSE_MULTIPLY SO THAT
C     VCOUNT WILL BE GIVEN PROPER COMPONENTS
C
      SNUMITER=0
      LVX=1
      SFIRST = .TRUE.
      VCOUNT = 0
      CALL SPARSE_MULTIPLY(SN,SP,SMAXPLANES,MA,MCOLUMNS,
     &                     LVX,LVANS,VCOUNT,SFIRST)
C     SET SFIRST TO FALSE SO SPARSE_MULTIPLY WILL NOT
C     RE-EVAULATE VCOUNT.
      SFIRST = .FALSE.
      LVR = LVB - LVANS
      LVD = LVR
      S_OLD_PROD = SUM(LVR*LVR)
C
C     BEGIN CG ITERATION
C
5     CONTINUE
      CALL SPARSE_MULTIPLY(SN,SP,SMAXPLANES,MA,MCOLUMNS,
     &                     LVD,LVANS,VCOUNT,SFIRST)
      SALPHA = S_OLD_PROD/SUM(LVD*LVANS)
      LVX = LVX + SALPHA*LVD
      LVR = LVR - SALPHA*LVANS
      S_NEW_PROD = SUM(LVR*LVR)
      SBETA = S_NEW_PROD/S_OLD_PROD
      S_OLD_PROD = S_NEW_PROD
      LVD = LVR + SBETA*LVD
      SNUMITER=SNUMITER+1
C     CHECK EXIT CRITERION. STOP ITERATING IF MET
      IF (SQRT(SUM(LVR*LVR)) .GT. SEXIT) GOTO 5
      GOTO 2000
1000  CONTINUE
C
```

```
C*********************************************************************
C
C                 POLYNOMIAL PRECONDITIONED C.G. ROUTINE
C
C*********************************************************************
C
C      PRODUCE AN INTEGER MATRIX WHOSE ENTRIES
C      CORRESPOND TO LONG-VECTOR ORDER 1 TO 4096
C      FROM INTRO TO DAP FORTRAN PROGRAMING PG. 5-5
C      USE THIS TO OBTAIN THE DIAGONAL OF THE SYSTEM MATRIX
C
       VTEMP=0
       PLACE=1
       DO 10 K=1,6
       VTEMP(ALT(PLACE)) = VTEMP + PLACE
10     PLACE = PLACE*2
       MLONG_INDEX = MATC(VTEMP) + MATR(64*VTEMP) + 1
C
C      NOW LOAD DIAGONAL ENTRIES OF SYSTEM MATRIX INTO A LONG
C      VECTOR AND INVERT THE RESULTING VECTOR.
C      THIS VECTOR IS THE BASIS OF  POLYNOMIAL PRECONDITIONING
C
       LVDIAG_INV = 0.0
       DO 20 I=1,SP
         MTEMP=MCOLUMNS(,,I)
         LVDIAG_INV(MTEMP.EQ.MLONG_INDEX) = MA(,,I)
20     CONTINUE
       LVDIAG_INV = 1.0/LVDIAG_INV
C
C      INITIALIZE VARIABLES.
C      SET UP SFIRST FOR FIRST CALL TO SPARSE_MULTIPLY SO THAT
C      VCOUNT WILL BE GIVEN PROPER COMPONENTS
C
       SNUMITER=0
       LVX=1
       SFIRST = .TRUE.
       VCOUNT = 0
       CALL SPARSE_MULTIPLY(SN,SP,SMAXPLANES,MA,MCOLUMNS,
      &                    LVX,LVANS,VCOUNT,SFIRST)
C      SET SFIRST TO FALSE SO SPARSE_MULTIPLY WILL NOT
C      RE-EVAULATE VCOUNT.
       SFIRST = .FALSE.
       LVR = LVB - LVANS
C
C      PERFORM POLYNOMIAL EVALUATION FOR PRECONDITIONING
C
       LVFIRST_TERM = LVDIAG_INV*LVR
       LVRESULT = LVFIRST_TERM
       SLOOPLIM = (SPRECON_ORDER - 1)
       IF (SLOOPLIM.EQ.0) GOTO 40
       DO 30 I = 1, SLOOPLIM
         CALL SPARSE_MULTIPLY(SN,SP,SMAXPLANES,MA,MCOLUMNS,
      &                   LVRESULT,LVANS,VCOUNT,SFIRST)
30       LVRESULT = LVFIRST_TERM + LVRESULT - LVDIAG_INV*LVANS
40     CONTINUE
       LVD=LVRESULT
       S_OLD_PROD = SUM(LVR*LVRESULT)
C
C      BEGIN PPCG ITERATION
C
50     CONTINUE
       CALL SPARSE_MULTIPLY(SN,SP,SMAXPLANES,MA,MCOLUMNS,
      &                    LVD,LVANS,VCOUNT,SFIRST)
       SALPHA = S_OLD_PROD/SUM(LVD*LVANS)
       LVX = LVX + SALPHA*LVD
       LVR = LVR - SALPHA*LVANS
```

```
C
C     PERFORM POLYNOMIAL EVALUATION FOR PRECONDITIONING
C
      LVFIRST_TERM = LVDIAG_INV*LVR
      LVRESULT = LVFIRST_TERM
      IF (SLOOPLIM.EQ.0) GOTO 70
      DO 60 I = 1, SLOOPLIM
        CALL SPARSE_MULTIPLY(SN,SP,SMAXPLANES,MA,MCOLUMNS,
     &                   LVRESULT,LVANS,VCOUNT,SFIRST)
60      LVRESULT = LVFIRST_TERM + LVRESULT - LVDIAG_INV*LVANS
70    CONTINUE
      S_NEW_PROD = SUM(LVR*LVRESULT)
      SBETA = S_NEW_PROD/S_OLD_PROD
      S_OLD_PROD = S_NEW_PROD
      LVD = LVRESULT + SBETA*LVD
      SNUMITER=SNUMITER+1
C     CHECK EXIT CRITERION. STOP ITERATING IF MET
      IF (SQRT(SUM(LVR*LVR)) .GT. SEXIT) GOTO 50
2000  CONTINUE
      RETURN
      END
```

Appendix B

SPARSE MULTIPLY ROUTINE

```
      SUBROUTINE SPARSE_MULTIPLY(SN,SP,SMAXPLANES,MA,MCOLUMNS,
     &                         LVMULCAND,LVANS,VCOUNT,SFIRST)
C
C*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*
C
C          PARKINSONS ALGORITHM FOR SPARSE MATRIX-VECTOR MULT.
C
C*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*
C
C                        PARAMETER DICTIONARY
C
C*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*
C
C SN - DIMENSION LINEAR SYSTEM
C SP - NUMBER OF PLANES OCCUPIED BY MA
C SMAXPLANES - NUMBER OF STORAGE PLANES TAKEN BY MA
C              MAXIMUM VALUE FOR SP
C MA - MATRIX OF COEFFICIENTS STORRED ONE ROW PER PROCESSOR
C MCOLUMNS - GIVES COLUMN NUMBER OF CORRESPONDING MA ELEMENT
C LVMULCAND  - MULTIPLICAND
C LVANS - RESULT
C VCOUNT - NEEDED FOR EFFICIENTCY OF ROUTINE
C          VCOUNT(I) CONTAINS THE NUMBER OF BROADCASTS NEEDED
C          TO PROCESS ONE PLANE OF THE COEFFICIENT MATRIX.
C          IF VCOUNT(I) > 500, IT IS CHEAPER TO USE PERMUTE.
C SFIRST - INDICATES THE FIRST CALL OF THIS PROCEDURE
C
C*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*
C
      INTEGER SN,SP,SMAXPLANES
      REAL MA(,,SMAXPLANES)
      INTEGER MCOLUMNS(,,SMAXPLANES), MCOLUMNSET(,), VCOUNT()
      INTEGER MTEMPINT(,)
      REAL LVMULCAND(,), LVANS(,), MTEMP(,)
      LOGICAL SFIRST, MTEST(,)
      INTEGER KEY(,), IFAIL
C
C*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*
C
C                        VARIABLE DICTIONARY
C
C*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*
C
C MCOLUMNSET - TEMPORARY THAT HOLDS A LAYER OF THE MCOLUMNS ARRAY.
C MTEMPINT - HOLDS RESULT FROM SORT SUBROUTINE.
C MTEMP - HOLDS RESULT OF PERMUTING LVMULCAND VIA BROADCASTING
C         OR THE PERMUTAION SUBROUTINE.
C MTEST - LOGICAL MATRIX USED FOR MASKED ASSIGNMENT WHEN BROADCASTING.
C KEY - RETURNS THE PERMUTATION NEEDED TO EFFECT THE SORT PRODUCED
C       BY MO1_SORTILV.
C IFAIL - ERROR INDICATOR.
C
```

```
C*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*
C
C         THESE SUBROUTINES AVAILABLE ON DAP SUBROUTINE LIBRARY
C
C*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*
C
C     MO1_PERMUTE_1 - GENERATES A NON-UNIQUE PERMUTATION OF AN INTEGER
C                     LONG VECTOR.  EFFECTS THE PERMUTATION
C
C                     RESULT( I ) = SOURCE_VECTOR( KEY(I) )
C
C                     WHERE MCOLUMNSET IS A GIVEN PLANE OF MCOLUMNS
C
C     MO1_SORTILV - SORTS AN INTEGER LONG VECTOR.
C                   USED TO DETERMINE HOW MANY BROADCASTS ARE
C                   NEEDED FOR A GIVEN PLANE OF THE SYSTEM MATRIX.
C
C*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*+*
C
      IF (SFIRST.LEQ..FALSE.) GOTO 50
C
C     IF THIS IS THE FIRST CALL, COUNT NUMBER OF BROADCASTS
C     NEEDED FOR EACH LAYER OF THE SYSTEM MATRIX
C
      DO 10 L=1,SP
C     PICK OFF FIRST LAYER OF THE COLUMN POINTER ARRAY
C     SORT IT AND COUNT HOW MANY DIFFERENT NUMBERS ARE IN THE LIST
C     THIS IS THE NUMBER OF BROADCASTS THAT NEED TO BE DONE.
      MCOLUMNSET=MCOLUMNS(,,L)
      CALL MO1SORTILV(MCOLUMNSET,MTEMPINT,SN,.TRUE.,KEY,IFAIL)
10    VCOUNT(L) = SUM(MTEMPINT.NE.MTEMPINT(+,))
50    CONTINUE
      LVANS=0
C     NOW PERFORM ACTUAL MULTIPLICATION. PROCEDE ONE LAYER AT A TIME
C     THROUGH THE MATRIX STORAGE ARRAY.
      DO 100 L=1,SP
C     IF NUMBER OF BROADCASTS FOR THIS LAYER > 500 PERMUTE IS CHEAPER
      IF (VCOUNT(L) .GT. 500) GOTO 150
C
C     PERFORM BROADCAST ASSIGNMENTS
C
      MTEMP = 0.0
      MCOLUMNSET = MCOLUMNS(,,L)
C     ONLY WORK WITH VALUES THAT EXIST IN MCOLUMNS
C     I.E. BETWEEN THE MAXIMUM AND MINIMUM COLUMN POINTERS
      DO 75 I = MINV(MCOLUMNSET),MAXV(MCOLUMNSET)
C     BUILD A LOGICAL MASK MATRIX CONTAINING .TRUE. WHEREVER A
C     VALUE OF 'I' IS PRESENT IN MCOLUMNSET
      MTEST = MCOLUMNSET .EQ. I
C     BROADCAST MULTIPLICAND VALUES TO PROPER POSITION IN
C     TEMPORARY MATRIX.
75    IF (ANY(MTEST)) MTEMP(MTEST) = LVMULCAND(I)
      GOTO 100
C
C     PERFORM PERMUTATION WHERE MCOLUMNS(,,L) IS THE 'KEY'
C     LVMULCAND IS THE VECTOR TO BE PERMUTED, AND
C     MTEMP IS THE LONG VECTOR RESULT.
C
150   CALL MO1PERMUTE1(MTEMP,LVMULCAND,MCOLUMNS(,,L))
C
C     NOW MULTIPLY ACCUMULATED MATRIX WITH THE CURRENT
C     LAYER OF THE COEFFICIENT MATRIX.
C
100   LVANS = LVANS + MA(,,L)*MTEMP
      RETURN
      END
```

REFERENCES

Axelsson, O., "Solution Of Linear Systems of Equations: Iterative Methods", in Sparse Matrix Techniques (Lecture Notes in Mathematics #572). Barker, V.A. (ed.), Springer-Verlag, Berlin, Germany, pp. 1-51, 1977.

Conway, L. and Mead, C., Introduction to VLSI Systems. Addison-Wesley Inc., Don Mills, Ontario, 1980.

Cornell, J.A., "PEPE: Parallel Element Processing Ensemble" in Multiprocessing Systems - Infotech State of the Art Report. White, C.H. (ed.), Infotech International Ltd., Nicholson House, Maidenhead, Berkshire, England, 1976.

Davis, P.J., Interpolation and Approximation. Dover Publications Inc., New York, 1975.

Dubois, P.F., Greenbaum, A., and Roodrigue, G.H., "Approximating the Inverse of a Matrix for Use in Iterative Algorithms on Vector Processors", in Computing, Vol. 22, pp. 257-268, 1979.

Flynn, M.J., "Some Computer Organizations and Their Effectiveness", in IEEE Transactions on Computers, C-21 (9), pp. 948-960, 1972.

Gajski, D.D., Padua, D.A., Kuck, D.J., and Kuhn, R.H., "A Second Opinion on Data Flow Machines and Languages", in IEEE Computer, Vol. 15(2), pp. 58-69, 1982.

Gottlieb, A., and Schwartz, J.T., "Networks and Algorithms for Very-Large-Scale Parallel Computation", in IEEE Computer, Vol. 15(1), pp. 27-36, 1982.

Hestenes, M.R., "The Conjugate-gradient Method for Solving Linear Systems", in Proceedings of the Sixth Symposium in Applied Mathematics of the American Mathematical Society (Curtiss J.H. (ed)). McGraw-Hill Book Company, Inc., Toronto, 1956.

Hockney, R.W. and Jesshope, C.R., Parallel Computers. Adam Hilger Ltd., Bristol, 1981.

ICL, DAP: APAL Language. ICL technical publication TP 6919, 1979.

ICL, DAP: Introduction to FORTRAN Programming. ICL technical publication TP 6755, 1980.

ICL, DAP: FORTRAN Language. ICL technical publication TP 6918, 1981a.

ICL, DAP: Developing DAP Programs. ICL technical publication TP 6920, 1981b.

IEEE Computer, special issue on Data Flow Systems, Vol. 15(2), 1982.

Johnson, O.G., Micchelli, C.H., and Paul, G., "Polynomial Preconditioners for Conjugate Gradient Calculations", in Siam J. Numerical Analysis, Vol 20 (2), pp. 362-376, April 1983.

Kershaw, D.S., "The Incomplete Cholesky-Conjugate Gradient Method for the Iterative Solution of Systems of Linear Equations", in Journal of Computational Physics, Vol. 26, pp. 43-65, 1978.

Kung, H.T., "Why Systolic Architectures?", in Computer, Vol. 15(1), pp. 37-46, 1982.

Lang, S., Linear Algebra. Addison-Wesley Publishing Company, Don Mills, Ontario, 1972.

Liddel, H.M., and Bowgen, G.S.J., "The DAP Subroutine Library", in Computer Physics Communications, Vol. 26, pp. 311-315, 1982.

Madsen, N.K., Rodrigue, G.H., and Karush, J.I., "Matrix Multiplication by Diagonals on a Vector/Parallel Processor", in Information Processing Letters, Vol. 5(2), pp. 41-45, 1976.

Meijerink, J.A., and van der Vorst, H.A., "An Iterative Solution Method for Linear Systems of Which the Coefficient Matrix is a Symmetric M-Matrix", in Mathematics of Computation, Vol. 31(137), pp. 148-162, Jan. 1977.

Meilander, W.C., "STARAN, An Associative Approach to Multiprocessor Architecture", in Multiprocessing Systems - Infotech State of the Art Report. White, C.H. (ed.), Infotech International Ltd., Nicholson House, Maidenhead, Berkshire, England, 1976.

Mirsky, L., An Introduction to Linear Algebra. Dover Publications, Inc., New York, 1982.

Nakonechny, R.L., A Preconditioned Conjugate Gradient Method Using a Sparse Linked-List Technique For the Solution of Field Problems. Masters Thesis, The University of Manitoba, Winnipeg, Manitoba, 1983.

Parkinson, D., Sparse Matrix Vector Multiplication on the DAP. DAP Support Unit Paper #2.11, Queen Mary College, London, England, 1981.

Parkinson, D., "Practical parallel processors and their uses", in Parallel processing systems. Evans, D.J. (ed.), Cambridge University Press, New York, 1982.

Ramamoorthy, C.V. and Li, H.F., "Pipeline Architecture", in Computing Surveys, Vol. 9(1), pp. 62-102, March 1977.

Schaefer, D.H. and Fischer, J.R., "Beyond the Supercomputer", in IEEE spectrum, Vol. 19(3), pp. 32-37, March 1982.

Shore, J.E., "Second thoughts on parallel processing", in Computers and Electrical Engineering, Vol. 1, pp. 95-109, 1973.

Snyder, L., "Introduction to the Configurable, Highly Parallel Computer", in Computer, Vol. 15(1), pp. 47-56, 1982.

Varga, R.S., Matrix Iterative Analysis. Prentice-Hall, Inc., Toronto, 1962.

Webb, S.J., McKeown, J.J., and Hunt, D.J., "The Solution of Linear Equations on a SIMD Computer Using a Parallel Iterative Algorithm", in Computer Physics Communications, Vol. 26, pp. 325-329, 1982.

Wexler, A., Finite Elements for Technologists. Department of Electrical Engineering Technical Report TR-80-4, University of Manitoba, Winnipeg, Manitoba, Canada, 1980.

Yau, S.S. and Fung, H.S., "Associative Processor Architecture-A Survey", in Computing Surveys, Vol. 9(1), pp. 3-27, 1977.