Systolic arrays for the matrix iterative methods

Haider, Shahid A.. 2019. “Systolic Arrays for the Matrix Iterative Methods”. figshare. https://hdl.handle.net/2134/28173.

SYSTOLIC ARRAYS FOR THE MATRIX ITERATIVE METHODS

By

Shahid Abbas Haider, BEng., MSc.

A Doctoral Thesis submitted in partial fulfillment
of the requirements for the
Award of the Degree of
Doctor of Philosophy
of Loughborough University of Technology

April 1993

SYSTOLIC ARRAYS FOR THE MATRIX ITERATIVE METHODS

By

Shahid Abbas Haider, BEng., MSc.

A Doctoral Thesis submitted in partial fulfillment
of the requirements for the
Award of the Degree of
Doctor of Philosophy
of Loughborough University of Technology

April 1993

CERTIFICATE OF ORIGINALITY

This is to certify that I am responsible for the work submitted in this thesis, that the original work is my own except as specified in acknowledgements or in footnotes, and that neither the thesis nor the original work contained therein has been submitted to this or any other institution for a higher degree.

Shahid Abbas Haider

ACKNOWLEDGEMENTS

I wish to express my sincere thanks, gratitude and appreciation to my supervisor, Professor D. J. Evans, Director, Parallel Algorithm Research Centre (PARC), for his constant advice, guidance, interest and above all, encouragement throughout my research, as well as his endeavor in reading the manuscript of this thesis.

I express my thanks to Professor J. L. Alty, Head of the Computer Studies Department, for his advice and interest; to the staff of the Computer Studies Department and especially PARC; and to the research students at PARC.

I would also like to convey my deep gratitude to the Ministry of Science and Technology, Government of Pakistan, for their financial support without which this thesis would have never been completed.

Eternal thanks to my parents, brothers and sisters, for their immense love, encouragement and their sound belief in my success, without which this work would have been inconceivable.

Abstract

The systolic array research was pioneered by H. T. Kung and C. E. Leiserson. Systolic arrays are special purpose synchronous architectures consisting of simple, regular and modular processors which are regularly interconnected to form an array.

Systolic arrays are well suited for computational bound problems in Linear Algebra. In this thesis, the numerical problems, especially iterative algorithms are chosen and implemented on the linear systolic array.

Iterative methods perform a sequence of repetitive steps to obtain a new approximation which converge to a solution. The accuracy of the method depends on the number of iterations performed. The iterative process is terminated when the difference between the successive approximation satisfies some tolerance.

Several iterative methods like the Jacobi, Gauss-Seidel, S.O.R., S.S.O.R., A.O.R., M.S.O.R. etc. are mapped on to a linear systolic array. The systolic designs are simulated in OCCAM.

Problems involving rates of change of two or more independent variables representing some physical quantity leads to a partial differential equation. These equations can be discretised and then solved by applying the iterative methods. The 2-dimensional and 3-dimensional problems are taken as example and are discretised. The discretised problems generate sparse matrix systems. The concept of a Virtual IPS cell is introduced to cope with wide sparse matrix systems. Using these cells, a linear systolic array is simulated and several iterative algorithms are used to solve PDE's by the second order Richardson iterative method, S.O.R., Steepest Descent, Conjugate Gradient and Preconditioned Conjugate Gradient methods. The concept of Virtual IPS cell reduces the area requirements of the array but the computational time remains the same.

Contents

List of Figures

List of Tables

Chapter 1

Introduction and Fundamentals of Parallel Computer Architectures

This chapter provides introduction to the thesis, an overview of parallel processing and different system organizations. Increased computational applications, new computer technologies and novel concepts in system organizations have lead to an increasing interest in highly powerful and reliable computer architectures. Several approaches have been developed to increase the power and throughput of the conventional Von Neumann architecture. Pipelining is used to achieve concurrency on a time overlap basis. The idea of having several memory units instead of one, is used to fetch the operands concurrently from separate memory units, thus improving throughput. The use of more than one processor to solve large problems fast gave birth to the field of parallel processing.

1.1 Introduction

The solution of many important problems related to scientific and engineering applications require solving large systems of linear equations. Examples include fluid dynamics, weather predictions, aircraft design, structural problems and many others. These systems arise from modeling with partial differential equations by the use of finite difference/element methods. Thus, solving large systems of equations has been a central issue in numerical methodology and analysis. Two basic families of methods for solving systems can be distinguished: direct and iterative. There are times that one should prefer to use a direct method but the opposite is also true.

Besides being important from this point of view, developments in computer technology bring additional interest to the study of iterative methods. Although some well known iterative schemes are very old, it was the growth of digital computers and their increased use in solving the partial differential equations that triggered considerable new interest in the area. Thus, the fifties and sixties became the period in which most of the significant results on iterative methods were produced. Memory limitations, that adversely affect direct methods, were the main reason for this. The memory and speed of computers have increased dramatically since then, but so has the computational needs of researchers and developers, reaching the point of requiring the solution of systems with millions of unknowns. The development of parallel and vector computers changed the nature of numerical algorithms. Computer algorithms and architectures became closely related and affect each other. Then in the new super computer architectures, communication costs and bottlenecks became a major limitation in the efficient implementation of algorithms. The developments in micro-electronics have revolutionised computer design. Integrated circuit technology has increased the number and complexity of components that can fit on a chip. The Very Large Scale Integrated (VLSI) technology makes it feasible to build low-cost special purpose designs to rapidly solve sophisticated problems. Iterative methods seem to behave considerably better than direct methods, thus attracted renewed interest.

This work is motivated not only by the importance of the iterative methods, as explained above, but also by the unresolved issues that these methods present.

With the exception of the Jacobi iterative method (which, when it converges, converges slowly), known iterative methods do not present inherent parallelism. One solution to this problem requires modification of existing methods or to discover new methods with increased parallelism. The other solution is to select existing computer architectures or design new ones that are most suitable for the implementation of parallel iterative methods. The designs developed will have different hardware requirements and execution times. So the design may be implementable but too expensive (difficult) to fabricate. Even if some methods are implementable, convergence may be too slow. Of the many methods one often does not know which one will converge significantly fast and definitely one will prefer to use the fastest possible. This leads to discover better implementation schemes in terms of parallelism. The issues related to the practical implementation will not be dealt with in this thesis.

Rest of this chapter briefly describes the different parallel computer organisations.

Chapter 2, contains basic mathematical definitions in the areas of numerical solution of the systems of linear equations and the discrete approximation to the solution of partial differential equations. Some basic matrix iterative methods are explained briefly as well.

Chapter 3, is an introduction to the basic concepts of the systolic approach in parallel processing. It starts with a description of the basic building block of the systolic design. Later in the chapter, the matrix vector multiplication example is presented on a linear systolic array, proposed by H. T. Kung. Then different techniques to improve the utilisation of the array, and to decrease the chip area are explained. We implement these new ideas to the basic iterative methods and describe their usefulness.

Chapter 4, describes the systolic implementation of the S.O.R. and S.S.O.R. iterative methods on the linear systolic array. The Conrad-Wallach strategy is implemented in order to reduce the iteration time of the array and hence the computational work is reduced to that of the S.O.R. iterative method.

Chapter 5, describes the first and second order stationary and nonstationary Richardson iterative methods. The A.O.R. iterative method and the S.A.O.R. iterative method are also described. The systolic implementation of these methods are developed. A similar strategy as Conrad-Wallach is applied to the S.A.O.R. iterative method. Different pipelining techniques to improve the utilisation are shown and explained.

Chapter 6, presents the systolic designs for the 2-dimensional and 3-dimensional problems obtained from the discretisation of partial differential equations. The discretised problems are solved using the J.O.R., S.O.R. and second order Richardson (with and without Chebyshev acceleration) iterative methods. Later we explain the Conjugate Gradient and Preconditioned Conjugate Gradient methods and implement them systolically.

Chapter 7, completes this thesis with a review of the main results and some general conclusions, that reflect the research areas mentioned in this thesis. A list of references is also given, consisting of material covering a wide spectrum of research interests in the systolic systems and computing.

1.2 Architectural configuration of parallel computers

The Von Neumann computer model (classical sequential model) of computation consists of a stream of instructions and a stream of data, executing one instruction at a time, to produce a computational result, as shown in figure (1.1). The Von Neumann architecture is improved by using multiple proces-

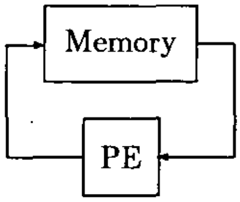


Figure 1.1: Von Neumann computer model.

sors that can communicate and cooperate, which gave birth to the field of parallel processing (see Almasi and Gottlieb [1]). Parallel computers can be classified according to their architectures namely; Pipelined computers, Array processors, Multiprocessor systems, Data flow computers and VLSI algorithmic processors. Data flow computers and VLSI algorithmic processor architectures demand extensive hardware to achieve parallelism. The revolution brought by the VLSI technology has made these two approaches an interesting research area.

1.2.1 Pipelined computers

Pipelined computers have emerged and received considerable attention as an attribute and an economical way of speeding up computer systems. Pipelining offers an economical way to realise "temporal" parallelism in digital computers. The pipelining approach is based on the fact that the execution of many machine instructions consumes several clock periods, usually using the same hardware iteratively. If such hardware is replicated serially, then a number of operations may be streamed into the processors at the same time and be executed in an overlapped fashion. Hence input tasks (processes) must be divided into a sequence of subtasks, each of which can be executed by a specialised hardware stage that operates concurrently with other stages in the pipeline. Therefore, a pipeline processor can be described as consisting of a sequence of processing circuits, called segments or stages, through which the data stream

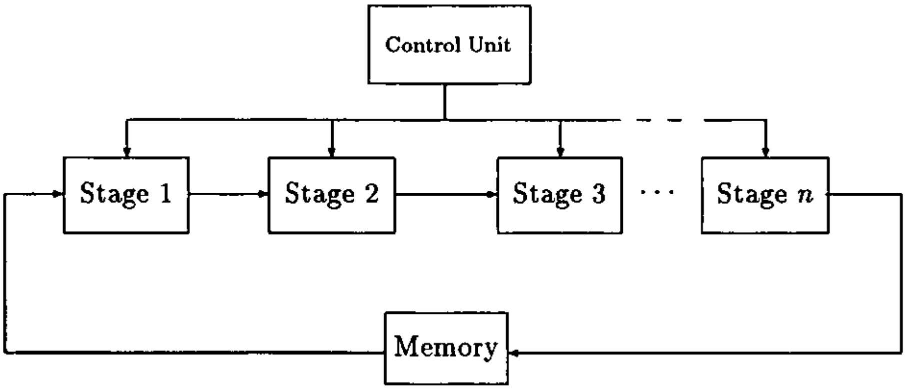


Figure 1.2: n-stage pipelined computer.

passes. The data is processed partially by each segment and the final result is obtained after the data has passed through all the segments of the pipeline. The result stream returns to memory. Here parallelism is achieved by having distinct operand sets or processes manipulated in several segments at the same time (see Hennessy and Patterson [38], Baer [3]). Figure (1.2) shows a schematic diagram of an n-stage pipelined machine. Due to the overlapped instruction and arithmetic execution, it is obvious that pipeline machines are better tuned to perform the same operations repeatedly through the pipeline. Whenever there is a change of operation, say from "add" to "multiply", the arithmetic pipeline must be drained and reconfigured, which will cause extra time delays. Therefore, pipeline computers are more attractive for vector processing, where component operations may be repeated many times such as in matrix computations.

Now to explain how parallelism can be achieved by pipelining, let us consider the process of executing an instruction. Usually the process of executing an instruction in a digital computer involves four major steps: fetching the instruction (IF) from the main memory; decoding it (ID) to identify the operation to be carried; fetching the operand(s) (OF) if required; and finally the execution (EX) of the decoded arithmetic logic operation. If this process is to be carried out in a non-pipelined computer, then these four steps must be

Instruction i				Instruction i+1				Instruction i+2				Instruction i+3			
IF	ID	OF	EX	IF	ID	OF	EX	IF	ID	OF	EX	IF	ID	OF	EX

Figure 1.3: Non-pipelined execution.

Instruction number	Pipe Cycle						
	1	2	3	4	5	6	7
i	IF	ID	OF	EX			
i+1		IF	ID	OF	EX		
i+2			IF	ID	OF	EX	
i+3				IF	ID	OF	EX

Figure 1.4: Pipelined execution.

completed before the next instruction can be issued, as can be seen from the space-time diagram in figure (1.3) for the non-pipelined processor.

In a pipelined computer, successive instructions are executed in an overlapped fashion, by the four pipelined stages, IF, ID, OF and EX, which are arranged into a linear cascade (see figure 1.4).

An instruction cycle consists of multiple pipeline cycles. A pipeline cycle can be set equal to the delay of the slowest stage. The operation of all stages is synchronised under a common clock control. For non-pipelined computers it takes four pipeline cycles to complete one instruction. Once a pipeline is filled up, an output result is produced from the pipeline on each cycle. The instruction cycle has been effectively reduced to one-fourth of the original cycle time by such overlapped execution. According to the levels of processing Handler, in [36], classified the pipeline processors into three classes which are:

- Arithmetic pipelining: These processors have been constructed for performing either a single arithmetic function or the four basic operations on both fixed- and floating-point numbers.
- Instruction pipelining: The purpose of such a pipeline is to overlap the execution of the current instruction with the subsequent instruction stages. This is done by the decomposition of instruction execution into a linear series of autonomous stages e.g. fetch, decode etc.
- Processor pipelining: In this type of pipelining, a cascade of processors, each with a specific task, process the same data stream. The first processor manipulates the passing data stream and the results are stored in a memory block which is accessible by the second processor. The 2^{nd} processor passes the refined data stream to the third processor and so on.

The first two classes are "low-level parallelism" and hence are excluded from the set of parallel architecture. There are two reasons for such exclusion; the first one being that the failure to adopt a more restrictive standard might make the majority of modern computers "parallel architecture" negating the term's usefulness. The second reason is that architectures having only these features do not offer an explicit, coherent framework for developing high-level parallel solutions. Theoretically, the maximum speed-up that can be gained from a pipeline processor with n-stages is n, i.e. when the pipeline is full. However, in practice this perfect speed-up cannot be achieved due to memory conflicts, data dependency, program branches and interrupt handling. Some of the most obvious machines that use pipelining technology are: IBM 360/195, CDC STAR, and the University of Manchester MU5 computer.

1.2.2 Array processors

An array processor can be defined as a synchronous parallel computer with multiple arithmetic logic units called processing elements (PE's). Those processing elements are controlled by a single control unit (CU) i.e. processor only operates on command from control unit. Each PE is independent i.e. has its own registers, an arithmetic and logical unit (ALU) and a local memory. "Spatial" parallelism can be achieved by the replication of the ALU's.

Two essential reasons were behind the idea of building array processors. The first is an economic reason for it is cheaper to build n processors with only one control unit rather than n similar computers. The second reason is that the communication bandwidth, of the interprocessor communication, can be fully utilised.

The PE's are passive devices without instruction decoding capabilities and they are synchronised to perform the same function at the same time. The control unit fetches and decodes the instructions and broadcasts the data, i.e. vectors to the PE's for distributed execution over different components. Operands are fetched directly from the local memories. Scalar and control instructions are directly executed in the control unit itself.

Different array processors may use different interconnecting patterns among the PE's. In order to maximise the parallelism in an array processor, utilisation of the available memory and processor bandwidths must be very high.

An example of an array processor is the ICL DAP computer. Figure (1.5) is a schematic diagram of the general structure of this machine.

The ICL DAP has simple identical processing elements (upto 4096 processors, arranged in a (64 × 64) grid). A single master processor broadcasts program instructions to individual PEs which execute the same instruction on their respective data items. In the DAP, the PEs are connected in a square array. Each PE can simultaneously shift one bit in one direction and receive a bit from the opposite direction. Repeated shifts can move large volumes of data

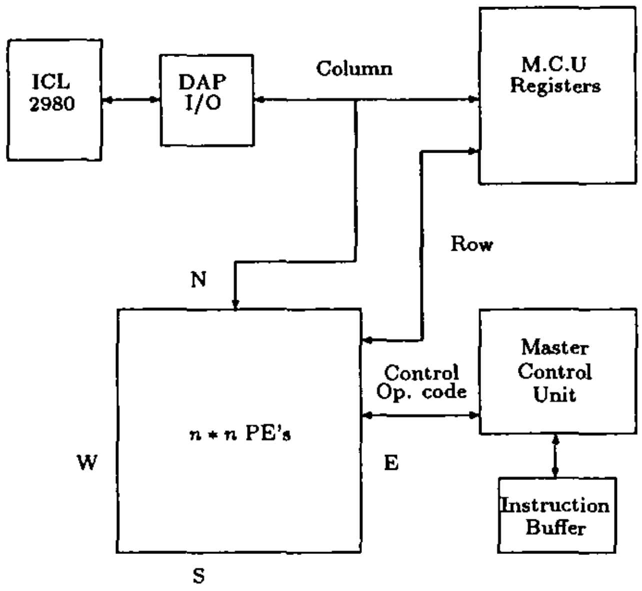


Figure 1.5: ICL DAP architecture.

from any part of the grid to any other. This is a basic example of a vector or array processor.

Speedup is achieved compared to the Von Neumann model as there is no separate instruction fetch for each data item, and the execution of the data items is done in parallel. This is an expensive solution as multiple processing elements are required. The operational speed of an array processor is supposed to increase linearly as the number of the PE's is increased. However, this is not the case due to the interprocessor communication and the data access overheads.

Some of the application areas that have been suggested as suitable for array processors and in particular for the Illiac IV, the BSP, the MPP and the STARAN systems include: Matrix algebra (multiplication, decomposition, inversion), matrix eigenvalue calculation, linear and integer programming, weather modeling, beam forming and convolution, filtering and Fourier analysis, image processing and pattern recognition, wind-tunnel experiments, automated map generation, and real-time scene analysis.

The above list is by no means exhaustive. Most of these applications need to process spatially distributed data.

1.2.3 Multiprocessor systems

Multiprocessor systems have been developed as part of the research efforts aimed to improve the speed, reliability, throughput and availability of computer systems. A multiprocessor system is defined in the American National Standard Vocabulary of Information Processing as "a computer employing two or more processing units under integrated control". This definition is not complete as the concept of sharing and interaction, which are at the core of the techniques of multiprocessing are not included in the ANSI definition. Enslow, in [13], defines the multiprocessor system which has the following characteristics.

A multiprocessor system contains two or more processors of approximately comparable capabilities. All processors share access to common memory modules, I/O channels and peripheral devices. Most importantly, the entire system is controlled by one operating system which provides interactions between processors and their programs at various levels. Each processor has its local memory and private devices. Figure (1.6) shows a basic multiprocessor organisation.

Processors, in these systems can communicate with each other, either through the shared memory or through an interrupt network. This interconnecting structure between the memories and the processors and between memories and I/O channels is the primary determination of multiprocessor hardware system organisation.

Among these interconnecting structures, three fundamental types have been identified by Hwang and Briggs, in [41], to be used in multiprocessors.

1. **Time-shared/Common bus**

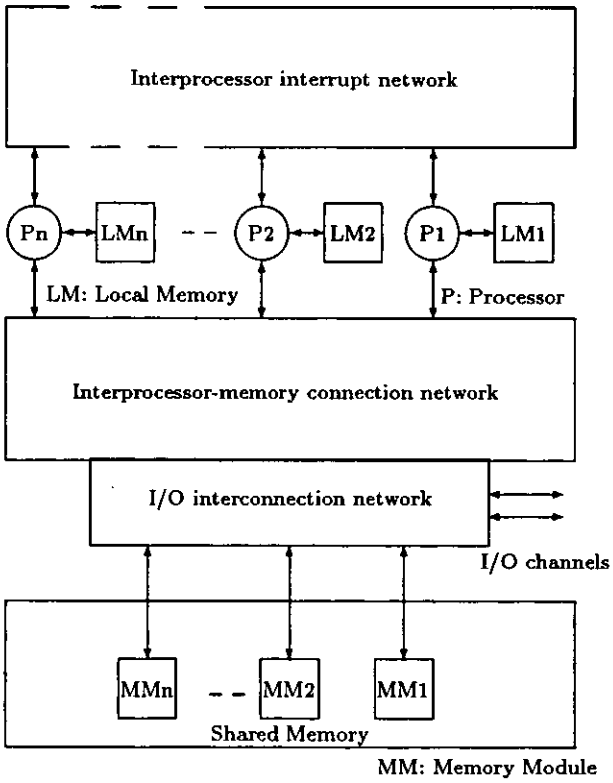

Interprocessor interrupt network
Pn
LMn
P2
LM2
P1
LM1
LM: Local Memory
P: Processor
Interprocessor-memory connection network
I/O interconnection network
I/O channels
MMn
MM2
MM1
Shared Memory
MM: Memory Module

Figure 1.6: Multiprocessor architecture.

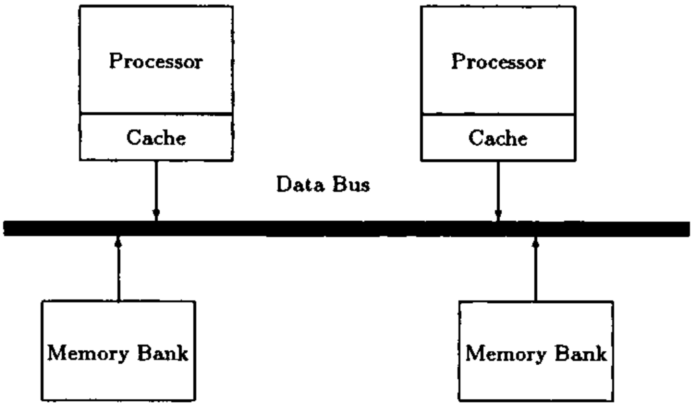


Figure 1.7: Time-shared bus (single bus) multiprocessor.

This is the simplest interconnecting structure for either single or multiple processors system. It consists of a common communication path (single bus) connecting all of the functional units, in which each one of them consists of a number of processors, memories and I/O devices.

The time shared bus involves the lowest overall system cost for hardware as the bus can be a simple multiconductor cable, and hence is least complex. The system configuration can be physically modified simply by adding or removing the functional units. However, expanding the system by the addition of functional units may degrade the overall system performance and hence is suitable for small systems. The major draw back of this organisation is that a bus failure halts the entire system. Figure (1.7) illustrates a general scheme of a common bus multiprocessor system.

2. **Cross-bar switch network**

   To overcome the major drawback of the time-shared bus organisation mentioned earlier, the crossbar switch networks are used.

   The crossbar interconnecting technology uses a crossbar switch of PxM cross points to connect P processors to M memories. Thus, the interconnections between processors and memory units are increased in such a

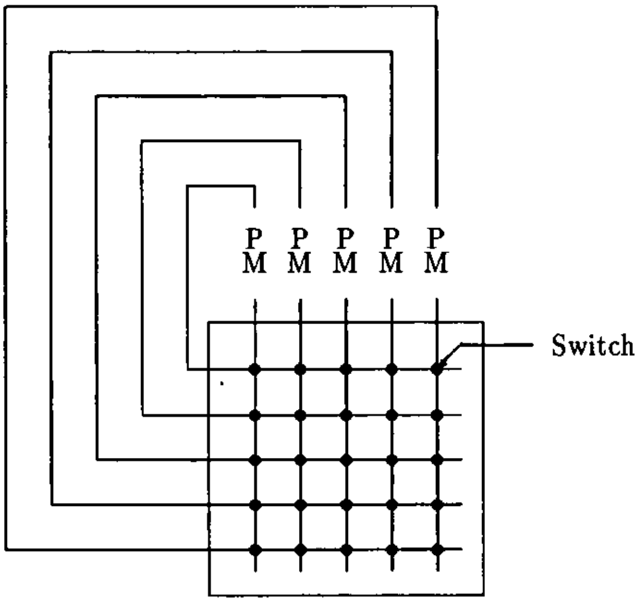


Figure 1.8: Crossbar switch network multiprocessor.

way that each processor is allowed to have equal access to any nonbusy memory unit. Therefore, each memory unit (M) has its own separate path to a processor (P) as shown in figure (1.8).

The crossbar switch is the most complex interconnection system. The functional units are the simplest and hence cheapest because the control and switching logic is in the switch. The main characteristics of this organisation are the high throughput, highest potential for system efficiency. System expansion improves the overall performance, the expansion of the system is limited only by the size of the switch. Also, this organisation has the potential for the highest total transfer rate. It is quite easy to partition the system to isolate the malfunctioning devices or to establish independent systems. However, the system proves to be too costly (PM) and complex for highly parallel systems because of too many connections.

3. **Multiport memories**

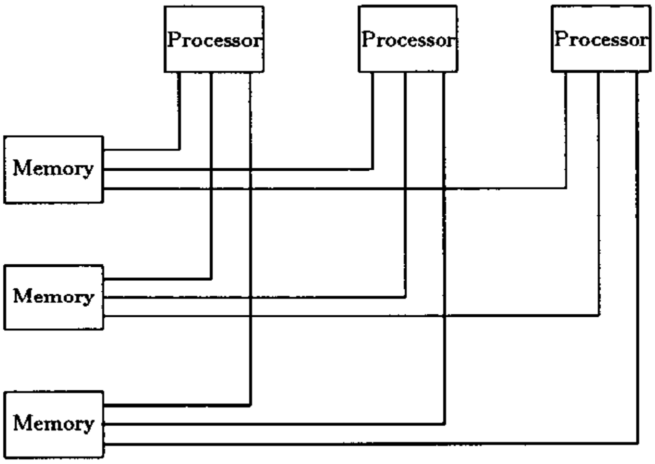


Figure 1.9: Multiport memories multiprocessor organisation.

In this organisation, the switching is concentrated in the memory module. Each processor has access through its own bus to all of the memory modules. Conflicts are resolved by assigning a fixed priority to the memory ports. It is possible to designate a portion of the memory as private to certain processors, I/O units, or a combination of both. Figure (1.9) shows a multiport memory multiprocessor organisation. Multiport memory requires the most expensive units since most of the control and switching circuitry is included in the memory units. The system has a very high transfer rate. The size and configuration options are determined by the number and type of memory ports available, and this decision is made quite early in the overall design process and is difficult to modify. This organisation requires number of cables and connectors.

Cost wise, the time-shared bus is the cheapest and the crossbar is the most expensive. The memory modules of the multiport organisation must have additional logic in their controllers and hence become expensive. The time-shared bus allows easy expansions, but the performance of the systems degrades rapidly if the system bus is overloaded. In terms of reliability, the time-shared bus and crossbar switches appear equally poor. Finally, both crossbar and multiport schemes allow maximum concurrency.

1.2.4 Data flow computers

The conventional Von Neumann machines are known as control flow computers because instructions are executed sequentially as controlled by a program counter. Therefore computation in such computers is done according to the flow of control in the program and each instruction is executed in turn making program execution inherently slow. To exploit parallelism in a program, data flow computers were developed. The basic concept behind this development is to schedule each operation in the function being carried out at run time when all the operands are available. The sequence of operations in the data flow computers obey the precedence constraints imposed by the algorithm rather than by the location of the instructions in the memory. The instructions in a program are not ordered, hence no program counter is required.

In theory, the amount of concurrency exploited in data flow machines is constrained by the availability of hardware resources, i.e. how many instructions can be executed simultaneously by the computer.

After executing the instruction, the result is distributed to all subsequent instructions, which make use of this partial result as an operand. In this way the data flow model of computation exploits the natural parallelism of algorithms. In computer architecture this makes it possible to create systems which can dynamically adopt their inner configuration to the natural structure of the algorithm being performed.

Programs for data-driven computations can be represented by "data flow graphs". An example of a data flow graph is given in figure (1.10) for the calculation of the mean and standard deviation (SD) given by following formulas:

$$\text{mean} = \frac{(x+y+z)}{3}, \tag{1.2.1}$$

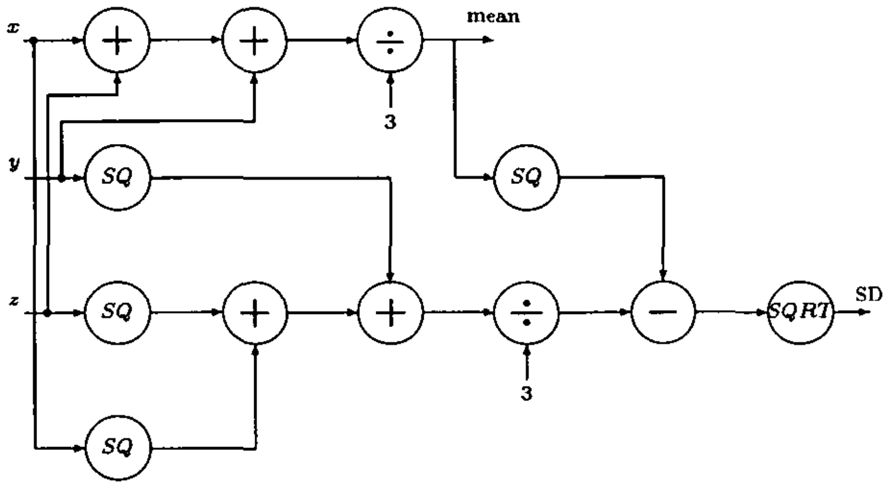


Figure 1.10: Data flow graph for calculating mean and standard deviation.

$$\mathrm{SD} = \sqrt{\frac{((x^2 + y^2 + z^2)}{3} - \mathrm{mean}^2)}. \qquad (1.2.2)$$

Miller and Cocke, in [58], designed the two basic models of data flow computer architectures which are:

- Search Mode configuration computer (SM-type) .
- Interconnecting Mode configuration computer (IM-type).

Both models are characterised by the possibility of dynamic adoption of its configuration to the structure of the algorithms. This is done by interconnecting (according to the graph) the processors that correspond to the operators in the data flow program of the problem. The reconfiguration is done either by hardware or software means. In the IM-type, the interconnection between processors is actually implemented through a large switch, i.e. by hardware means. In the SM-type, the interconnection is simulated by using a special instruction format, i.e. by software means. Due to reconfigurability, the data flow computer is able to achieve the performance of a specialised system whilst still keeping its general purpose capabilities.

In data flow computers, information items are either operation packets or data tokens . An operation packet is composed of the op-code, operands and destination of its successor instructions. A data token is formed with a result value and its destinations. Depending on the way these data tokens are handled, data flow computers are subdivided into a static model or a dynamic model.

The static model of data flow computers provides a fixed amount of storage per arc, i.e. only one data token is allowed to exist on any arc at any given time, otherwise the successive sets of tokens cannot be distinguished. In the dynamic model the machine uses tagged tokens so that more than one token can exist in an arc at the same time. An architectural comparison of data flow machines is described in Srini [72].

1.2.5 VLSI systems

As a result of the advancement in large-scale integration (LSI) technology, Very Large-Scale Integration (VLSI) technology has been developed. In this technology, circuit designs were introduced in which the number of transistors has been increased by a factor of 10 to 100 compared to the LSI technology. The VLSI technology has made possible for a 32-bit processor with memory and I/O support to be fabricated on a single chip.

The main advantages offered by the VLSI technology are; its capacity to implement enormous number of devices on a chip, at a very low cost, reduced power consumption and physical size, and reliability. Additionally, the high level of integration conceivably eliminates the need to physically separate processors from the memory, thus eliminating the bottleneck between them. The main problem with VLSI technology is to overcome the design complexity. This problem can be solved by using regular design structures e.g. a memory chip.

Software tools as well as hardware aspects are becoming more and more important in the design and testing stages of VLSI circuits. This leads to the fact that the production of a new chip requires software as much as hardware engineering knowledge.

The two major difficulties exercised by conventional computers are the separation between the processor from its memory and the limited opportunities for concurrent processing. By using VLSI technology it became possible to overcome these difficulties because memory and processing architecture can be implemented with the same technology and by their close proximity. Beside that the potential power of VLSI has come from the large amount of concurrency that the technology supports.

The degree of concurrency or parallelism in VLSI computing structure is largely determined by the underlying algorithm. The I/O bottleneck problem in VLSI systems presents a serious restriction imposed on the algorithm design. The challenge is to design parallel algorithms which can be partitioned such that the amount of communication between modules is as small as possible. Enormous parallelism can be obtained by introducing a high degree of pipelining and multiprocessing while designing the algorithm. Properly designed parallel structures that need to communicate only with their nearest neighbours will gain the most out of the VLSI design.

One way of achieving parallelism is by attaching a special-purpose parallel processor to the system bus of a microcomputer to speed-up the compute-bound algorithm rather than the I/O-bound computation. In a compute-bound algorithm, the number of computing operations is larger than the total number of I/O operations. Otherwise, the problem is I/O-bound. The attached parallel processor shown in figure (1.11) has two general architectural designs. The first being the multiprocessor lattice architecture based on the idea of several processing elements operating under centralised control and the second is a systolic array architecture which makes extensive use of pipelining.

A multiprocessor lattice architecture is an $(n * n)$ array of processing elements working concurrently under a centralised control and transmitted via a long

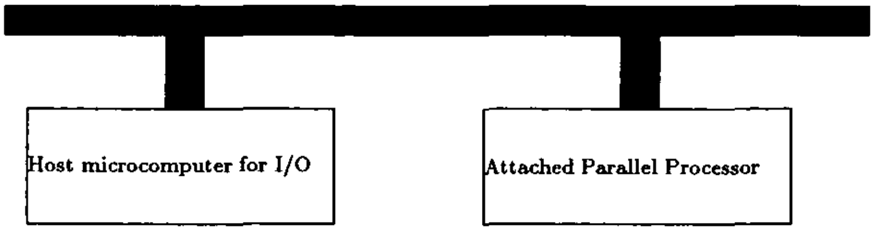


Figure 1.11: Achieving parallelism by using a special-purpose parallel processor.

local communication path connecting neighbouring processing elements. Each processing element has a private memory to store both results and any other temporary values which might be needed. The ICL DAP (see figure 1.5) is an example of an array lattice architecture. Designing a VLSI systolic system it is assumed that the processing cells (elements) in the array can be of different types and perform different operations. However, one design goal is to reduce the number of processing cell types.

The mapping of algorithms into systolic arrays is different than mapping of algorithms into architectures with fixed number of processors and interconnections. In case of systolic arrays, one has to deal with issues ranging from the organisation of the network of processing elements to the detailed operation of the processing elements. In fact, the mapping is the design of the VLSI array according to the properties of the algorithm and a set of design goals.

The design and implementation of a VLSI array can be broken into three main phases; task definition, design and processing (see Kung [44]). The design phase can be subdivided into system level, algorithms, logic circuit level and geometric layout. The processing phase concentrates on pattern generation, mask generation, wafer fabrication, device packaging and circuit testing.

A computer organisation can be obtained by having several special purpose VLSI processor arrays connected to the host computer and main memory via an interconnection network, (see Moldovan [60]). Figure (1.12) presents such a

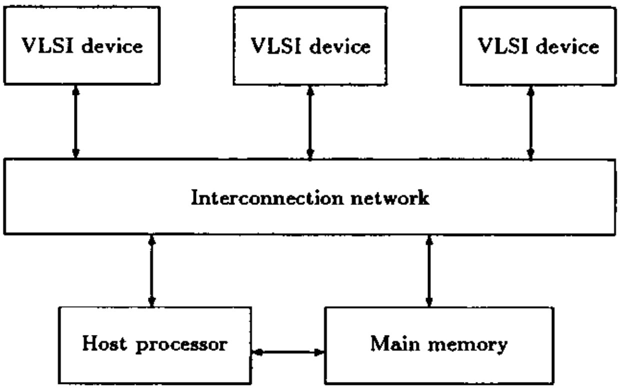


Figure 1.12: Computer system with several VLSI processor arrays.

design, the idea is to improve the performance of an existing computer system, using VLSI array processors. For example a systolic VLSI device implementing linear algebra routines can be built and connected to the system bus. The host computer when required calls these hardware routines by downloading the data into the device, and collects the results when available. Such devices are known as hardware accelerators (see Megson [56]). Each VLSI device consists of a number of processors working in parallel. The systolic array can be defined according to Kung and Leiserson, in [46], as a computing structure for making use of special-purpose VLSI chips.

A chip based on a systolic design consists, essentially of a few types of very simple cells (processing elements) which are mesh-interconnected in a regular and modular way and which achieve high performance through extensive concurrent and pipeline use of the cells. Figure (1.13) shows the basic principles of a systolic array.

The name systolic given by Kung and Leiserson is taken from the physiology of living cells. A "*systole*" is a contraction of the heart by means of which blood is pumped to the different components of the organism. In the systolic system, the information (data and instructions) is rhythmically given into a structure of elementary processors (cells) to be processed and passed to a neighbour cell until a result reaches some boundary of the system communicating with the

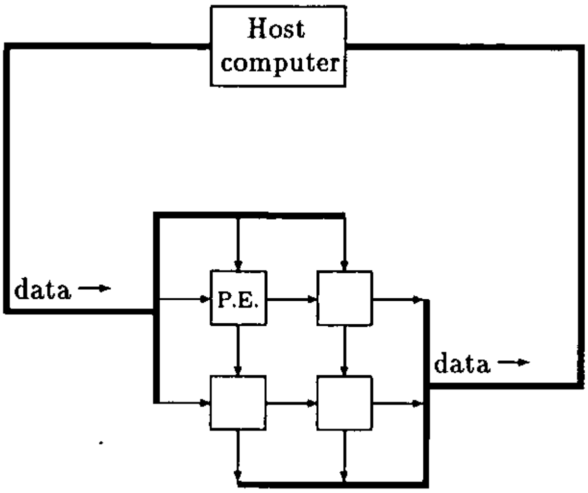


Figure 1.13: Systolic processor array.

host computer.

The speedup obtained in systolic arrays is not linear. The computational time can be divided into a parallel portion and serial portion. The speedup is asymptotically limited by the serial portion (performed on a single processor); independent of the parallel portion. This principle is known as Amdahl's law. If there are a sequence of 100 operations to be performed in parallel, compared to a single processor machine an 80 processor machine would attain a speedup of 5 or less but not 80 ! Similarly systolic arrays have the Von Neumann bottleneck. Systolic arrays have applications in the field of numerical problems, signal processing, pattern recognition.

1.3 Classification of computer systems

The Von Neumann model (classical sequential model) of computation consists of a stream of instructions and a stream of data, executing one instruction at a time, to produce a computational result, as shown in figure (1.1). The Von Neumann architecture is improved by using multiple processors and gave birth to the field of parallel processing. Flynn, in [23], give a taxonomy to classify these computational models as follows.

A computer architecture, can be classified in terms of parallelism within the data stream and instruction stream. A stream simply means a sequence of items as executed or operated on by a processor. In this context, instruction stream means the sequence of instructions that are executed in a processing unit and data stream means the sequence of operands that are manipulated in a processor.

Flynn classified computer systems into four classes according to the replication of instruction streams and data streams. These classes which are demonstrated in figure (1.14) are:

1. **Single-Instruction Single-Data stream (SISD):** This, in fact, is the sequential computer (Von Neumann machine), where at any time, only one instruction is in execution affecting at most one item of data.
2. **Single-Instruction Multiple-Data stream (SIMD):** This is a class of computers in which the data stream has been replicated. Therefore, each instruction operates on a data vector rather than on a single operand. One well known class of SIMD machines is the array processors (e.g. Illiac IV which contains an array of 64 fast floating point processors). Other SIMD computers, like the the Distributed Array Processor (DAP), Connection machine are the classical examples.
3. **Multiple-Instruction Single-Data stream (MISD):** Here in this class the replication is in the instruction stream instead of the data stream. Each operand is operated upon simultaneously by several instructions. This mode of operation is generally unrealistic for parallel computers. The nearest example for such a machine is the punched card processor (see Stone [74]). There has been no commercial machine built of this type.

4. **Multiple-Instruction Multiple-Data stream (MIMD):** In this type of machine parallelism is combined in both the instruction and data streams. A computer system of this class is composed of n processors each of which is a complete computer with the processors connected together to provide a means for cooperating during computation.

Multiprocessors are a subclass of MIMD systems in which processors have common access to the primary memory and I/O channels with a single operating system controlling the entire complex. Shared memory machines like (Sequent Balance) and distributed memory systems (The Hypercube and Transputer networks) discussed later in this chapter are the examples of MIMD machines.

Of the four types of computer systems mentioned above, the two of immediate interest are the SIMD and MIMD computers. These two types, which will be discussed in more detail later, are vastly different in how they attain parallelism of operations.

Flynn's macroscopic classification of parallel architecture does not depend on the structure of the machines, but rather on how the machine relates its instructions to the data being processed. Therefore, this classification scheme can be considered too broad.

Other classification schemes were designed depending on different criteria. One of these classifications is Shore's classification (see Shore [68]) which was based on how the computer is organised from its constituent parts. According to this criteria six different types of machines were recognized. These types which are distinguished by a numerical designator are:

- Machine I : This is the conventional Von Neumann computer with one control unit, one processing unit, an instruction memory and a data memory. This data memory reads all bits of any word for processing in parallel by the processing unit.

- Machine II : Same as machine I except that the data memory reads a

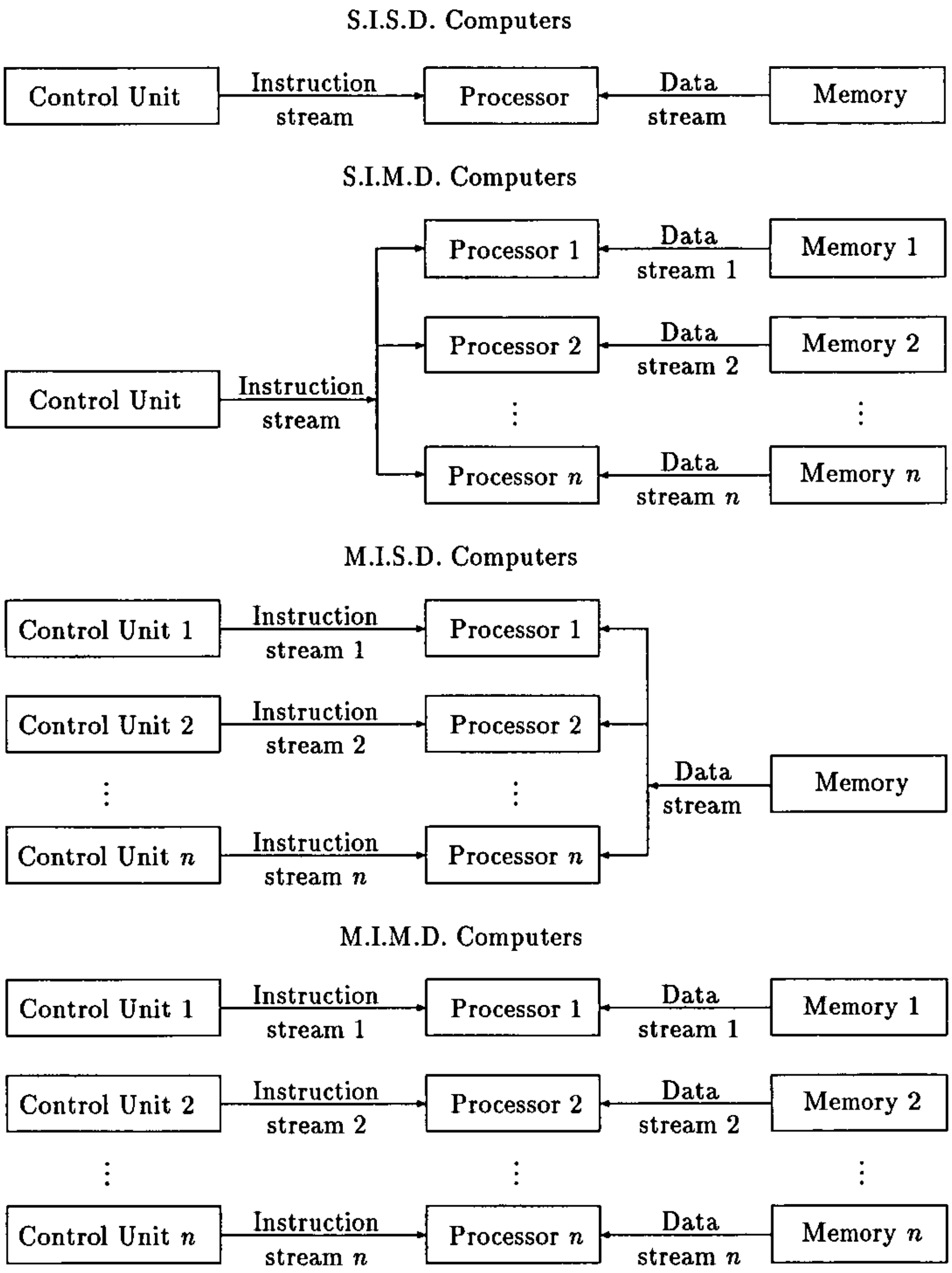


Figure 1.14: Flynn's classification of computer systems.

bit slice from all words in memory instead of all bits of one word. The processing unit in this machine is organised to perform its operations in a bit-serial fashion.

- Machine III : Combination of machines I and II.
- Machine IV : This machine can be obtained by replicating processor units and data memory units of machine I.
- Machine V : This is machine III with additional facility that processor units are arranged in a line and nearest-neighbour connections are provided.
- Machine VI : In this machine processor logic is distributed throughout the memory and because of that it is called logic-in-memory array (LIMA).

Skillicorn, in [69], presented a classification scheme, or taxonomy, that extends Flynn's to make some discrimination. This taxonomy is based on a functional view of architecture and on the information flow between units.

Feng's scheme, in [21], is based on serial versus parallel processing. It is a performance oriented classification that describes the parallelism of the set of processors in a machine in terms of the number of bits that can be processed simultaneously.

Another approach used by Reddi and Feustel, in [65], made use of the physical organisation, information flow and representation and transformation of information as the basis for classification.

Handler, in [36], in his classification describes architectures by giving the number of processors and how they can be pipelined together.

1.3.1 SIMD computers

The original motivation for developing SIMD computers was to perform parallel computations on vector or array types of data.

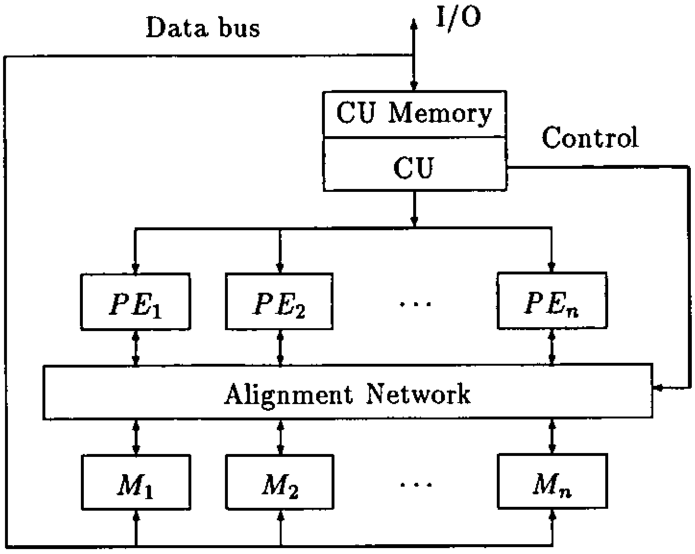


Figure 1.15: The SIMD computer organisation.

The SIMD computers consist of identical processing elements (PE's) arranged in an array and controlled by a single control unit. In this type of machine the same operation is performed at the same time over-all data in all processing elements. A simplified model of an SIMD computer is shown in figure (1.15).

This organisation has an identical number of processors and memory units beside the alignment network that allows the data to be transferred from one processor to another.

Because the PE's are passive devices without the capability of instruction decoding, user's programs are therefore loaded into the control unit (CU) which decodes the instructions and decides where to send them for execution. Scalar-control-type instructions are executed inside the CU itself whereas the vector-type ones are broadcast to the PE's for execution.

One of the major issues in the design of SIMD computers is the interconnection and transfer of data between the PE's. Therefore different interconnection networks have been proposed for this type of computer with the complete network, where each processor is connected to all other processors and which is the most expensive and most difficult to manage by both designers and users. Hwang and Briggs [41] is one of the good references covering this issue.

Many parallel processing algorithms have been developed for the SIMD computers. These algorithms can be used to perform matrix multiplication, Fast Fourier Transforms (FFT), matrix transposition, summation of vector elements, matrix inversion, parallel sorting, linear recurrence, Boolean matrix operations and solving partial differential equations (PDE's).

1.3.2 MIMD computers

The MIMD computer system architecture is an alternative design of great promise to produce high speed computers. It can be considered as a collection or network of minicomputers or microcomputers and collectively as a multiprocessor system.

An MIMD computer is composed of a number of processors each of which is a separate computer generating its own instruction stream which it executes on its data stream. These processors are connected either through a shared memory or via high-speed or low-speed data links.

Figure (1.16) shows an MIMD structure consisting of n processing units, m memory modules and p I/O channels.

By aiming to reduce the number of expensive components, i.e., the arithmetic and logic function units, Flynn, Podvin and Shmizu, in [24], presented a new approach for MIMD computer design. They proposed to interconnect several independent processors each executing an independent instruction stream. These processors, according to the proposal, are to be converted into "skeleton" processors by removing all the arithmetic functions and computational logic units from them. These functions are to be performed by highly-specialised high-speed processors that are shared amongst the "skeleton" processors. Hence, the new system avoids many of the contention problems usually associated with a shared resource system (figure 1.17).

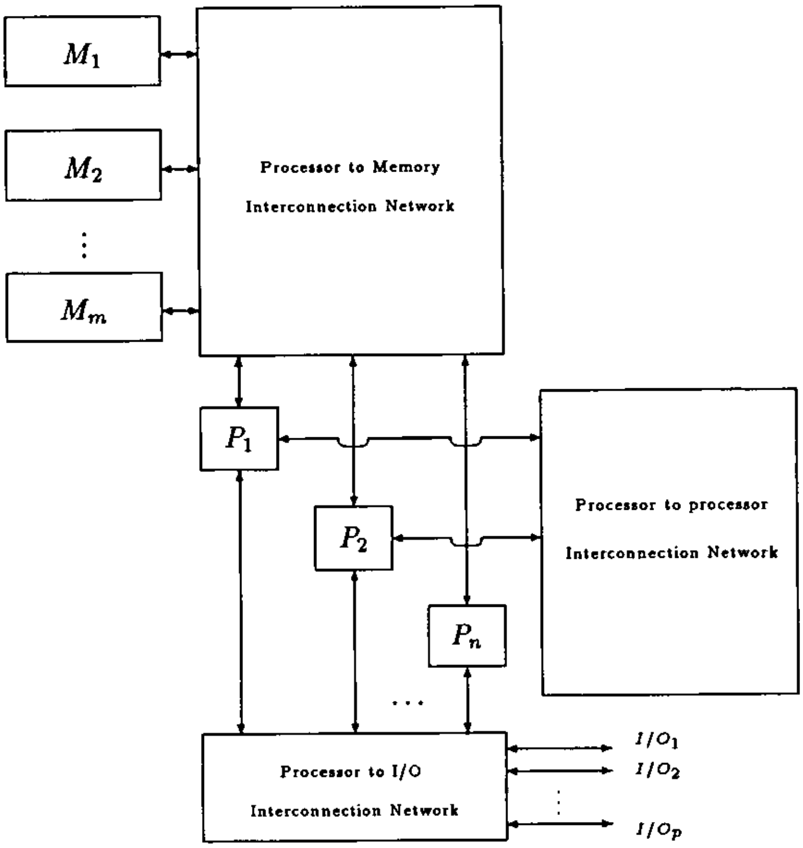

M_1
M_2
M_m
Processor to Memory Interconnection Network
P_1
P_2
P_n
Processor to processor Interconnection Network
Processor to I/O Interconnection Network
I/O_1
I/O_2
I/O_p

Figure 1.16: MIMD computer organisation.

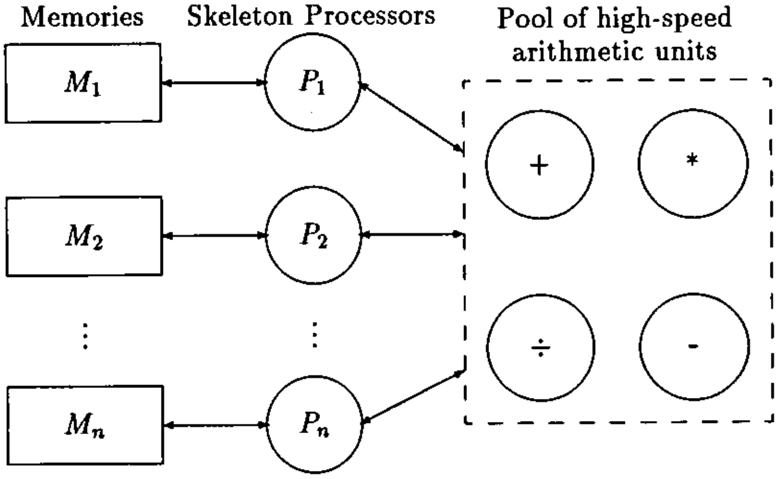


Figure 1.17: MIMD systems with skeleton processors.

There are two factors that degrade the MIMD system performance: the memory conflicts - in the software and the hardware - and the processor's interconnections. A software memory access conflict occurs when a processor attempts to use data that is currently being accessed by another processor which has activated a "lock" to prevent any other processor from accessing the same data set. This data set is called the "critical section". These types of conflicts are often known as "memory lockout". However, when a processor encounters a memory lockout it waits and repeatedly checks the status of the lock until the "unlock" state is set by the "locking" processor.

In a multiprocessing system with shared global memory, large memory latency is introduced by the communication subsystems (bus or multistage network). This memory latency can be mostly avoided by providing each processor with a cache memory so that most memory references can be satisfied locally. This kind of solution presents a special problem when copies of a given piece of information may potentially exist in several caches as well as in the main memory. When a processor changes one of these copies, the modification must eventually be reflected in all the others.

A hardware memory conflict occurs when two or more processors attempt to access the same memory module (or unit) simultaneously, i.e. the conflicting requests are issued during a single memory cycle. Since only one access can be made per memory cycle therefore the other requests must wait usually for a cycle or two in each case. To reduce the effect of these two factors, the use of the private memory (memory associated with each processor to store important data frequently needed by the processor) was increased.

The main advantages of MIMD computers are the high throughput and the greater reliability. High throughput is achieved by dividing the processes into many subprocesses which can run on different processors concurrently. Greater reliability can be achieved by easily isolating the faulty resources (processors and memory modules) and results in a better fault tolerant system.

MIMD designs are capable of performing well over a broader range of applications than the SIMD computers. The processors in an MIMD system need not be synchronised instruction-by-instruction as in a SIMD system. However, it is required that the processing algorithms exhibit a high degree of parallelism so that several processors are active at the same time.

Next two sections describe two classifications of the MIMD computers known as the loosely-coupled systems and the tightly-coupled systems.

1.4 Loosely-Coupled Systems (LCS)

The essence of the Loosely-Coupled Systems is a group of processors each of which has its own local memory, a CPU and a set of input-output devices or communication links. This combination is called a node or a computer module.

Communications between computer modules is established at the I/O level through a Message-Transfer System (MTS). This method of communication differentiates this model of architecture from a tightly-coupled architecture discussed in the next section.

A loosely-coupled system's characteristics can be summarised by the following points:

1. Each processor in the system has its own memory, therefore they do not encounter the same degree of memory conflicts experienced with tightly-coupled systems.
2. An explicit communication interface between the processors is needed.
3. Concurrent tasks may not be performed in synchronisation.
4. Each processor can stand by itself with its own storage.
5. Efficiency is high when the interactions between tasks is minimum.

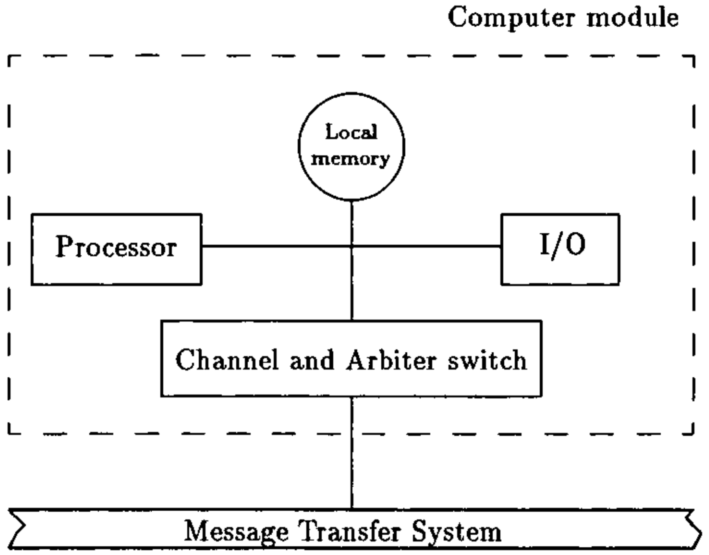


Figure 1.18: Loosely-coupled (LCS) multiprocessor module.

A computer module of a loosely-coupled multiprocessor system with a connection to the message-transfer system is shown in figure (1.18). Each computer module consists of a processor, a local memory, local input-output devices and an interface to the other modules called the channel and arbiter switch (CAS). The CAS is needed only when there is a conflict in accessing a physical segment of message-transfer system, the arbiter resolves the conflict. The channel within the CAS may have a high-speed communication memory which is accessible by all processors and is used for buffering block transfers of messages. With the advent of VLSI technology, the computer module can be fabricated on a single integrated circuit and be used as a building block for multiprocessor systems.

In this type of coupling, normally one of the processors is designated as an overall system control (global processor) while the other processors are considered as local processors. All the tasks enter the system through the global processor and if this processor fails, one of the local processors may take over as a global processor. One of the important factors that determines the performance of a LCS is the message-transfer system.

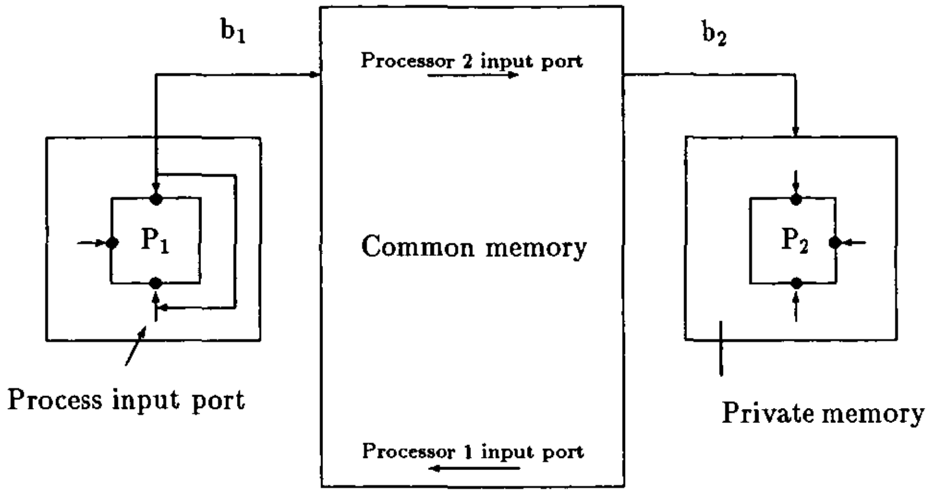


Figure 1.19: Communications between tasks in a LCS multiprocessor.

The message-transfer system could be a single time-shared bus or a shared memory system. For a LCS configuration using a single time-shared bus, the performance is limited by the message arrival rate on the bus, the message length and the bus capacity (in bits per second). Contentions for the bus increase as the number of computer modules increases. For the LCS with shared memory MTS, the limiting factor is the memory conflict problem imposed by the processor-memory interconnection network. The communication memory may also be centralised and connected to a time-shared bus or be part of the shared memory system. This type of memory can, conceptually, be considered as consisting of logical ports which can be accessed by the processors. As figure (1.19) shows, for each processor there is an associated communication port residing in the communication memory that works as the processor's input port. Thus, communication messages between tasks allocated to different processors are transferred in two steps through the communication memory, whilst communications between tasks allocated to the same processor take place through the processor's local memory.

In a loosely-coupled system's synchronisation, task partitioning, software control and data transfers are the problems to be taken into consideration. To improve the performance the user must determine how to divide the task between the nodes or processors so that they can operate in an efficient mode of parallelism.

A loosely-coupled parallel computer can be characterised by three parameters: f, the floating-point speed , s the start-up time for an I/O operation and r the transfer rate, with all the parameters being measured in seconds per operation. It is important to note that, at least in current implementations s is dominated by software costs (system calls, memory allocation) and thus restricts the overlapping between I/O operations (or communications) with computations on the nodes. Now in current implementations $s \gg f$ since much effort has been devoted to fast floating-point chips. For similar reasons $s \gg r$. The pattern of connections between processors is called the topology of a parallel processor. The most common topologies for loosely-coupled systems are the ring, the mesh (usually 2-dimensional) and the binary n-cube or the Hypercube.

1.5 Tightly-Coupled Systems (TCS)

In this model of MIMD architecture, the number of processing units is fixed and they operate under the supervision of a strict control scheme. Processors in this system communicate with each other through a shared main memory, allowing each one of them access to all the other's memories.

As a result the data transfer rate is dependent on the bandwidth of the memory. The complete connectivity between the processors and the memory can be accomplished either by inserting an interconnection network or by a multi-ported memory.

The system's performance is limited by two major factors. The first one is the degradation due to conflicts to access the main memory or the I/O devices while the second one arises in synchronising and scheduling jobs to the multiple processors. Therefore, and because of these two limiting factors, it is realised that a large number of processors cannot be utilised effectively in a multiprocessor system having such coupling. Hence, a small number of processors is preferable in designing an MIMD tightly-coupled system.

The set of processors used in a multiprocessor system may be homogeneous or heterogeneous. It is homogeneous if the processors are functionally identical. Even if the processors are homogeneous, they may be asymmetric. That is, two functionally identical components may differ along other dimensions, such as I/O accessibility, performance or reliability. The symmetry or asymmetry of the processors in a multiprocessor system is usually transparent to the user processes.

Multiprocessor systems with a heterogeneous set of processors perform several different operations in parallel - rather than the same operation on several sets of data. These machines have become available in the last few years (see Browne [8]). The Heterogeneous Element Processors (HEP) built by Denelcor was probably the first commercially available general-purpose computer that can perform several operations concurrently. The HEP system may have up to 16 process execution modules and up to 128 data memory modules connected by a packet-switched network. Each of the process execution modules can use multiplexing to work on up to 100 active processes or instruction streams concurrently.

Finally, the tightly-coupled model of architecture can be used to build either SIMD machines for specialized applications or MIMD machines with custom-designed functional units or a mixture of both as in pipeline processors. Moreover this architecture is more suitable for systems which are aimed to have raw computing power and fast response time or real-time operation.

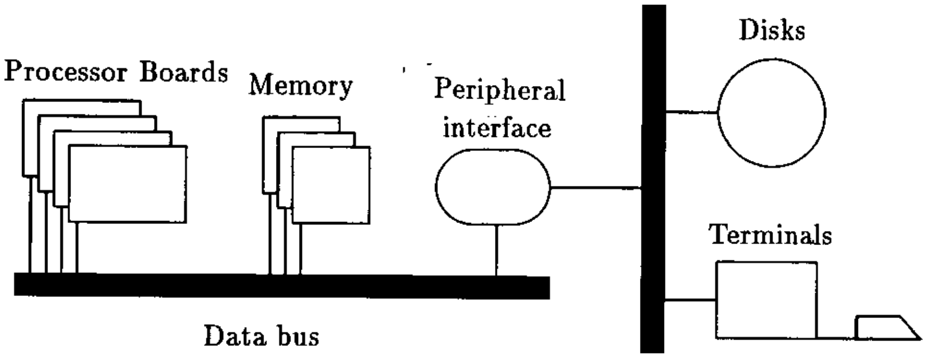


Figure 1.20: Sequent Balance 8000 architecture.

1.5.1 The Sequent Balance and Symmetry systems

An example of the tightly-coupled MIMD architecture system or more precisely the shared bus architecture, which we will discuss in this section, is the Sequent computer architecture.

The Sequent Computer Systems Inc., have developed two families of parallel computers; the Balance Series and the Symmetry Series. These two series are very similar in their structure, configuration, operating system, and user software. However, the primary difference between them is the type of microprocessor used to build the CPU's in each of them which has led to a substantial difference between the two series at the machine language level. There are, of course, other differences such as the speed, performance and memory size (see Trew and Wilson [76]).

In this discussion we shall concentrate on the Balance series and in particular, on the Balance 8000 model of the series because most of the early part of this research has been carried out using a simulator running on this type of machine.

The Balance 8000 model can have up to twelve 32-bit processors in a tightly-coupled manner (see figure 1.20). The machine at Loughborough University's Parallel Algorithm Research Centre (PARC) has 12 processors. These processors are connected via a high-speed bus to all peripherals and shared memory and concurrently execute a shared copy of a Unix-based operating system. Any processor can execute any program to achieve dynamic load balancing and multiple processors can work in parallel on a single application and to minimise accesses to the system bus each processor has its own cache memory.

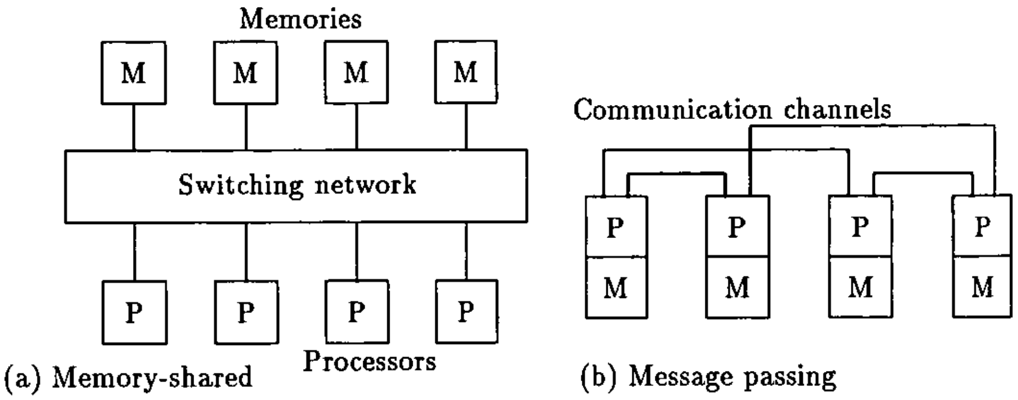
All of the CPU units have a Floating Point Unit (FPU) and a Memory Management Unit (MMU) and a System Link and Interrupt Controller (SLIC) whose task is to manage the control of multiple processors.

The DYNIX (Dynamic Unix) operating system is an enhanced version of Berkely Unix 4.2bsd which can emulate Unix System V at the system-call and command levels. To support the Balance multiprocessing architecture, the DYNIX operating system kernel has been made completely shareable so that multiple CPU's can execute identical system calls and other kernel codes simultaneously.

1.6 The Hypercube systems

As computer systems consisting of tens and even hundreds of processors have already been built, research is carried out, nowadays, to investigate the feasibility of practical machines that can be expanded economically to have thousands of processors all working on one problem instantaneously. As previously mentioned, MIMD parallel computers can be classified, in very broad terms, into tightly-coupled represented by the shared-memory systems and loosely-coupled represented by the message-passing systems.

A computer of loosely-coupled type of architecture is, essentially, composed of a large number of identical processors connected to each other according to some convenient pattern Each of these processors contains a local memory , a CPU and communication links, over which messages are sent to or received from other processors. This combination forms a node. On the con-



(a) Memory-shared (b) Message passing

Figure 1.21: Hardware structures for memory-shared and message-passing systems.

trary to the computer systems of shared-memory type of architecture where communications between processors are carried out through messages left in the common memory bank, we see that the communication between computer systems of message-passing type of architecture is performed by exchanging messages across the links or communication channels. Figure (1.21) shows, schematically, the hardware structure of both the shared-memory and the message-passing multiprocessor systems.

To solve problems on a concurrent machine requires partitioning the problems into a number of segments that can run independently on more than one processor. Many applications, particularly in scientific computing, lend themselves to this form of partitioning. One of the most suitable parallel architecture designs for these partitioned applications is the Hypercube system. A Hypercube is a concurrent processor in which nodes can be imagined to be at the vertices of an n-dimensional cube. Each node of the Hypercube is linked to d neighbouring processors and the entire network contains 2^d nodes (see Fox and Otto [28]). The important features of Hypercube systems are; the number of links is fairly small, although it is still dense enough to support efficient communication between arbitrary processors. The number of links between a node and its neighbours increases slowly relative to the increase in the number

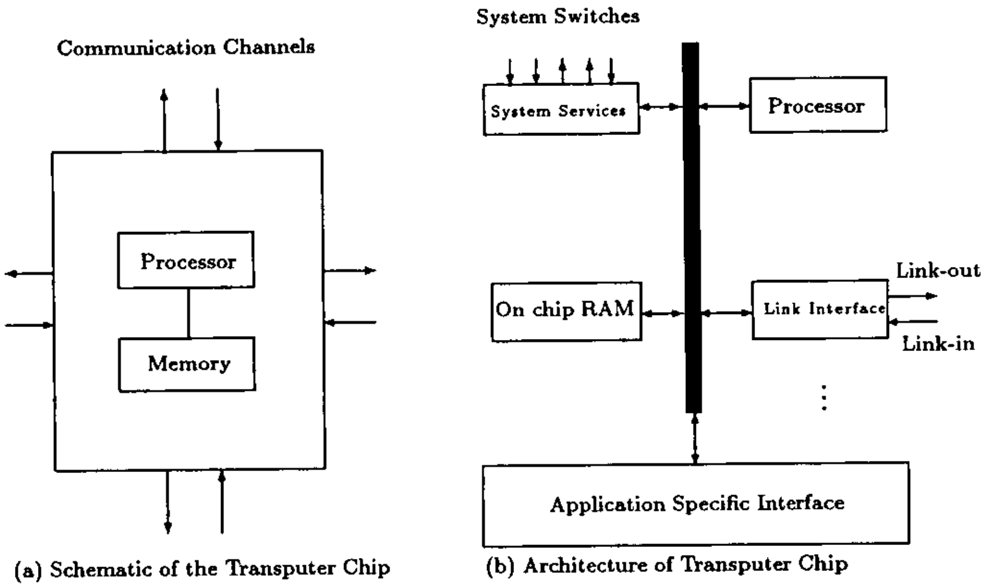


Figure 1.22: The Transputer chip.

of nodes, or in other words, the Hypercube computational capacity. The Hypercube design is flexible due to its interconnection scheme which allows these machines to possess excellent mapping capabilities.

1.7 Transputer systems

Among the most suitable processors for parallel processing systems is the INMOS transputer, which has been designed explicitly as a basic building block for processor arrays of limited size due to the communication problems referred to previously. The transputer provides a direct implementation of the message-passing mode of parallel computation of loosely-coupled systems. It is a high performance single chip computer (see figure 1.22) with link interfaces which enable a number of transputers to be readily assembled into networks of general topologies like linear array, square mesh etc.

The transputer architecture defines a family of programmable VLSI components. A typical member of the transputer product family is a single chip containing processor, memory and communication links which provide point to point connection between transputers. The treatment of parallel processing in transputer systems is based on a model of computation called Communicating Sequential Processes (CSP), proposed by Hoare, in [40]. In this model a computing system is a collection of concurrent active sequential processes which can only communicate with each other over channels (i.e. using message passing rather than shared memory communication). Only two processors may be connected by a channel, which can only carry messages in one direction therefore if communication in both directions between the two processors is required, two channels must be used. Channel communication must be synchronised (i.e. a process wanting to send a message over a channel is always forced to wait until the receiving process reads the message).

The transputer has been designed to be a hardware realisation of the CSP model. Each transputer processor has four serial links to connect it with other transputers. Each link has two hardware channels, one in each direction. The hardware channels behave exactly like the abstract channels; they provide synchronised unidirectional communication. Arbitrary networks of transputers can be constructed simply by connecting their links together with ordinary wires; the only limitation being that each processor cannot be directly connected to more than four others. Transputers can be programmed efficiently in most high level languages like C, Fortran, Pascal and Occam (the INMOS parallel processing language). If it is required to exploit concurrency, but still to use standard languages, Occam can be used as a harness to link modules written in the selected standard language. To acquire the greatest advantage from transputer architecture, the whole system should be programmed in Occam. This language provides all the benefits of a high level language, the maximum program efficiency and the ability to use the special features of the transputers related to parallelism.

One of the most important aspects of parallel processing is the ability to build a system that models the problem to be solved. High performance systems to match different applications can quickly and easily be assembled using transputer technology and also by linking transputers together, massive processing power can be provided for super-computing applications.

2.1 Introduction

In this chapter some basic definitions and theory concerning linear algebra, linear systems and partial differential equations are given. Linear algebra deals with the specification and solution of linear systems (of equations) which can be derived from various problems in Mathematics, Engineering, Business and Economics. In fact partial differential equations can be discretised on a finite difference/element grid in terms of linear systems.

The chapter includes definitions of vectors, matrices along with relevant properties and relations. This base is then used to represent the linear systems in matrix/vector notation and to discuss direct and indirect methods for solving linear systems. Partial differential equations are then defined. Finally finite difference techniques are explained to obtain the linear system of equations corresponding to the continuous partial differential equations.

Introductory material can be found in (Kolman [42], Burden and Faires [9], Lipschutz [49], Forsythe and Wasow [25], Smith [70]), and for the advanced text see (Varga [78], Young [82], Mitchell and Griffiths [59]).

2.2 Vectors and Matrices

Matrices provide a concise method for specifying and manipulating large numbers of linear equations. This section provides some basic definitions of matrix computations specifically related to the numerical algorithms used later on.

A linear system of m equations is represented in the form

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ \vdots \qquad\qquad\qquad \vdots \quad &\quad \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &= b_m. \end{aligned} \tag{2.2.1}$$

where A is a $(m * n)$ coefficient matrix, x is a unknown vector and b is the known right hand side vector.

A solution to a linear system (2.2.4) is a sequence of n numbers $s_1, s_2, \ldots, s_n$, which have the property that each equation in (2.2.4) is satisfied when $x_1 = s_1$, $x_2 = s_2$, ..., $x_n = s_n$ are substituted.

If the linear system represented by equation (2.2.4) has no solution it is said to be *inconsistent*; if it has a solution, it is called *consistent.* If $b_1 = b_2 = \ldots = b_n = 0$ then the linear system (2.2.4) is called a *homogeneous* system. The solution $x_1 = x_2 = \ldots = x_n = 0$ to a homogeneous system is called the *trivial* solution. A solution to a homogeneous system in which not all of the x_1, x_2, ..., x_n, are zero is called a *nontrivial* solution.

If A and B are $(m * n)$ matrices then $C = A \pm B$ is defined by; $c_{ij} = a_{ij} \pm b_{ij}$, $i = 1, 2, \ldots, m$, $j = 1, 2, \ldots, n$. If A is a $(m * n)$ matrix and B is a $(n * p)$ matrix then $C = AB$ is a $(m * p)$ matrix with,

$$c_{ij} \quad = \quad \sum_{k=1}^{n} a_{ik} b_{kj} \quad \begin{array}{l} i = 1, 2, \ldots, m \\ j = 1, 2, \ldots, p \end{array} \tag{2.2.5}$$

In general $AB \neq BA$.

If x and y are two vectors with n components then $\underline{z} = \underline{x} \pm \underline{y}$ is defined by, $z_i = x_i \pm y_i$, $i = 1, 2, \ldots, n$.

If A is a $(m * n)$ matrix and $\underline{x}$ is a vector with n entries then $\underline{y} = A\underline{x}$ is a vector of m entries, with $y_i = \sum_{k=1}^{n} a_{ik} x_i$, $i = 1, 2, \ldots, m$. Similarly for $\underline{x}^T$ a row vector with m elements, $\underline{y}^T = \underline{x}^T A$ produces a row vector with n entries.

The inner product of two vectors of n elements $\underline{y}^T \underline{x}$ is a scalar, while the outer product of the two vectors, $\underline{x}\underline{y}^T$ is a $(n * n)$ matrix. The inner product of two vectors can be written as

$$\sum_{i=1}^{n} x_i y_i. \tag{2.2.6}$$

This relation can also be written as a recurrence relation,

$$z^{(1)} = 0, \qquad (2.2.7)$$

$$z^{(i+1)} = z^{(i)} + x_i y_i, \quad i = 1, 2, \ldots, n \qquad (2.2.8)$$

$$z = z^{(n+1)} \qquad (2.2.9)$$

The computation of $z^{(i+1)} = z^{(i)} + x_i y_i$ is called the inner product step. The inner product of two vectors is usually called the dot product.

Suppose that we have n vectors $\underline{x}_1, \underline{x}_2, \ldots, \underline{x}_n$ all with n components. If the relation,

$$c_1 \underline{x}_1 + c_2 \underline{x}_2 + \cdots + c_n \underline{x}_n = 0$$

has the only solution

$$c_1 + c_2 + \cdots + c_n = 0,$$

the vectors are said to be *linearly independent.*

A scaled matrix A is the matrix sA where "s" is a scalar chosen so that all entries of A are kept within desirable size bounds. Usually "s" is a power of 2. Normalization of A is a scaling operation where "s" is a suitably chosen element of A. Similarly, a scaled or normalized vector, x can be defined. The scaling or normalization operation is useful in cases where the entries of a given matrix or vector differ greatly in size. Also, in cases where successive matrix-vector or matrix-matrix multiplications are performed and there is a rapid increase or decrease in the size of the matrix or vector elements.

Zero (null) matrix is the matrix with all its elements equal to zero and is represented by "O". Zero (null) vector is a vector with all its elements equal to zero and is denoted by $\underline{o}$.

The set of elements a_{ii}, $i = 1, 2, \ldots, n$ of a matrix A is the main diagonal of A. The diagonals above (below) the main diagonal are called super_ (sub_) diagonals.

If $a_{ij} = 0$ for $i \neq j$, the matrix A is said to be a diagonal matrix, usually denoted by D.

The identity matrix, I, is a diagonal matrix with all diagonal elements equal to 1.

The matrix A is a lower triangular matrix if $a_{ij} = 0$ for $i < j$ $(i \leq j)$. Similarly, A is an upper triangular matrix if $a_{ij} = 0$ for $i > j$ $(i \geq j)$. A unit lower (upper) triangular matrix has all the main diagonal elements equal to 1. A lower (upper) triangular matrix is usually denoted by L(U).

An $(n * n)$ matrix A is said to be a banded matrix if integers p and q, $1 < p$, $q < n$, exist, with the property that $a_{ij} = 0$ whenever, $i + p \leq j$ or $j + q \leq i$. The bandwidth of the matrix is defined to be $w = p + q - 1$. Since this is the number of non-zero diagonals in the band. If $p = q = 2$ then matrix A is called tridiagonal, if $p = q = 3$ then matrix is called quindiagonal.

The inverse of an $(n \times n)$ matrix A is an $(n \times n)$ matrix A^{-1}, such that $AA^{-1} = A^{-1}A = I$. Matrices with inverses are termed nonsingular and those without singular. A matrix is said to be singular if $\det(A) = 0$.

The transpose of $(n \times n)$ matrix A is an $(n \times n)$ matrix A^T, obtained by interchanging the rows and columns of A, i.e. $a_{ij} = a_{ji}$ for all $i \neq j$. A matrix whose transpose is itself (i.e. $A^T = A$) is said to be symmetric. A symmetric $(n \times n)$ matrix A is said to be positive definite if $\underline{x}^T A \underline{x} > 0$ for every n-dimensional column vector $x \neq 0$.

If most of the elements a_{ij} of a matrix A are zero, then A is said to be a *sparse* matrix. If most of the elements of the matrix A are non-zero, then the matrix A is a *full (dense)* matrix. A matrix is termed sparsely banded if there are null diagonals between the significant diagonals.

A matrix is called upper (lower) *Hessenberg* matrix if $a_{ij} = 0$ for all $i > j + 1$

(for all $j > i+1$), $i, j = 1, 2, \ldots, n$.

A matrix is called Toeplitz matrix if its elements are constant along each diagonal, i.e. a_{ij} is a function of $| j - i |$.

2.2.1 Vector and Matrix Norms

We can measure the size of a vector "$\underline{x}$", for "size" the word norm is used, with notation $\|\underline{x}\|$ and is a real positive number. The norm $\|\underline{x}\|$ is a non-negative number with the properties,

$$\|\underline{x}\| \neq 0 \text{ unless } \underline{x} = \underline{0}, \text{ and } \|\underline{o}\| = 0 \tag{2.2.10}$$

$$\|\alpha\underline{x}\| = | \alpha | \, \|\underline{x}\| \text{ where } \alpha \text{ is a constant} \tag{2.2.11}$$

$$\|\underline{x} + \underline{y}\| \leq \|\underline{x}\| + \|\underline{y}\| \tag{2.2.12}$$

Different vector norms are denoted by a subscript e.g. $\|\underline{x}\|_p$ for the p-norm. Three different vector norms most commonly used are given by,

$$\begin{aligned} \|\underline{x}\|_\infty &= \max_i | x_i | \\ \|\underline{x}\|_1 &= \sum_{i=1}^{n} | x_i | \\ \|\underline{x}\|_2 &= \left[\sum_{i=1}^{n} | x_i |^2 \right]^{\frac{1}{2}} . \end{aligned} \tag{2.2.13}$$

The importance of the norm lies in the fact that $\underline{x}^{(k)} \rightarrow \underline{x}$ if and only if

$$\|\underline{x}^{(k)} - \underline{x}\| \rightarrow 0. \tag{2.2.14}$$

This means that a series of vectors $\underline{x}^{(k)}$ converges to a vector $\underline{x}$ under a norm if for $\varepsilon > 0$, and some integer s

$$\|\underline{x}^{(k)} - \underline{x}\| < \varepsilon \text{ for all } k > s \tag{2.2.15}$$

Equation (2.2.15) indicates an error bound on the approximation and is used as an algorithm termination criteria. In iterative algorithms the value "ε" is termed as the tolerance representing the accuracy of the solution i.e. when the distance between an approximate and exact solution vector is considered close enough. A matrix "norm" for a $(n * n)$ matrix "A" is a function denoted by $\|A\|$ and is a real positive number giving a measure of the "size" of the matrix. For matrices A, B and a scalar α, matrix norms satisfy the following properties,

$$\|A\| > 0, \text{ if } A \neq 0 \qquad (2.2.16)$$

$$\|\alpha A\| = \mid \alpha \mid \|A\| \qquad (2.2.17)$$

$$\|A + B\| \leq \|A\| + \|B\| \qquad (2.2.18)$$

$$\|AB\| \leq \|A\| \|B\| \qquad (2.2.19)$$

Since matrices and vectors occur together in iterative methods, for example the left hand side of equation (2.4.1) uses matrix vector multiplication, hence matrix norms connected with vector norms are required. This means that the two norms are compatible. Compatibility means satisfying the condition,

$$\|A\underline{x}\| \leq \|A\| \|\underline{x}\|, \ \underline{x} \neq \underline{0} \qquad (2.2.20)$$

which is as equation (2.2.19) where the matrix B is replaced by the vector $\underline{x}$.

2.2.2 Eigenvalues and Eigenvectors

An *eigenvalue* of a matrix A is a number "λ" such that for some $\underline{v} \neq 0$ we have

$$A\underline{v} = \lambda \underline{v} \qquad (2.2.21)$$

An eigenvector of A is a vector $\underline{v}$ such that $\underline{v} \neq \underline{0}$ and for some "λ" in equation (2.2.21) holds. Equation (2.2.21) can be written as,

$$(A - \lambda I)\underline{v} = \underline{0} \tag{2.2.22}$$

The nontrivial solution $\underline{v} \neq \underline{0}$ to this matrix exists if and only if the matrix of the system is singular, i.e.,

$$\det(A - \lambda I) = 0 \tag{2.2.23}$$

Equation (2.2.23) is called the characteristic equation of A and the left hand side is called the characteristic polynomial of A, which can be written as,

$$\alpha_0 + \alpha_1 \lambda + \alpha_2 \lambda^2 + \cdots + \alpha_{n-1} \lambda^{n-1} + (-1)^n \lambda^n = 0. \tag{2.2.24}$$

Since the coefficients of λ^n are not zero, then the equation (2.2.24) has n roots (complex and real) which are the n eigenvalues of the matrix A, namely, $\lambda_1, \lambda_2, \ldots, \lambda_n$ (not necessarily all distinct), each of them corresponding eigenvectors. Wilkinson, in [81], described many methods for obtaining the eigenvalues along with corresponding eigenvectors.

If A is a square matrix of order n with eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_n$, then

$$\det A = \prod_{i=1}^{n} \lambda_i, \text{ trace } A = \sum_{i=1}^{n} \lambda_i \tag{2.2.25}$$

If A and B are similar, then they have the same eigenvalues. Moreover, "λ" is an eigenvalue of A of multiplicity k if and only if "λ" is an eigenvalue of B of multiplicity k.

The *spectral radius* of a matrix A is defined as

$$S(A) = \max_{\lambda \in S_A} \mid \lambda \mid, \tag{2.2.26}$$

where λ is an eigenvalue of A. From equation (2.2.26) it can be easily shown that for any $(n * n)$ matrix A and any norm,

$$S(A) \leq \|A\| \tag{2.2.27}$$

This can be proved as follows,

Let λ_i be an arbitrary eigenvalue of A and $\underline{v}_i$ its corresponding eigenvector then,

$$A\underline{v}_i = \lambda_i \underline{v}_i \tag{2.2.28}$$

and

$$\mid \lambda_i \mid .\|\underline{v}_i\| = \|A\underline{v}_i\| \leq \|A\|.\|\underline{v}_i\|$$

for any compatible norm. Thus ,

$$\lambda_i \leq \|A\|,$$

Since λ_i was arbitrary chosen, therefore we get,

$$S(A) \leq \|A\|.$$

2.3 Computational linear algebra

Given a linear system of the form

$$A\underline{x} = \underline{b} \tag{2.3.1}$$

where A is a $(n * n)$ coefficient matrix, $\underline{b}$ is a known n_vector and $\underline{x}$ is an unknown n_vector whose value is to be found. If the determinant of A is non-zero i.e. $(\det A \neq 0)$, the unique solution of the equation (2.3.1) can be expressed as,

$$\underline{x} = A^{-1}\underline{b}, \tag{2.3.2}$$

where A^{-1} is the inverse of matrix A.

In numerical analysis, however, the computation of this inverse is preferably avoided. The matrix A can be dense or sparse. Different solution methods are used to solve these two types of matrices and these are classified as direct or indirect (iterative) methods.

The system represented by equation (2.3.1), can be solved by using a direct method i.e. by changing the matrix A (most suitable for dense matrices) to a easily inverted form i.e. triangular, or by simply using iterative methods which does not alter matrix A (most suitable for sparse matrices). In the case of dense matrices, when stored in full, n^2 memory locations are required and any zero elements can be changed to non-zero elements which can be disastrous for the later category of matrices i.e. sparse. In that case, a matrix is likely to be generated and only the non-zero elements need to be stored.

2.3.1 Direct methods

The direct methods are principally based on elimination techniques and the amount of work involved is fixed and known beforehand. Furthermore, the solution process is done just once and the only errors in the solution are the round-off errors introduced in the computation.

The most common direct method for the solution of the linear system represented by equation (2.3.1) is known as the *Gaussian* elimination method (see Golub and Loan [30]). The method decomposes the matrix A into lower and upper triangular matrices L and U respectively, of the same order as the matrix A, with matrix U having 1's on its diagonal. This method is known as triangular decomposition, or LU-decomposition method and is feasible only if the matrix A is nonsingular. Hence, equation (2.3.1) can be replaced by

$$LU\underline{x} = \underline{b}, \tag{2.3.3}$$

or by introducing an auxiliary vector say $\underline{z}$ the linear system of equations

(2.3.3) can be rewritten as,

$$L\underline{z} = \underline{b}, \tag{2.3.4}$$

$$U\underline{x} = \underline{z}. \tag{2.3.5}$$

The original set of equations (2.3.1) is solved in two stages, by solving the equation (2.3.4) for $\underline{z}$, followed by the equation (2.3.5) for $\underline{x}$, called forward elimination and backward substitution procedures, respectively. However, for this proposition to be viable, equations (2.3.4) and (2.3.5) must be easily solvable.

2.3.2 Indirect methods

In contrast to the direct methods are the iterative methods. These methods generate a sequence of approximate solutions $\underline{x}^{(k)}$ and essentially involve the matrix A only in the context of matrix-vector multiplication. The evaluation of an iterative method invariably focuses on how quickly the iterates $\underline{x}^{(k)}$ converge. The iterative procedure is said to be convergent when the difference between the exact solution and the successive approximation tends to zero, as the number of iterations increases.

The basic idea of iterative methods can be described as follows:

- The matrix A is written as the difference of two matrices M and N, so that

  $$A = M + N. \tag{2.3.6}$$

  This decomposition is known as *splitting* the matrix A.

- An initial approximation $\underline{x}^{(0)}$ is made for the solution vector $\underline{x}^{(t)}$ which is the true solution of the system.

- A sequence $\underline{x}^{(1)}, \underline{x}^{(2)}, \underline{x}^{(3)}, \ldots,$ of estimates to $\underline{x}^{(t)}$ is generated by the formula,

$$M\underline{x}^{(k+1)} = \underline{b} - N\underline{x}^{(k)}, \quad k = 0, 1, 2, \ldots \ . \tag{2.3.7}$$

It can be observed that solving the system (2.3.1) is equivalent to solving the system

$$M\underline{x} = \underline{b} - N\underline{x}, \tag{2.3.8}$$

In order to find out what will be the best choice for M and N, we consider formula (2.3.7). This formula suggests that if we have $\underline{x}^{(k)}$, then we can get the next iterate $\underline{x}^{(k+1)}$ provided we can solve the linear system,

$$M\underline{x}^{(k+1)} = \underline{h}^{(k)}, \tag{2.3.9}$$

where the vector $\underline{h}^{(k)}$ is given by

$$\underline{h}^{(k)} = \underline{b} - N\underline{x}^{(k)}. \tag{2.3.10}$$

Thus, it is clear that we must require M to be nonsingular in order to be assured that we can implement the iteration. Furthermore, for an iterative procedure to be efficient, M should be chosen so that the linear system (2.3.9) is quite easy to solve; this is the case if, for instance, M is chosen to be a triangular or diagonal matrix. However, the total amount of work involved is not known, as the calculations continue indefinitely until the answers have converged and to sufficient accuracy. In fact, the process may not even converge and therefore it is important to know of any conditions under which an iterative procedure can be guaranteed to converge (see Varga [78]).

The first and simplest iterative method is the *Jacobi* iterative method. For the purposes of briefly discussing this method it is convenient to think of the matrix N as the sum of lower and upper triangular matrices. To be specific, let L, D, and U to be the lower triangular, diagonal, and upper triangular components of the $(n * n)$ matrix A. Thus,

$$A = L + D + U, \tag{2.3.11}$$

and so the Jacobi splitting is given by,

$$\begin{aligned} M &= D, \\ N &= (L + U). \end{aligned} \tag{2.3.12}$$

The *Jacobi* iterative method for solving the system (2.3.1) is given by,

$$D\underline{x}^{(k+1)} = \underline{b} - (L + U)\underline{x}^{(k)}, \tag{2.3.13}$$

while the matrix

$$M_J = -D^{-1}(L + U), \tag{2.3.14}$$

is called the *Jacobi iteration* matrix.

In actual computation, equation (2.3.13) needs to be written out element-wise. Suppose the vector $\underline{x}^{(k)}$ is given by

$$\underline{x}^{(k)} = \begin{bmatrix} x_1^{(k)} \\ x_2^{(k)} \\ \vdots \\ x_n^{(k)} \end{bmatrix}, \quad k = 0, 1, 2, \ldots \tag{2.3.15}$$

Then, equation (2.3.13) leads to the following iteration for the i^{th} component of $\underline{x}^{(k)}$:

$$x_i^{(k+1)} = \frac{-1}{a_{ii}}[\sum_{\substack{(j=1) \\ (j \neq i)}}^{n} a_{ij}x_j^{(k)} - b_i], \quad i = 1, 2, \ldots, n \tag{2.3.16}$$

which shows that the Jacobi iteration is quite easy to program. The only real problem is to determine an efficient test for terminating the iteration. However, in order for the Jacobi iterative method to be used, the diagonal elements of A must all be non-zero, but this requirement causes no real difficulty. The Jacobi iterative method will produce the solution $A^{-1}\underline{b}$ for arbitrary $\underline{x}^{(0)}$ if and only if all the eigenvalues of $D^{-1}(L+U)$ are less than one in absolute. Equation (2.3.13) can be rewritten as

$$\underline{x}^{(k)} = D^{-1}(\underline{b} - (L+U)\underline{x}^{(k-1)})$$

$$\underline{x}^{(k)} = D^{-1}\ \underline{b} + H\underline{x}^{(k-1)}$$

where $H = -D^{-1}(L+U)$. Now,

$$\underline{x}^{(k)} = H^k\underline{x}^{(0)} + (I + H + H^2 + \cdots + H^{k-1})D^{-1}\underline{b}.$$

If all the eigenvalues of $| H |$ are less than one, then

$$\lim_{k\to\infty} H^k\underline{x}^{(0)} = 0$$

and

$$\lim_{k\to\infty} \sum_{i=0}^{k-1} H^i = (I-H)^{-1}$$

The sum is convergent. Therefore,

$$\lim_{k\to\infty} \underline{x}^{(k)} = (I-H)^{-1}D^{-1}\underline{b}$$

substituting the value of H we get,

$$\lim_{k\to\infty} \underline{x}^{(k)} = (I + D^{-1}(L+U))^{-1}D^{-1}\underline{b}$$

$$\lim_{k\to\infty} \underline{x}^{(k)} = (D+L+U)^{-1}\underline{b}$$

$$\lim_{k\to\infty} \underline{x}^{(k)} = A^{-1}\underline{b}$$

There are many other, much faster, iterative methods besides Jacobi, the most important being, the Gauss-Seidel method (see Norton [61]) , the Successive Overrelaxation (S.O.R.) method, Symmetric Successive Overrelaxation (S.S.O.R.) method, for which a good advanced reference is Varga [78]. Finally, arranged in a tabular form, we shall summarize the merits of iterative methods compared with the elimination methods:

Advantages

- Probably more efficient for large order systems.
- Implementation is simpler.
- Advantages can be taken of a known approximate solution, if one exists.
- Low accuracy solutions can be obtained quickly.
- Where the equation have a repetitive form, their coefficients need not be stored but can be generated.
- In case of sparse matrices only:
  - Less storage space is required for an iterative solution,
  - The storage requirement is more easily defined in advance,
  - The order of specification of the variables is not, usually, important.

Disadvantages

- Additional right hand sides are not easily processed.
- Convergence, even if assured, may be slow and so the amount of work is not predictable.
- The time and accuracy of the result depends on a judicious choice of parameters.

- If the convergence rate is poor, the results must be interpreted with caution.
- No advantage in time per iteration can be gained if the coefficient matrix is symmetric. For the elimination method the time can be halved.

All the iterative methods are similar to $\underline{x}^{(k+1)} = H\underline{x}^{(k)} + \underline{b}$, so that if H is a $(n*n)$ full matrix, the number of multiplications is of $O(n^2)$ per iteration. Since this number for elimination methods is of $O(\frac{n^3}{3})$, an iterative method for solving the equation (2.3.1) is likely to be viable if the number of iterations is less than $\frac{n}{3}$ (or less than $\frac{n}{6}$ for a symmetric matrix A).

The iterative method represented by equation (2.3.7) can be easily solved if M^{-1} exists i.e. M is nonsingular. If M is a diagonal matrix then the resulting iterative method is called the *Simultaneous Displacement* (SI) iterative method. The Jacobi and Richardson iterative methods are examples of simultaneous displacement methods. If M is a lower triangular matrix then the resulting iterative method is known as the *Successive Displacement* (SU) iterative method. The Gauss-Seidel, S.O.R. and S.S.O.R. iterative methods are all examples of such methods. In the case of the simultaneous displacement methods the order in which the elements of $\underline{x}^{(k)}$ are updated is unimportant, where as in the case of successive displacement methods a sequential modification order of the evaluation of the unknowns $x_i^{(k)}$ $i = 1, 2, \ldots, n$ is imposed.

2.3.3 Simultaneous Displacement (SI) methods

Consider the Jacobi iterative method explained in section (2.3.2) represented by the equation (2.3.13). which can also be written as,

$$D\underline{x}^{(k)} = \underline{b} + D\underline{x}^{(k-1)} - D\underline{x}^{(k-1)} - (L+U)\underline{x}^{(k-1)},$$

$$D\underline{x}^{(k)} = \underline{b} + D\underline{x}^{(k-1)} - (D+L+U)\underline{x}^{(k-1)}.$$

Rearranging we get,

$$\underline{x}^{(k)} - \underline{x}^{(k-1)} = D^{-1}(\underline{b} - A\underline{x}^{(k-1)}) \qquad (2.3.17)$$

illustrating that the difference between successive approximations is proportional to the difference between the true solution $\underline{x}^{(t)}$ and the $\underline{x}^{(k-1)}$ estimate to the solution. If we define "α" a scalar and $\underline{r}^{(k-1)} = D^{-1}(\underline{b} - A\underline{x}^{(k-1)})$ the convergence to the exact solution $\underline{x}^{(t)}$ can be accelerated by the formula,

$$\underline{x}^{(k)} = \underline{x}^{(k-1)} + \alpha\underline{r}^{(k-1)} \qquad (2.3.18)$$

This is known as the accelerated simultaneous displacement method. The choice of acceleration parameter "α" is clearly important, and for some types of matrices derived from differential equations bounds can be placed on it.

An alternative to equation (2.3.18) which is more sensitive to the round off errors in approximation theory is to define "$\alpha_i \quad i = 1, 2, \ldots$" a series of vector constants for each iteration giving the Richardson iterative method,

$$\underline{x}^{(k)} = \underline{x}^{(k-1)} + \alpha_i\underline{r}^{(k-1)} \qquad (2.3.19)$$

The simultaneous displacement method represented by equation (2.3.18) is called stationary because the error of approximation is always affected by the same amount "α", whereas in the case of equation (2.3.19) the error at each iteration is not the same because of the different "α_i", the methods are termed as non-stationary. Consequently the choice of "α_i" is generally more difficult than in the case of stationary methods.

2.3.4 Successive Displacement (SD) methods

If we substitute $A = D + L + U$ and, $M = (D + L)$ and $N = U$ then equation (2.3.1) can be written as,

$$(D + L)\underline{x}^{(k)} = \underline{b} - U\underline{x}^{(k-1)} \qquad (2.3.20)$$

or

$$\underline{x}^{(k)} = (D+L)^{-1}\underline{b} - (D+L)^{-1}U\underline{x}^{(k-1)} \tag{2.3.21}$$

and is known as the Gauss-Seidel iterative method. This iterative method is superior to the Jacobi iterative method because $(D+L)$ is a lower triangular matrix and so the latest estimates of the components in $\underline{x}^{(k)}$ can be incorporated to produce the remaining components. Implicitly using the most recent values implies that $\underline{x}^{(k-1)}$ can be overwritten with the $\underline{x}^{(k)}$ and hence requires the storage of only one approximation vector instead of two needed by the Jacobi iterative method.

As for the simultaneous displacement methods, an acceleration parameter "ω" can be introduced. From equation (2.3.21)

$$D(\underline{x}^{(k)} - \underline{x}^{(k-1)}) = \underline{b} - L\underline{x}^{(k)} - U\underline{x}^{(k-1)} - D\underline{x}^{(k-1)} \tag{2.3.22}$$

or

$$(\underline{x}^{(k)} - \underline{x}^{(k-1)}) = D^{-1}(\underline{b} - L\underline{x}^{(k)} - U\underline{x}^{(k-1)} - D\underline{x}^{(k-1)}). \tag{2.3.23}$$

If we introduce the acceleration parameter "ω" then,

$$(\underline{x}^{(k)} - \underline{x}^{(k-1)}) = \omega D^{-1}(\underline{b} - L\underline{x}^{(k)} - U\underline{x}^{(k-1)} - D\underline{x}^{(k-1)}) \tag{2.3.24}$$

$$(I + \omega D^{-1}L)\underline{x}^{(k)} = \omega D^{-1}\underline{b} + \omega D^{-1}(-U - D + \omega^{-1}D)\underline{x}^{(k-1)} \tag{2.3.25}$$

or

$$\underline{x}^{(k)} = (I + \omega D^{-1}L)^{-1}\left\{\omega D^{-1}\underline{b} + [(-\omega D^{-1}U + (1-\omega)]\underline{x}^{(k-1)}\right\} \tag{2.3.26}$$

The iterative method represented by equation (2.3.26) is known as the Successive Over-relaxation (S.O.R.) method.

2.4 Convergence of iterative methods

The methods discussed in the section (2.3) can be written in the general iterative form

$$\underline{x}^{(k)} = M\underline{x}^{(k-1)} + \underline{c} \tag{2.4.1}$$

$$\begin{aligned} &M = (I + D^{-1}A),\ \underline{c} = D^{-1}\underline{b} \text{ Jacobi method} \\ &M = (D+L)^{-1}U,\ \underline{c} = (D+L)^{-1}\underline{b} \text{ Gauss-Seidel method} \\ &M = (I - \omega D^{-1}L)^{-1}[(1-\omega) - \omega D^{-1}U],\ \underline{c} = (I - \omega D^{-1}L)^{-1}\omega D^{-1}\underline{b} \text{ S.O.R. method} \end{aligned} \tag{2.4.2}$$

The error vector $\underline{e}^{(k)}$ associated with the k^{th} iterate $\underline{x}^{(k)}$ is,

$$\underline{e}^{(k)} = \underline{x}^{(t)} - \underline{x}^{(k)} \tag{2.4.3}$$

and with $\underline{x}^{(t)}$ the exact solution substituted in equation (2.4.1) is given as,

$$\underline{x}^{(t)} = M\underline{x}^{(t)} + \underline{c} \tag{2.4.4}$$

subtracting equation (2.4.1) from equation (2.4.4)

$$\underline{x}^{(t)} - \underline{x}^{(k)} = M(\underline{x}^{(t)} - \underline{x}^{(k-1)}) \tag{2.4.5}$$

or,

$$\underline{x}^{(t)} - \underline{x}^{(k-1)} = M(\underline{x}^{(t)} - \underline{x}^{(k-2)})\,. \tag{2.4.6}$$

Hence

$$\underline{e}^{(k)} = M\underline{e}^{(k-1)} \tag{2.4.7}$$

and consequently,

$$\underline{e}^{(k)} = M\underline{e}^{(k-1)} = \ \dots\ = M^k\underline{e}^{(0)} \tag{2.4.8}$$

where $\underline{e}^{(0)} = \underline{x}^{(t)} - \underline{x}^{(0)}$ with $\underline{x}^{(0)}$ an arbitrary but known set of initial values. The sequence of iterative values $\underline{x}^{(1)}, \underline{x}^{(2)}, \ldots, \underline{x}^{(k)}$ will converge to $\underline{x}^{(t)}$ as $k \rightarrow \infty$ if

$$\lim_{k \rightarrow \infty} \underline{e}^{(k)} = 0.$$

From equation (2.4.8) this can happen if and only if $M^k \underline{e}^{(0)} \rightarrow \underline{o}$ (the null vector) as $k \rightarrow \infty$. This condition, as the following theorem states will be true if and only if $S(M) < 1$.

Theorem 2.1 *If A is an $(n * n)$ matrix, then A is convergent if and only if $S(A) < 1$, where $S(A)$ is the spectral radius of the matrix A.*

For proof see Varga [78], page 13.

Corollary 2.1 *A sufficient condition for the convergence of the system in equation (2.4.1) is that,*

$$\|M\| < 1.$$

Since $S(M) \leq \|M\|$, in most cases it happens that $\|M\| > 1$ but $S(M) < 1$ which guarantees the convergence of the iterative process, therefore $\|M\| < 1$ is a sufficient condition but not a necessary condition.

The error vectors of equation (2.3.18) the simultaneous displacement iterative methods and (2.3.19), Richardson's iterative method, satisfy,

$$\underline{e}^{(k)} = (I - \alpha A)^k \underline{e}^{(0)} \tag{2.4.9}$$

and

$$\underline{e}^{(k+1)} = \prod_{i=0}^{k} (I - \alpha_i A) \underline{e}^{(0)} \tag{2.4.10}$$

indicating the stationary and non-stationary nature of the methods.

If a method converges, it might converge too slowly to be of practical value. Therefore, it is necessary to determine the effectiveness of an iterative method. To accomplish this both the work required per iteration and the number of iterations necessary for convergence to a specified accuracy must be considered. The usual approach is to iterate until the norm of the vector $\underline{e}^{(k)}$ is reduced to less than some predetermined factor (tolerance) "ε" of the initial vector $\underline{e}^{(0)}$.

From equation (2.4.8) and using (2.2.20) we have

$$\|\underline{e}^{(k)}\| = \|M^k\underline{e}^{(0)}\| \le \|M^k\|.\|\underline{e}^{(0)}\|. \tag{2.4.11}$$

Then if and only if $\underline{e}^{(0)} \neq \underline{0}$, $\|\underline{e}^{(k)}\| < \varepsilon\|\underline{e}^{(0)}\|$.

By theorem (2.1) we know that $\|M^k\| \to 0$ as $k \to \infty$ if $S(M) < 1$. Therefore equation (2.4.11) can be satisfied by choosing k sufficiently large such that

$$\|M^k\| < \varepsilon. \tag{2.4.12}$$

If k is large enough so that $\|M^k\| < 1$, it follows that equation (2.4.12) is equivalent to,

$$k \ge \frac{-\log\varepsilon}{|-\frac{1}{k}\log\|M^k\||}. \tag{2.4.13}$$

From the inequality (2.4.13) a lower bound for the number of iterations for the iterative methods can be obtained.

Young, in [82], concluded that the *average* rate of convergence after k iterations for any convergent iterative method in the form of equation (2.4.1) is the quantity,

$$R_k(M) \equiv -k^{-1}\log\|M^k\| \tag{2.4.14}$$

or

$$R_k(M) \equiv -\frac{\log\|M^k\|}{k}. \tag{2.4.15}$$

Note that the number of iterations is inversely proportional to the average rate of convergence. If $R_k(M_1) < R_k(M_2)$, then M_2 is iteratively faster for k iterations than M_1. The *asymptotic* rate of convergence of equation (2.4.1) is defined as,

$$R\ (M) \equiv \lim_{k\to\infty} R_k(M) = -\log S(M). \tag{2.4.16}$$

Equation (2.4.16) holds since,

$$\lim_{k\to\infty} \|M^k\|^{\frac{1}{k}} = -\log S(M). \tag{2.4.17}$$

which is a result proved by Young, in [82] on page 87. $R(M)$ is referred to as the rate of convergence.

It is normal for iterative processes to converge slowly in substantial or large problems corresponding to $S(M)$ being only slightly less than 1 and consequently the rate of convergence is nearly zero.

To obtain an estimate of the number of iterations, k, upon replacing $\|M^k\|$ by $[S(M)]^k$ in equation (2.4.12) $\|M^k\| \le \varepsilon$ we see that

$$\varepsilon \ge [S(M)]^k \tag{2.4.18}$$

and hence

$$k \ge \frac{-\log\varepsilon}{-\log S(M)} \ge \frac{-\log\varepsilon}{R\ (M)} \tag{2.4.19}$$

giving an upper bound for the number of iterations required for the algorithm to converge.

Equation (2.4.1) can be written in the norm notation represented by equation (2.2.20) for any p-norm as,

$$\|\underline{e}^{(k)}\|_p \le \|M^k\|_p \|\underline{e}^{(0)}\|_p \tag{2.4.20}$$

Thus $\|M^k\|_p$ gives a measure by which the norm of the error has been reduced after k iterations.

Different iterative methods produce different rates of convergence and this together with the amount of work required for each iteration dictates which iterative method is used for a particular problem. A complicated iterative method may converge significantly faster than a simple iterative method, but may involve much work per iteration; unless the work done in two competing methods is approximately the same. The simple iterative method may out perform a complex iterative method in terms of total number of operations. Further the choice of initial approximation $\underline{x}^{(0)}$ is very important too. A bad choice of $\underline{x}^{(0)}$ may force even an efficient method to perform a large amount of computations.

2.5 Partial differential equations

Partial differential equations represent functions of more than one independent variable. Most of the scientific problems have mathematical models that are governed by second-order equations, i.e., the highest order of the derivative being second. The most general form of the two dimensional second order partial differential equations can be written as:

$$A\frac{\partial^2 u}{\partial x^2} + B\frac{\partial^2 u}{\partial x \partial y} + C\frac{\partial^2 u}{\partial y^2} + D\frac{\partial u}{\partial x} + E\frac{\partial u}{\partial y} + Fu + G = 0. \tag{2.5.1}$$

The independent variables x and y may both be space coordinates, or one may be a space coordinate and the other a time variable. $A, B, C, \cdots, G$ may be functions of the independent variables and of the dependent variable u. Equation (2.5.1) can be classified into three particular types depending on the sign of the discriminant $B^2 - 4AC$. Therefore it is said to be:

1. Ellipitic if $B^2 < 4AC$,

2. Parabolic if $B^2 = 4AC$,

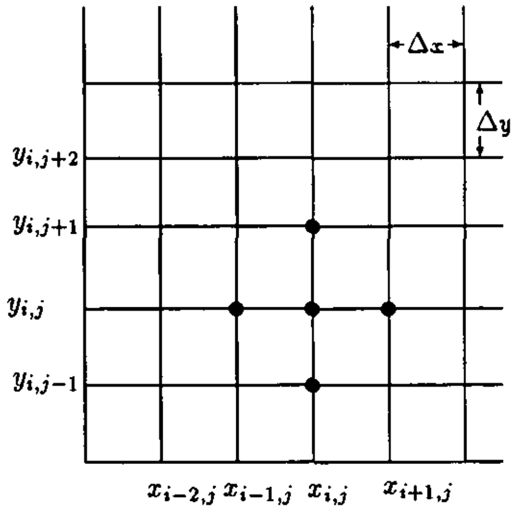


Figure 2.1: Grid.

3. Hyperbolic if $B^2 > 4AC$.

If u is a continuous function of two independent variables x and y then

$$\nabla^2 u = (\frac{\partial u^2}{\partial x^2} + \frac{\partial u^2}{\partial y^2}) = 0 \tag{2.5.2}$$

is called Laplace's equation and is an example of an elliptic partial differential equation. Laplace's equation describes the velocity potential for the steady flow of an incompressible non-viscous fluid and is the mathematical expression of the physical law which states that the rate at which such a fluid enters a given region is equal to the rate at which it leaves the region. Such problems are referred to as "boundary value problems", since the dependent variable is usually specified on the boundary of the region underconsideration. To solve the Laplace equation on a region in the xy-plane, we subdivide the region with equally spaced lines parallel to the x- and y-axis. Let $h = \Delta x = \Delta y$ be the grid spacing in the x- and y-direction as depicted in figure (2.1). If the function $U(x)$ is continuous, the solution can be approximated by applying at each grid point the Taylor series to obtain,

$$U(x_n+h) = U(x_n)+hU'(x_n)+\frac{h^2U''(x_n)}{2!}+\frac{h^3U'''(x_n)}{3!}+\frac{h^4U^{iv}(x_n)}{4!}+\cdots \tag{2.5.3}$$

$$U(x_n - h) = U(x_n) - hU'(x_n) + \frac{h^2 U''(x_n)}{2!} - \frac{h^3 U'''(x_n)}{3!} + \frac{h^4 U^{iv}(x_n)}{4!} - \cdots \quad (2.5.4)$$

Equation (2.5.3) and (2.5.4) are known as the forward and backward difference equations. Adding (2.5.3) and (2.5.4) and truncating the Taylor series we obtain,

$$\frac{[u(x_n + h) + u(x_n - h) - 2u(x_n)]}{h^2} = u''(x_n) + O(h^2) \quad (2.5.5)$$

which can be rewritten as,

$$\frac{u_{n+1} + u_{n-1} - 2u_n}{h^2} = u_n'' + O(h^2) \quad (2.5.6)$$

$O(h^2)$ means that the error approaches proportionality to h^2 as $h \rightarrow 0$. When U is a function of both x and y, we derive the second-order partial derivative with respect to x, $\frac{\partial^2 u}{\partial x^2}$, by holding y constant and evaluating the function at three points where x equals x_n, $x_n + h$ and $x_n - h$. The partial derivative of U with respect to y is similarly computed, holding x constant.

To solve the Laplace equation on a region in the xy-plane, we subdivide the region with equal spaced lines parallel to the x- and y-axis. Consider a portion near (x_i, y_j) where we will approximate

$$\nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}. \quad (2.5.7)$$

Replacing the derivative by difference quotients which approximates the derivative at the point (x_i, y_j), see figure (2.1),

$$\begin{aligned} \nabla^2 u_{(i,j)} \approx \; & \frac{u(x_{i+1}, y_j) - 2u(x_i, y_j) + u(x_{i-1}, y_j)}{(\Delta x)^2} + \\ & \frac{u(x_i, y_{j+1}) - 2u(x_i, y_j) + u(x_i, y_{j-1})}{(\Delta y)^2} \end{aligned} \quad (2.5.8)$$

or

$$\nabla^2 u_{i,j} \approx \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{(\Delta x)^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{(\Delta y)^2}. \quad (2.5.9)$$

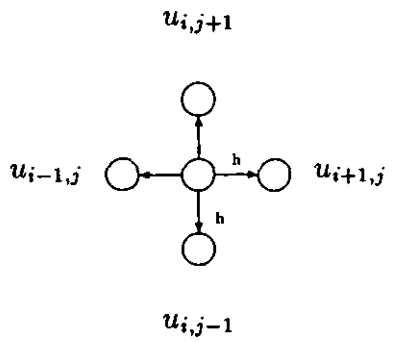


Figure 2.2: Laplace computational molecule.

Note that five points are involved in the relationship of equation (2.5.9). Points to the right, left, above and below the central point. Since $\Delta x = \Delta y = h$ equation (2.5.9) can be rewritten as,

$$\nabla^2 u_{i,j} \approx \frac{u_{i+1,j} - 4u_{i,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1}}{h^2}. \tag{2.5.10}$$

The relation can be represented in the form of a computational molecule as shown in figure (2.2).

From the molecule we can write

$$u_{ij} \approx \frac{u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1}}{4}. \tag{2.5.11}$$

Equation (2.5.11) can be written as an iterative procedure using the subscripts,

$$u^{(k+1)}_{ij} = \frac{u^{(k)}_{i+1,j} + u^{(k)}_{i-1,j} + u^{(k)}_{i,j+1} + u^{(k)}_{i,j-1}}{4}. \tag{2.5.12}$$

Adding and subtracting $u^{(k)}_{i,j}$ on the right hand side we get,

$$u^{(k+1)}_{ij} = u^{(k)}_{ij} + [\frac{u^{(k)}_{i+1,j} + u^{(k)}_{i-1,j} + u^{(k)}_{i,j+1} + u^{(k)}_{i,j-1} - 4u^{(k)}_{i,j}}{4}]. \tag{2.5.13}$$

The term in brackets is known as the "residual" and will be zero when the solution is achieved. The bracketed term is an adjustment to the old approximation, which gives the improved approximation. Equation (2.5.13) represents the Jacobi iterative method. If the new approximation of the components of

$\underline{u}^{(k+1)}$ is incorporated to produce the remaining components then equation (2.5.13) can be rewritten as,

$$u_{ij}^{(k+1)} = u_{ij}^{(k)} + [\frac{u_{i+1,j}^{(k)} + u_{i-1,j}^{(k+1)} + u_{i,j+1}^{(k)} + u_{i,j-1}^{(k+1)} - 4u_{i,j}^{(k)}}{4}] \qquad (2.5.14)$$

to represent the Gauss-Seidel iterative method. If a larger adjustment to the old approximation is added, faster convergence may result. This is known as overrelaxation. If 'ω' is the overrelaxation factor then the equation (2.5.14) can be written as,

$$u_{ij}^{(k+1)} = u_{ij}^{(k)} + \omega[\frac{u_{i+1,j}^{(k)} + u_{i-1,j}^{(k+1)} + u_{i,j+1}^{(k)} + u_{i,j-1}^{(k+1)} - 4u_{i,j}^{(k)}}{4}] \qquad (2.5.15)$$

to give the S.O.R. iterative method. Maximum acceleration is obtained for a optimum value of 'ω', which lies between 1 and 2.

Chapter 3

Design of Systolic Arrays

This chapter briefly explains the systolic array concepts introduced in late 1970s and early 1980s, in the field of Computer Science and Electrical engineering. A systolic array represents a computing engine with symmetric, regular organisation of simple processing elements with pipelined data flow and multiprocessing. The algorithms implemented on a systolic array are termed as systolic algorithms. The advances in the VLSI technology and the suitability of systolic arrays, due to their simple and modular structure (a key feature for VLSI designs) triggered the research interest in this area. Later in the chapter different systolic designs for the matrix vector multiplication algorithm are explained. These designs are then used to realise the systolic array designs for the basic iterative methods to solve linear systems of equations.

3.1 Introduction

Systolic arrays are the result of advances in semiconductor technology and applications that require extensive computation. An application is said to be compute bound if the number of computing operations is larger than the total number of input and output elements, otherwise it is termed as I/O bound. As regards to structure they resemble a grid in which each point corresponds to a processor and a line corresponds to a link between processors. Systolic architectures are regular and modular, and hence adopt well to VLSI (Very Large Scale Integration) architectures for area efficient layouts. The term systolic in the context of systolic arrays means that pipelined computations take place along all dimensions of the array and result in a high throughput. Systolic arrays schedule computations in such a way that a data item is not only used when it is input but is reused as it moves through the pipelines in the array. This results in balancing the processing and I/O bandwidths. For the excellent introduction to the area see Kung and Leiserson [46], Kung [43], Fisher and Kung [22], Quinton and Robert [64], Evans [15], Megson [56], Petkov [16], Fortes and Wah [26], Quinton [62].

The VLSI technology permits the manufacturing of regular and modular layouts at reasonable cost for small and simple interconnection patterns, replicated in one or two dimensions. Systolic arrays have triggered extensive related work and research in the areas of processor array architecture, algorithm design and analysis, and parallel programming.

The systolic array design can either be general purpose or special purpose. Generally a systolic array is designed to implement a specific algorithm. The design philosophy of a special purpose VLSI chip is explained in the paper by Foster and Kung [27]. A design of systolic arrays which is capable of executing more than one algorithm is a promising architecture, Warp [1985] by H. T. Kung and CHiP [1982] by Laurence Snyder are examples of such architectures. These two architectures use different approaches, for the general

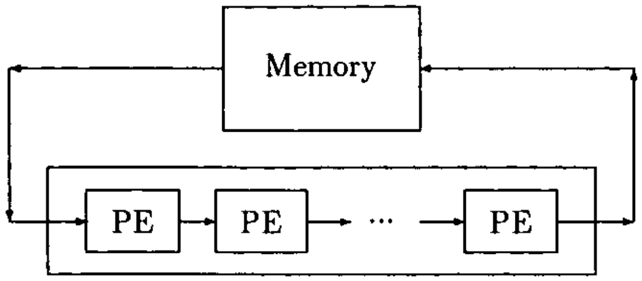


Figure 3.1: Linear systolic array.

purpose systolic design.

3.2 Systolic array

A systolic array is a combination of simple processing elements (PEs), each capable of performing some simple operations, connected to each other by communication links. The basic principle of a systolic array is shown in figure (3.1). This architecture differs from that of Von Neumann, where the single processing element is replaced by an array of processing elements (cells). This results in a higher computation throughput without increasing the memory bandwidth. Suppose that the I/O bandwidth between the host computer and the processing element is 10 million bytes per second and assume that at least two bytes of data are read from or written to the host computer per operation. Then the maximum rate will be 5MOPS (million operations per second), irrespective of the speed of the PE. With the systolic array consisting of say 5 processing elements the peak rate of 25 MOPS is possible, if multiple computations are performed per I/O access.

The array is connected to a host computer. The data stream flows from the host computer memory in a rhythmic fashion, passing through each processing element before returning to the host computer memory. This rhythmic data flow resembles the blood circulation from and to the heart (the host memory), and hence gets the name "*systolic*".

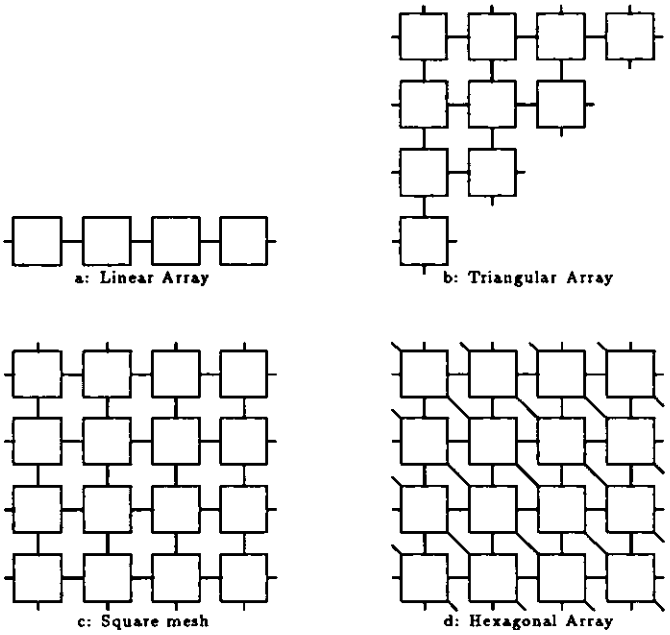


Figure 3.2: Systolic structures.

The cells that connect the array to the host computer are called the *boundary* cells, and communication with the outside world occurs only through them. The systolic arrays can assume different structures, figure (3.2) shows various systolic array configurations. The processing elements in these arrays are interconnected with each other in vertical, horizontal and diagonal directions. The linear array can be uni-directional or bi-directional according to the data flow, i.e. the number of data paths and the flow direction. In a uni-directional linear array data streams flow in the same direction. The processing elements can communicate by one, two or three data paths. In the case of a bi-directional linear array two data streams flow in opposite directions. The data flow for the matrix vector multiplication (mvm) algorithm for the uni-directional and bi-directional processor arrays are shown in figures (3.3) and (3.4) respectively. In a uni-directional array the "y" values do not move in the array, hence are termed as stationary, whereas "x" and "a" values are nonstationary. The uni-directional and bi-directional implementations of the matrix vector multiplication execute the algorithm in the same time but the uni-directional linear

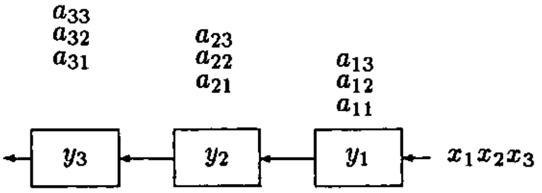


Figure 3.3: Uni-directional systolic array.

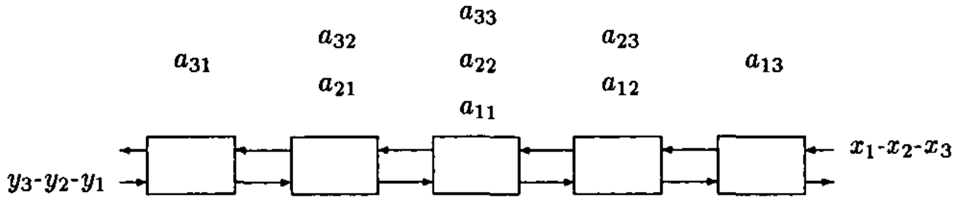


Figure 3.4: Bi-directional systolic array.

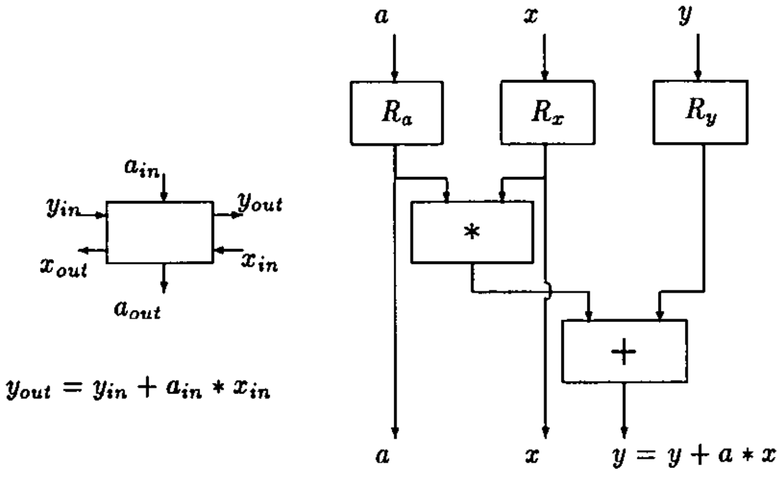
array uses half the number of processors compared to the bi-directional linear array. However the uni-directional implementation uses resident data and hence additional load/store times are required for the total processing time.

The simplest cell used in the matrix-vector multiplication is called the inner product step processor, as described below.

The single operation common to all the algorithms is the inner product step, defined as,

$$y = y + (a * x).$$

The processor capable of performing the inner product step is called inner product step processor. The processor consists of three registers R_a, R_x, R_y, a multiplier and an adder. Each register has two connections, one for input and one for output. On each cycle the data on the input lines represented by a, x, y is shifted into R_a, R_x and R_y respectively; $R_y \leftarrow R_y + (R_a * R_x)$ is computed; and the input values of R_a, R_x and newly computed R_y are output on the output lines denoted by a, x and y respectively. All outputs are latched and the logic is clocked so that when one processor is connected to another,



IPS representation (b) IPS Architecture

Figure 3.5: IPS processor.

the changing output of one during a unit time interval will not interfere with the input to another during this time interval. Figure (3.5) shows such an processor.

The systolic arrays can be classified as pure or semi systolic arrays (see Bekakos and Evans [6], Leiserson [48]).

3.2.1 Semi systolic designs

A design is said to be a Semi systolic design if the data is globally communicated, the array is then termed a semi systolic. An example of polynomial multiplication is illustrated below to describe such a design (see Margaritis [51]).

Given two polynomials $f(x)$ and $g(x)$ of degree n their product $f(x)g(x)$ is a polynomial of degree $2n$. For simplicity we assume that our polynomials are of degree two and are defined as,

$$f(x) = a_0x^2 + a_1x + a_2 \text{ and}$$

$$g(x) = b_0x^2 + b_1x + b_2$$

The product

$$f(x)g(x) = c_0x^4 + c_1x^3 + c_2x^2 + c_3x + c_4.$$

where the coefficients are defined as,

$$\begin{aligned} c_0 &= a_0b_0 \\ c_1 &= a_0b_1 + a_1b_0 \\ c_2 &= a_0b_2 + a_1b_1 + a_2b_0 \\ c_3 &= a_1b_2 + a_2b_1 \\ c_4 &= a_2b_2 \end{aligned} \tag{3.2.1}$$

From the equation (3.2.1) it is obvious that each of the coefficients b_0, b_1, and b_2 are multiplied by all the coefficients of $f(x)$. Notice that the coefficient b_0 meets the first coefficient a_0, then a_1 and finally a_2. Similarly for b_1, but starts with the calculation of c_1, and a similar pattern follows, i.e. b_2 starts with the calculation of c_2. The calculation at most involves only two arithmetic operations, i.e. multiplication and addition to produce the new coefficients as a series of inner products.

A straight forward systolic design, that maps the computation of the product onto a processing structure is shown in figure (3.6). A global clock synchronises the computations in the system, and has a time cycle (step, unit) which can accommodate the most complex function performed by a processor, including the data transfer. In each time cycle the processors in the array simultaneously perform their I/O and then execute their operation, i.e. inner product, see figure (3.6). The processor consists of an adder and multiplier.

To solve a polynomial multiplication problem of degree n the array consists of $n + 1$ processors, (for the example discussed 3 processors are used as shown in figure (3.6)). The coefficients of $f(x)$, i.e. a_0, a_1 and a_2 are stored in their respective processors. The coefficients of $g(x)$ form a data stream and is broadcast to all the processors, one coefficient per computational step. At time unit 1, b_0 is broadcast to all the processors and c_0 enters the right most processor

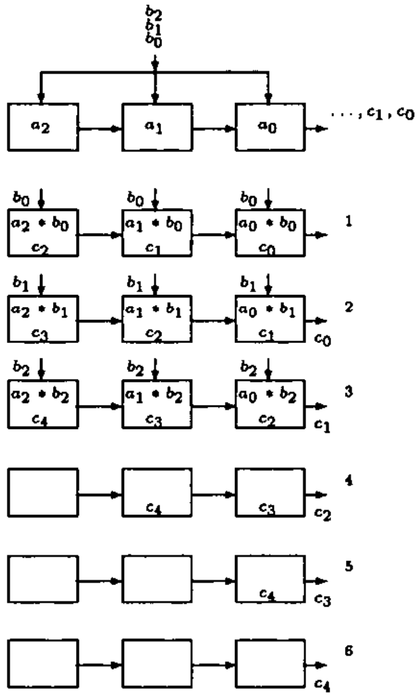


Figure 3.6: Polynomial multiplication on a Semi systolic array.

and initially contains zero. The computation of resulting coefficients is obtained by accumulating the partial products in the pipeline. For example, c_1 is computed in two steps. First $a_1 b_0$ is calculated in the middle processor, and then in the next time step this value is passed to the right hand side processor, which computes $a_0 b_1$ and is added to $a_1 b_0$ to form c_1. On the successive time step c_1 is output. It takes $(2n + 2)$ time units to complete the polynomial multiplication.

The main draw back of this design is the presence of the broadcast mechanism. This design involves large wire interconnections and therefore long communication delays, and clocking problem.

3.2.2 Pure systolic designs

If the data in the systolic array is not communicated globally then the array is classified as Pure systolic array and the design is termed as Pure systolic design. Pure systolic systems avoid long or irregular wires for data communications. The only global communication is the system clock. In figure (3.7) a pure systolic design is shown, along with the I/O data streams and the corresponding snapshots of its computations.

The broadcasting is eliminated by having two data streams b's and c's moving in opposite directions. The c stream accumulates the partial results as it passes through the array till the final result is obtained. Consecutive b's and c's are separated by two cycles. There is a delay element between the two consecutive b's and c's. For example at time step 2, the partial result for c_1, i.e. a_1b_0 is computed in the middle processor and at time step 3, no useful computation is performed by the middle processor. In time step 3 the c_1 partial result is shifted to the right processor, where a_0b_1 is computed and at time step 4, c_1 is output. This delay element is used to synchronise the data stream movement in the array. The design has no global interconnections and therefore it is easily expandable to accommodate polynomials of any degree. In this design the processor performs one useful computation every two cycles.

3.2.3 Hybrid systolic designs

The Hybrid systolic designs allow for programmable components with significant amounts of local memory and control.

3.3 Systolic architectures and VLSI

Until the advent of VLSI, the development of parallel computers with large number of processing elements had been limited by the high production cost. Using the VLSI technology in the circuits the size and the cost of processing logic, memory and communication hardware has dramatically reduced. VLSI has made feasible to produce highly parallel architectures like systolic systems (Mead and Conway [55], Haynes et al. [37]).

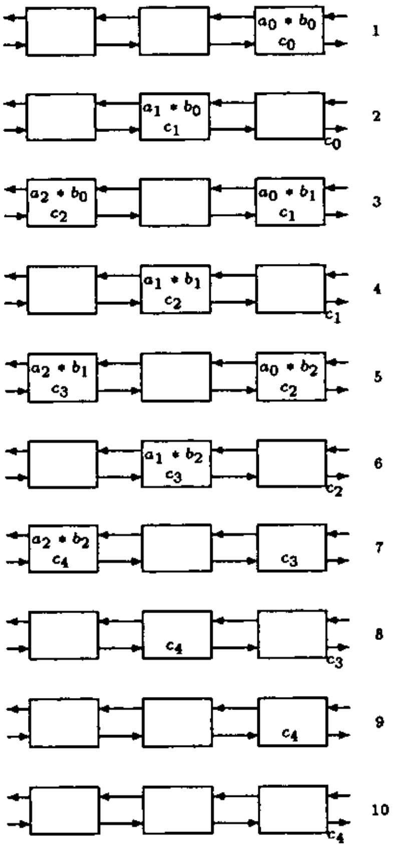


Figure 3.7: Polynomial multiplication on a Pure systolic array.

Mapping of a complex systolic system directly on a silicon wafer, is confined to a two dimensional plane, a VLSI constraint. VLSI is achieved by a combination of circuit design with high resolution photographic techniques, where it is convenient to place wires on rectangular grids, and limit the number of parallel layers of semiconductor material containing wires and circuit elements. A two dimensional graph is termed planar if it can be drawn in the plane with no arcs intersecting at places other than nodes. The designer of the VLSI runs into problems, when the number of wires and transistors approach the limits of photographic resolution, and hence becomes impossible to achieve further miniaturisation and the actual circuit becomes a key issue. Furthermore, the chip area is limited in order to maintain high yield, and the number of pins (connections to the outside world) is limited by the finite size of the chip perimeter.

Some of these constraints are eliminated when the systolic algorithms are implemented on processor arrays. For example, the actual chip design is not an issue any longer, since it is a programmable processor. Also the interconnections need not be strictly planar. However, simplicity and regularity remains the most important factor for an efficient systolic design. In systolic architectures these factors are ensured by relatively few types of simple processing elements, which mainly communicate with their nearest neighbours only (see Kung [45]).

The replication of a simple processing element in large numbers makes the design cost-effective and easy to produce. The simplicity of the cell design depends on the implementation requirement of a particular algorithm. In case a systolic system is to be implemented on a single chip, each processing element should contain a simple control logic, arithmetic unit and a few words of memory. On the other hand for a board array implementations each processing element can have a complex control logic, a high performance arithmetic unit and a couple of thousand words of memory. The processing element can even be a simple microcomputer. The trade off between simplicity and flexibility is always there in the terms of control, programming overheads and the system performance. The systolic algorithms can be classified as (see Megson [56])

1. **Hard Systolic Algorithms**

   These algorithms are hardwired programmed, i.e. the electronic circuits for control and switching logic are built as a part of overall system. Such systems implement a particular algorithm and hence are categorised as special purpose systems.

2. **Hybrid Systolic Algorithms**

   These designs permit certain degree of flexibility by microprogramming or control tags attached to the data stream. Such designs are cost effective and more desirable as the processing elements can perform various types of operations. The array structure remains the same.

3. **Soft Systolic Algorithms**

   These designs provide high degree of programmability. Using the principle of systolic computation, the algorithm is mapped onto an available parallel architecture.

3.4 Performance issues

A measure of performance for a systolic system is given by the speedup factor defined by S. Y. Kung, in [47], as,

$$s = \frac{\text{sequential computation time}}{\text{systolic computation time}} \tag{3.4.1}$$

A speedup of order n, where n is the number of processors in the systolic system indicates a successful systolic system. In the case of polynomial multiplication s is between $n/2$ and $n/4$ for the respective designs. Here n is the degree of the polynomials and the number of processors in the designs. This indicates a linear speedup and therefore a good performance of systolic algorithms. A similar measure can be obtained by the computation to communication ratio (see Dew et al. [12]).

$$c = \frac{\text{sequential total number of processors}}{\text{systolic number of boundary processors}} \tag{3.4.2}$$

A large c indicates high utilisation of the input data. In the case of polynomial multiplication, $c \cong n$, which indicates a high utilisation.

Processor utilisation is defined as

$$u = \frac{\text{total number of active processors during all computation steps}}{\text{number of processors * number of steps}} \tag{3.4.3}$$

A processor is called "active" if, at a given time step, it performs useful computation, i.e. a computation that contributes in formulation of required results, otherwise it is called "neutral" (dummy) computation and the processor's cycle is termed an "idle" cycle. In the polynomial multiplication example utilisation of processors is 1/2 and 1/4 respectively.

Area of the systolic array is measured in terms of number of processors (cells) in the design. Since the basic component of the systolic array is the IPS processor, all the other processors in the design are defined in terms of IPS processor units, i.e. equivalent of multiply-add operations that are involved. The area occupied by interconnections, registers is considered negligible, and not taken into account. For example, all the designs discussed for polynomial multiplication have area equivalent to $(n + 1)$ IPS processors.

Computation time is usually measured in terms of steps (cycles, units), where a time unit is taken to be the time required for the most complex cell function to be performed. The data transfer time is assumed to be negligible due to the locality of the communication. So an IPS step is the time required for a sequence of one multiplication and one addition to be performed.

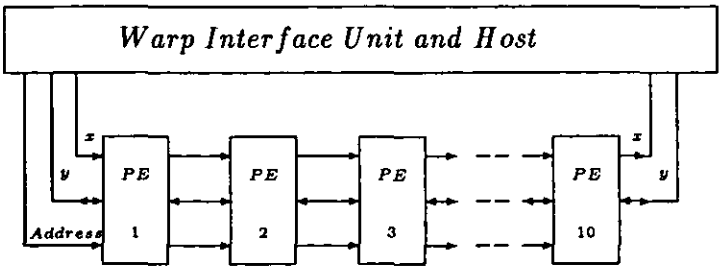


Figure 3.8: The Warp programmable, one dimensional systolic array.

3.5 WARP architecture

The Warp machine (see Kung [2]) is a systolic array computer of linearly connected cells, each of which is a programmable processor capable of performing 10 million floating point operations per second (10 MFLOPS). A typical Warp array includes ten identical cells as shown in figure (3.8), thus having a peak computation rate of 100 MFLOPS.

The first prototype was completed in 1986. The Warp machine is attached to a general purpose host computer running the Unix operating system via an interface unit. The interface unit handles the input/output between the array and the host computer. The host computer carries out high level application routines and supplies data to the Warp processor array.

The data flows through the systolic array in two channels (links) x and y. Each Warp processing element (cell) is a microprocessor with its own sequencer, program memory, data memory (4Kwords) and a high communication bandwidth (80 Mbytes/sec) to and from its neighbouring cells. A high level language called W2 is used to program the Warp. The Warp array can be extended to include more cells to accommodate applications capable of using the increased computational bandwidth.

Warp uses software support to map different algorithms onto its fixed architecture, hence tools like compilers, programming languages, operating systems, which help in implementing an efficient systolic design are required.

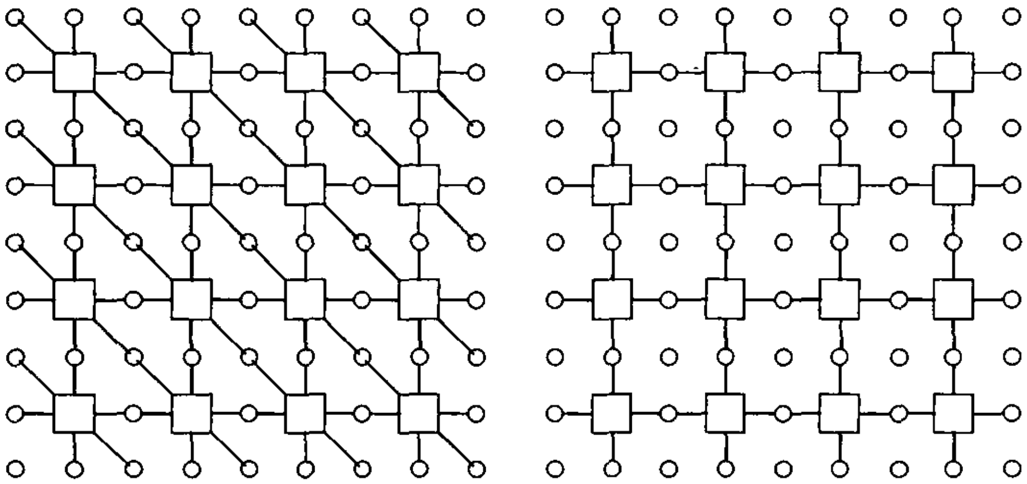
Several application programs have been developed to evaluate the Warp. The application areas are computer vision, signal processing and scientific computing. The Successive overrelaxation method was also developed for the solution of partial differential equations.

3.6 CHiP architecture

The CHiP (Configurable, Highly Parallel) computer design (see Snyder [71], Hwang and Briggs [41]) uses hardware support to reconfigure the array's interconnection pattern, a programmable lattice of switches is used for reconfiguration. The CHiP computer is a lattice of identical processing elements set into a lattice of identical switches and a controller. Figure (3.9) shows an example of the CHiP computer configured as a square systolic array. The square boxes represent the processing elements and circles represent the switches. The number of ports per switch are four to eight, while the number of ports on the cells is eight or less. Each switch in the lattice has its own local memory to store several switch settings for different configurations. The controller is responsible for loading the configuration instructions into the switch memories. Configuration instructions are loaded in parallel with the loading of programs into the processing elements, prior to the execution of the programs. The whole grid is usually square in shape with perimeter switches being connected to external storage devices.

3.7 Design improvements

Research work has been done to improve the efficiency of the conventional systolic arrays. The attempts are made to decrease both the time and number of processors using different techniques. The traditional systolic array proposed



CHiP configured as a Hex array

CHiP configured as a rectangular array

Figure 3.9: CHiP lattice of intermixed switch and processing elements.

by H. T. Kung has delays in the data stream and results in 50% utilisation of the processing element (cell). This can be seen from figure (3.12) which describes the matrix-vector multiplication (mvm) algorithm for a $(n * n)$ dense matrix with $n = 4$, i.e. the order of the problem represented by equation (3.7.1).

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} * \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} \tag{3.7.1}$$

The matrix-vector multiplication can be described by the following recurrence relation.

$$y_i^{(1)} = 0,$$

$$y_i^{(k+1)} = y_i^{(k)} + a_{ik}x_k,$$

$$y_i = y_i^{(n+1)}, \ i, k = 1, 2, \ldots, n.$$

To fully utilise the potential throughput, two independent matrix-vector computations can be performed in the same systolic array.

Folding is used to obtain less processors with more efficient processor activity with the same or less execution time. Megson and Evans, in [57], used this idea to fold the array and obtain smaller processor arrays consisting of processor elements working more efficiently. Their solution results in an irregular data flow in a subset of processors. Evans and Gusev, in [17], propose a design which offers double efficiency with regular data flow. Their design is well suited for solving a dense system.

Suros and Montagne, in [75], proposed the Fitted Diagonal method in order to halve the number of processors and improve the processor utilisation. The computational time of the algorithm remains unchanged but the hardware is halved with some modifications.

3.7.1 Kung's design

The linear array proposed by Kung for the matrix vector multiplication consists of $(2n-1)$ cells. The cells communicate to each other by the two links indicated by arrowed lines. One link communicates with the right hand side cell while the other communicates with the left hand side cell. There is an input link for the multiplicand in each cell. Input data stream and output data stream use one of these links for the I/O. The cell operations are shown in figure (3.10). The cell has three inputs, x_{in}, y_{in} and a_{in}, and two output links x_{out} and y_{out}, hence three registers R_x, R_y and R_a are present at the I/O links. The additive-multiply operation is required to perform the inner product of two vectors and is obvious from the recurrence relation. The value received on the x_{in} link is multiplied with the value on the a_{in} link and the result is added to the input on the y_{in} link. The whole operation can be written as $y_{out} = y_{in} + x_{in} * a_{in}$, and is the output. The x_{in} value is passed unaltered by the x_{out} link.

The first computation takes place after $(n-1)$ time units, the algorithm requires $(2n-1)$ time units and the last result leaves the array after n more time units. The total execution time for the matrix vector multiplication al-

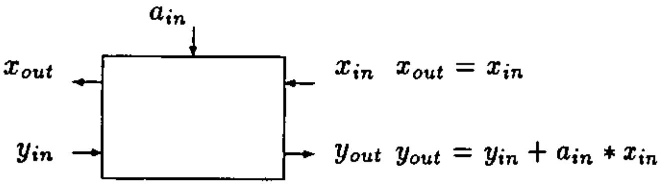


Figure 3.10: The inner product step cell.

gorithm is $(4n - 2)$ time units. The diagonals of the matrix are fed in from the top, and a new element is input to each processing element every second time step. The components of the vector x are pumped in from the right and move left wards without modification. The components of the vector y are fed through the left hand end, initially all the components of y are initialised to zero. The elements of x and y flow in the opposite directions in the array. The computation is performed when the elements of x and y reach the same cell, the new value of y continues the data flow in the right hand side cell while accumulating successively all its terms. The implementation of the algorithm is explained below. Assume that the cells are numbered by integral $1, 2, \ldots, 7$ from left to right end cell in figure (3.12) which shows the snapshots of the matrix vector multiplication algorithm. Initially all the cells contain zeros in their respective registers. Each step of the algorithm consists of the following operations :

1. **Shift**

   R_a gets the new element of the matrix A. R_x gets the contents of register R_x from the right neighbouring node (In case of processor 7 R_x gets the new component of vector x from the host computer). R_y gets the contents of the register R_y from the left neighbouring node (Processor 7 outputs its R_y contents for the host computer and the R_y in processor 1 gets zero). Note that for odd numbered time steps, only odd numbered processors are active (i.e. perform useful computation) and for even number time steps only even numbered processors are active.

2. Multiply and Add

$$R_y = R_y + (R_a * R_x)$$

At time step 1, x_1 is fed into the right most processor, i.e. processor 7 and y_1 is fed into the the left most processor, i.e. first processor, initialised to zero. At time step 2, x_1 and y_1 move one place left and right respectively, and keep moving this way. At time step 3, x_2 enters the right most processor and y_2 enters from the left most cell. At time step 4, a_{11} enters the 4^{th} processor where y_1 is updated

$y_1 \leftarrow (y_1 + a_{11} * x_1)$. Thus

$y_1 = a_{11}x_1$. At time step 5, a_{12} and a_{21} enter the 5^{th} and 3^{rd} processor respectively.

$y_1 = a_{11} * x_1 + a_{12} * x_2$

and $y_2 = a_{21} * x_1$.

At time step 6, a_{13}, a_{22} and a_{31} enter the 6^{th}, 4^{th} and 2^{nd} processor respectively.

$y_1 = a_{11} * x_1 + a_{12} * x_2 + a_{13} * x_3$,

$y_2 = a_{21} * x_1 + a_{22} * x_2$,

and $y_3 = a_{31} * x_1$.

At time step 7, a_{14}, a_{23}, a_{32} and a_{41} enter the 7^{th}, 5^{th},3^{rd} and 1^{st} processor respectively.

$y_1 = a_{11} * x_1 + a_{12} * x_2 + a_{13} * x_3 + a_{14} * x_4$,

$y_2 = a_{21} * x_1 + a_{22} * x_2 + a_{23} * x_3$,

$y_3 = a_{31} * x_1 + a_{32} * x_2,$

and $y_4 = a_{41} * x_1.$

At time step 8, y_1 is output, a_{24}, a_{33} and a_{42} enter the 6^{th}, 4^{th} and 2^{nd} processor respectively.

$y_2 = a_{21} * x_1 + a_{22} * x_2 + a_{23} * x_3 + a_{24} * x_4,$

$y_3 = a_{31} * x_1 + a_{32} * x_2 + a_{33} * x_3,$

and $y_4 = a_{41} * x_1 + a_{42} * x_2.$

At time step 9, a_{34} and a_{43} enter the 5^{th}, and 3^{rd} processor respectively.

$y_3 = a_{31} * x_1 + a_{32} * x_2 + a_{33} * x_3 + a_{34} * x_4,$

and $y_4 = a_{41} * x_1 + a_{42} * x_2 + a_{43} * x_3.$

At time step 10, y_2 is output and a_{44} enter the 4^{th} processor.

$y_4 = a_{41} * x_1 + a_{42} * x_2 + a_{43} * x_3 + a_{44} * x_4.$

3.7.2 The Evans and Bekakos design

Evans and Bekakos make use of the delays appearing in the data stream of the Kung's approach. They fold the opposite extreme ends of the matrix to form a deque with elements of the lower half of the matrix interleaved in the previous delay spaces. This technique is not a simple folding of the matrix, but involves a simultaneous rotation of the off diagonals. The systolic design for matrix vector multiply explained in section (3.7.1) and by figure (3.12), i.e. the Kung's design for a (4*4) matrix is exemplified to illustrate this approach.

The design proposed by Evans and Bekakos uses less area and time as compared to the Kung's design. The data should be rotated and folded prior to

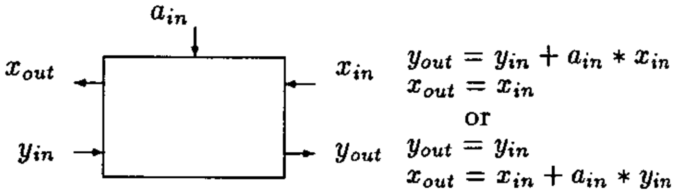


Figure 3.11: Switch inner product step cell.

processing by array, in the host computer. The elements of the product also come out in irregular order, hence the host computer must store them in the right order. The other overhead for the host computer is that the x vector needs to be repumped twice. Once in the forward direction and then in the reverse order for the 2^{nd} time, this fact is evident from figure (3.13). For a more detailed discussion see Bekakos [5]. The matrix vector multiplication for a square matrix takes $(3n - 1)$ time units and the number of IPS processors (cells) required is $(n + 1)$.

3.7.3 The Evans and Gusev design

To implement the idea of folding proposed by Evans and Gusev, the cells in the array are modified. The modified design results in a smaller processor array, circular data flow, decreased I/O time and efficiently utilised processing elements (cells). The modified cell is called the Switch Inner Product Step (SIPS) cell and is shown in figure (3.11). Each cell performs an inner product step operation as the normal cell, but the inputs to the arithmetic unit are switched on every time cycle. On one time cycle processor computes $y_{out} = y_{in} + a_{in} * x_{in}$ and passes the value $x_{out} = x_{in}$, and on alternative time cycle processor computes $x_{out} = x_{in} + a_{in} * y_{in}$ and passes the value $y_{out} = y_{in}$.

The matrix vector multiplication design using the idea of folding proposed by Evans and Gusev is shown in figure (3.14). Note that only the right hand side boundary processor communicates with the host computer. Input data stream (x and y values) enter the array from the upper link and the output data stream leaves the array from the lower link. The left hand side boundary cell is short connected, so that the output data stream becomes the input data stream to this cell.

The first computation takes place after n time units, the algorithm requires $(2n-1)$ time units and the last result leaves the array after $(n+1)$ more time units. The total execution time for the matrix vector multiply algorithm is $4n$ time units.

A comparison can be made with the number of cells used in the arrays proposed by Kung, Evans and Gusev. It shows that the number of cells is halved using the idea of folding. On the other hand the complexity of the cell is increased, i.e. the control logic and the switching circuitry for the arithmetic unit. However the gain obtained by saving the chip area, surpasses the complexity issue. The time required by this design for the matrix vector operation is two time steps more than that of the conventional design, and is not significant.

3.7.4 The Suros and Montagne design

Suros and Montagne, in [75], proposed the idea of the Fitted Diagonal Method (FDM). In this method the number of diagonals is halved and therefore halves the number of processors. If A is a diagonal matrix with even bandwidth w, then the number of diagonals can be reduced to $\frac{w}{2}$ ($\frac{(w+1)}{2}$ for odd bandwidth) by fitting into adjacent diagonal pairs.

Now we apply the FDM method to the matrix vector multiplication example described in section (3.7.1). The resulting array and the snap shots of the execution of the algorithm are shown in figure (3.15). The number of processors required are four instead of seven (not halved as the bandwidth is odd). The time required to execute the algorithm is one time step more than that of Kung's design.

The reduction in the number of processors is quite impressive. To execute the algorithm correctly the IPS cells need some minor modification. The FDM requires that the elements of vectors x and y must be kept in each processor for two time steps. The processor utilisation is also improved.

3.7.5 Comparison of different mvm designs

A comparison can be made for area and time requirements, and the utilisation of the processors, for the linear systolic matrix vector multiplication designs presented in sections (3.7.1), (3.7.2), (3.7.3) and (3.7.4). Table (3.1) shows these requirements for different designs, for brevity the designs are valid for square dense matrices which result in an odd bandwidth. The following observations can be made regarding these designs.

1. **Area**

   The area requirements for Evans and Gusev, and Suros and Montagne designs are same and are almost half (half + one) than that of Kung's design. Area required by the Evans and Bekakos design for small size problems is bit more than half but for large problems it is almost half.

2. **Time**

   The Evans and Bekakos design takes about 25% less time as compared to the Kung's design. The Evans and Gusev and Suros and Montagne designs take one and two time step more than that of the Kung's design respectively.

3. **Utilisation**

   When n is small, Suros and Montagne design performs the best. For large n, Evans and Bekakos design is the best. For large n, the utilisation for the Evans and Bekakos, Evans and Gusev, Suros and Montagne, and Kung's design are 1/3, 1/4, 1/4 and 1/8 respectively. In general, all the designs perform better than Kung's original design. The choice of design depends on a particular application, cost, time and the area

Architecture	Time	Area (IPS cells)	Utilisation
Kung	$4n-2$	$2n-1$	$n^2/((4n-2)(2n-1))$
Evans and Bekakos	$3n-1$	$n+1$	$n^2/((3n-1)(n+1))$
Evans and Gusev	$4n$	n	$1/4$
Suros and Montagne	$4n-1$	n	$n/(4n-1)$

Table 3.1: Statistics for matrix vector multiply.

requirements.

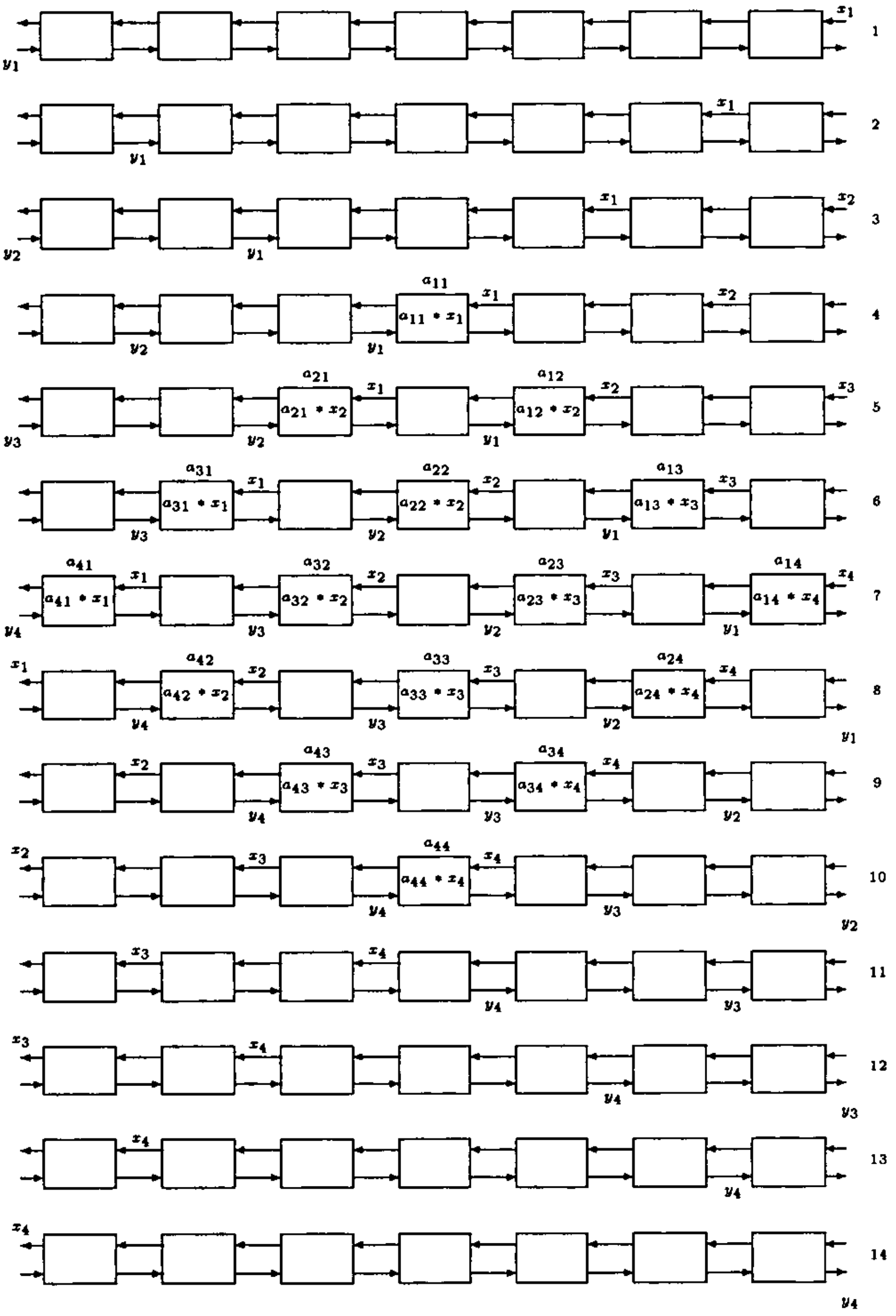


Figure 3.12: The matrix vector multiply design proposed by H. T. Kung.

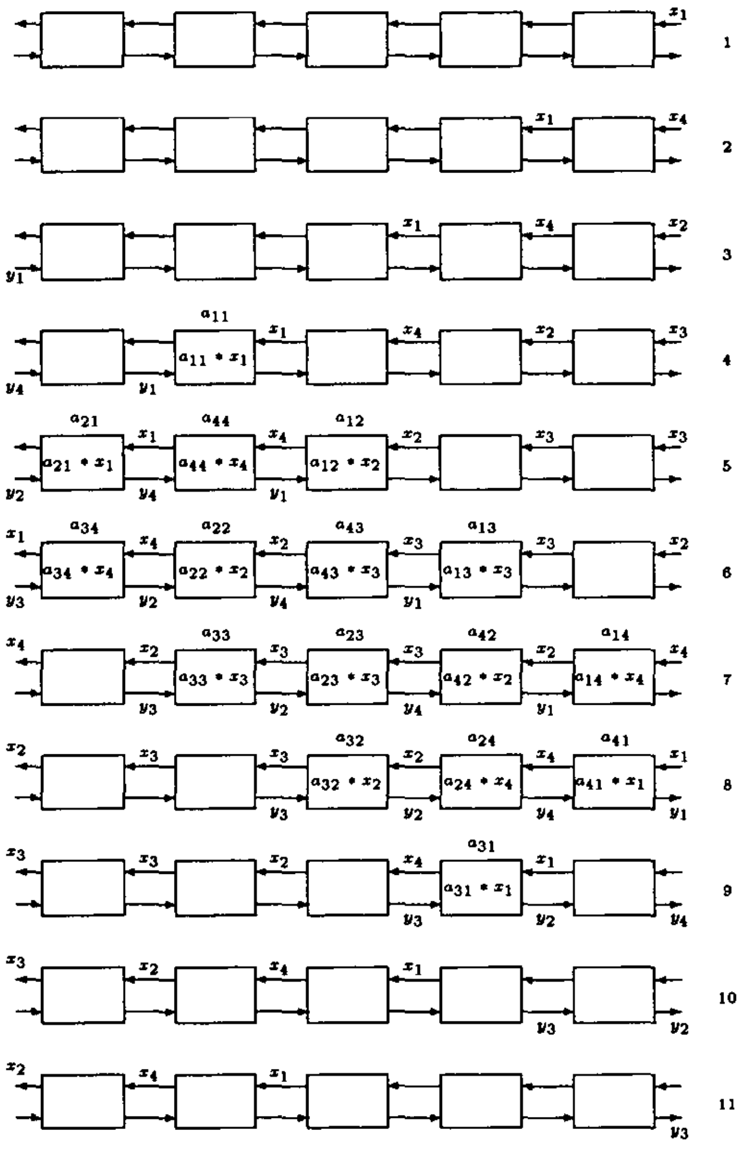


Figure 3.13: The matrix vector multiply design proposed by Evans and Bekakos.

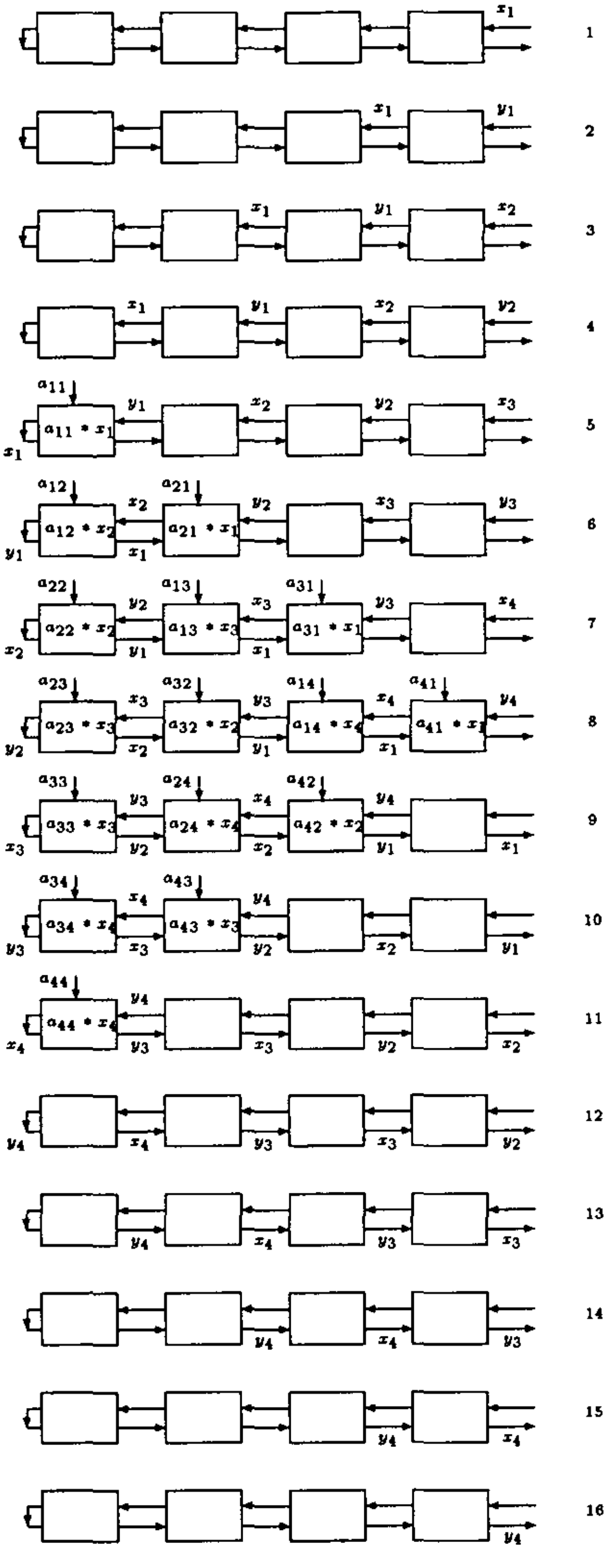


Figure 3.14: Matrix vector multiply design proposed by Evans and Gusev.

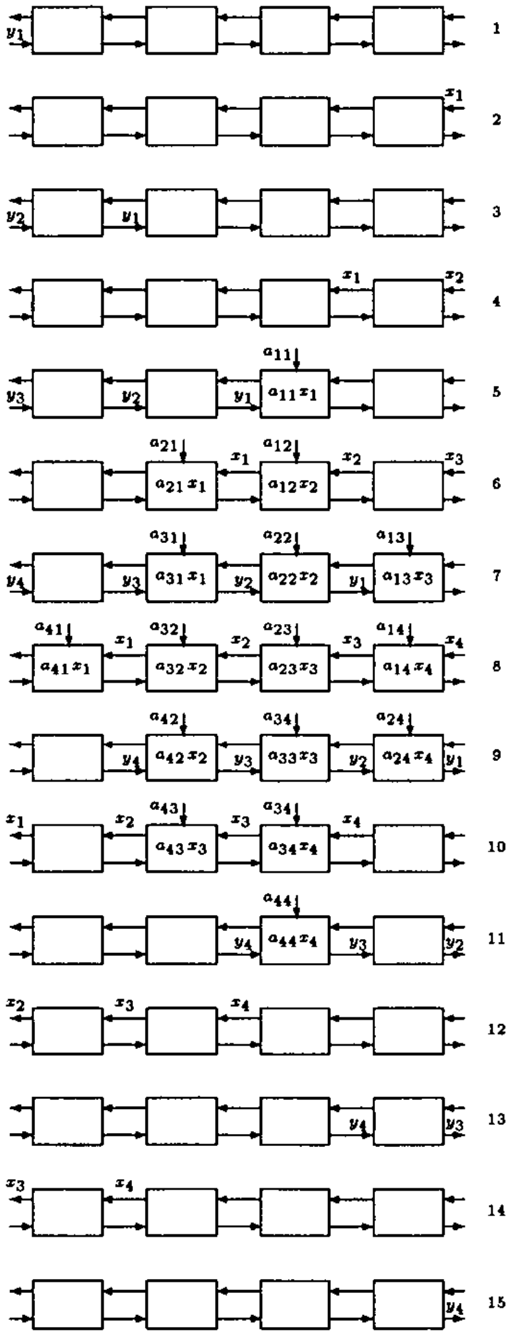


Figure 3.15: Matrix vector multiply design proposed by Suros and Montagne.

3.8 Application of systolic designs to the iterative solvers

The most simplest of the iterative methods is known as the Jacobi iterative method. It is written in mathematical form as,

$$\underline{x}^{(k+1)} = D^{-1}(\underline{b} - M\underline{x}^{(k)}). \tag{3.8.1}$$

Equation (3.8.1) suggests that the matrix vector multiplication is the basic ingredient of the iterative methods. In the following section we describe the implementation of the various matrix vector multiplication designs, discussed earlier, to the Jacobi iterative method.

3.8.1 Systolic designs for the Jacobi iterative method

The systolic array to compute the Jacobi iterative method for the $(4 * 4)$ example discussed for Kung's design is shown in figure (3.16).

There is a special cell at the right hand side end of the linear systolic array. The special (boundary) cell gets the value $y_i = a_{ij}x_j^0$ for $i, j = 1, 2, \ldots, n$, with $i \neq j$ from the right hand side processor and computes the new approximation to the solution, i.e. $x_i^1 = (b_i - y_i)/a_{ii}$ for $i = 1, 2, \ldots, n$. This cell performs a division operation instead of the multiplication operation done by the IPS cell. Also the addition operation is changed to a subtraction operation, we call this cell a SDC (Subtract Divide Cell). The division operation can be avoided if the given matrix A is divided by its diagonals, i.e. $(D^{-1}A)$ is performed by the host computer prior to sending data to the array. Similarly the subtraction operation can be changed into a addition operation by multiplying the matrix A by -1, which can be done in parallel when $(D^{-1}A)$ is being performed, i.e. to perform $(-D^{-1}A)$. The remainder of the array is the well known matrix vector multiplication array previously discussed. The designs to perform the Jacobi iterative method proposed by Kung [45], Evans and Bekakos [5], Evans and

Architecture	Time	Area ((IPS) + SDC cells)	Utilisation
Kung	$4n$	$(2n-1)+1$	1/8
Evans and Bekakos	$3n+1$	$(n+1)+1$	$n^2/(3n+1)(n+1)$
Evans and Gusev	4n+2	$(n)+1$	$n^2/(4n+2)(n+1)$
Suros and Montagne	$4n+1$	$(n)+1$	$n^2/(4n+1)(n+1)$

Table 3.2: Time, area and utilisation of the Jacobi iterative method.

Gusev [17] and Suros and Montagne [75] are shown in figure (3.16), (3.17), (3.17) and (3.19) respectively. The following table (3.2) shows the area and time requirements of the linear systolic array implementing the Jacobi iterative method on various matrix vector multiply designs. Table (3.2) presents the area, time and utilisation of the Jacobi iterative method on different linear systolic arrays. It is assumed for simplicity that the IPS and SDC cells take the same amount of time to perform their respective operations.

The utilisation of the arrays implementing the Jacobi iterative method is decreased. This is due to the diagonal elements of the matrix, which are removed from the matrix vector multiply portion of the array. This means that the portion performing $a_{ii} * x_i$ for $i = 1, 2, \ldots, n$ is replaced by the dummy computation $0 * x_i$ for $i = 1, 2, \ldots, n$ due to the algorithm. This can be seen very prominently in figures (3.16) and figure (3.17) implementing the Jacobi iterative method on Kung's and the Evans and Bekakos array. The results come out of order in the Evans and Bekakos design as explained in section (3.7.2).

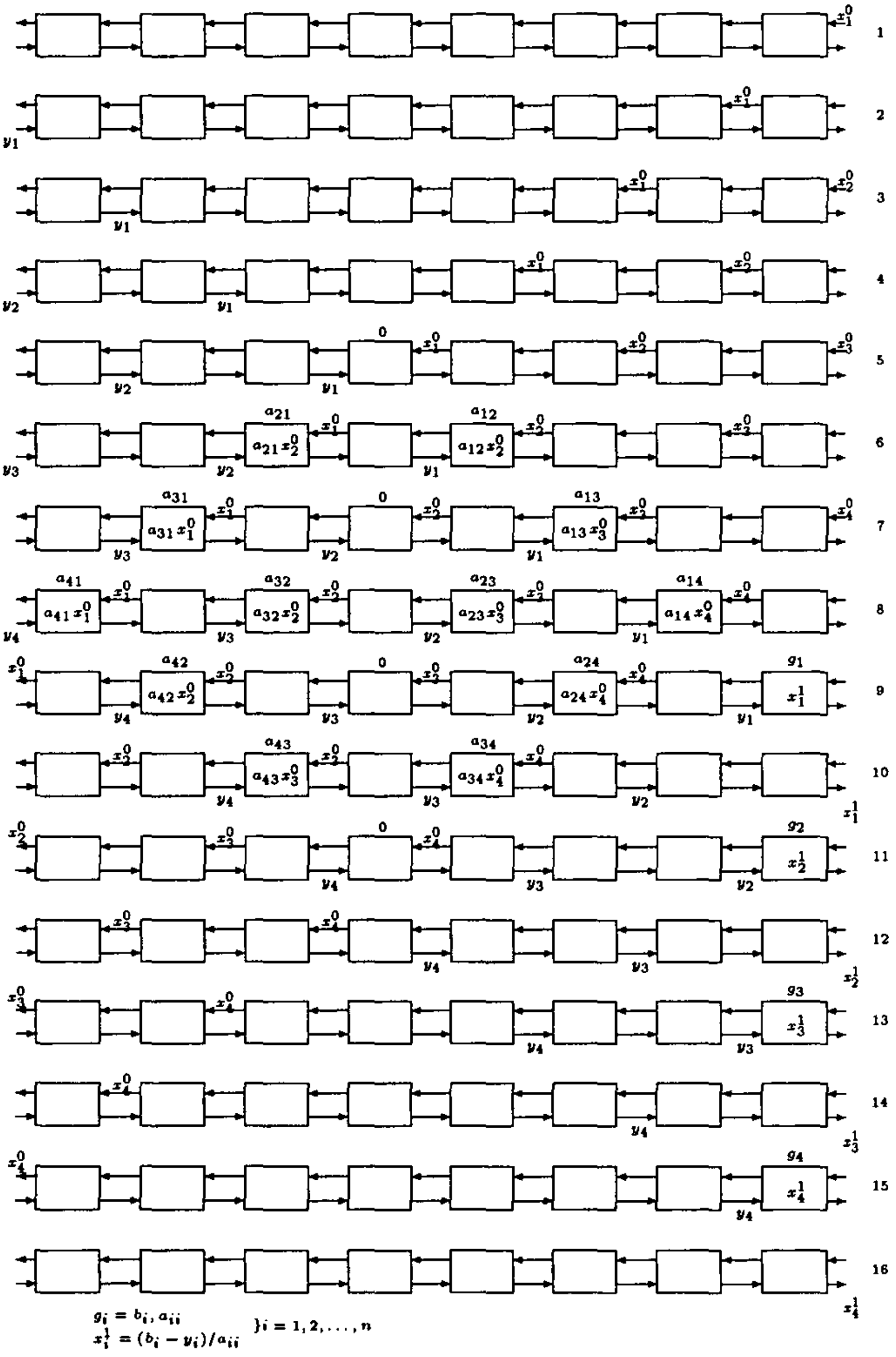
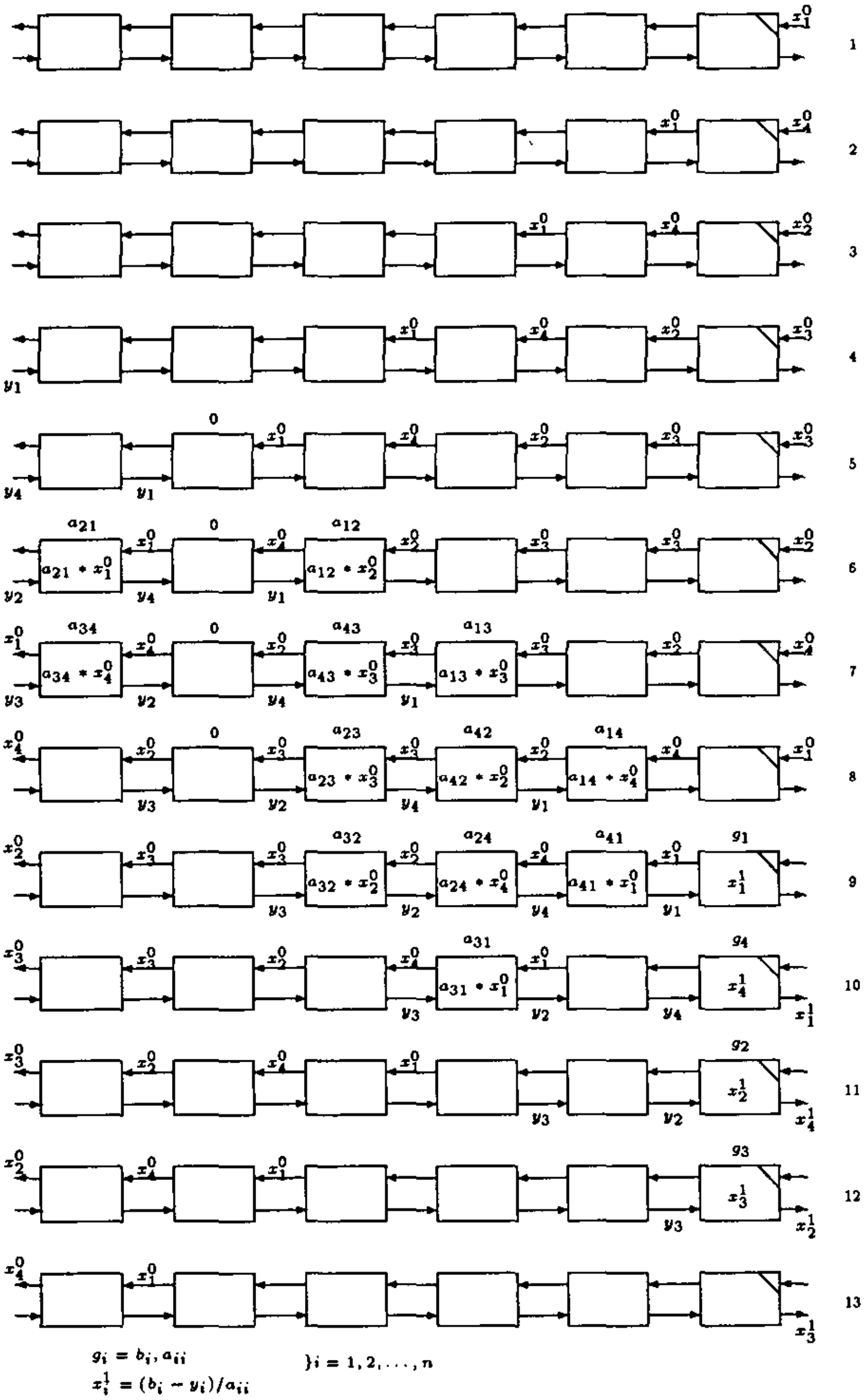
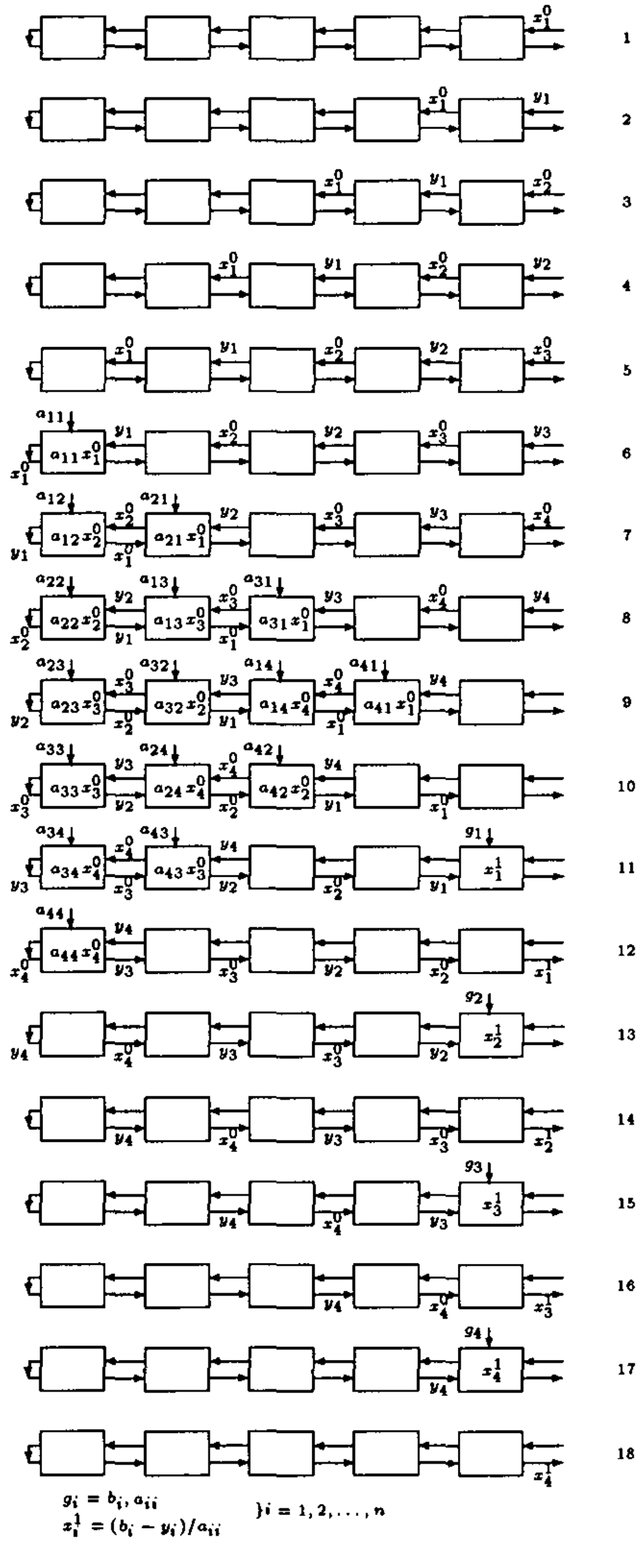


Figure 3.16: The Jacobi iterative method on Kung's array.



x_1^0 1
x_1^0 x_4^0 2
x_1^0 x_4^0 x_2^0 3
x_1^0 x_4^0 x_2^0 x_3^0 y_1 4
0 x_1^0 x_4^0 x_2^0 x_3^0 x_3^0 y_4 y_1 5
a_{21} x_1^0 0 x_4^0 a_{12} x_2^0 x_3^0 x_3^0 x_2^0 $a_{21} * x_1^0$ $a_{12} * x_2^0$ y_2 y_4 y_1 6
x_1^0 a_{34} x_4^0 0 x_2^0 a_{43} x_3^0 a_{13} x_3^0 x_2^0 x_4^0 $a_{34} * x_4^0$ $a_{43} * x_3^0$ $a_{13} * x_3^0$ y_3 y_2 y_4 y_1 7
x_4^0 x_2^0 0 x_3^0 a_{23} x_3^0 a_{42} x_2^0 a_{14} x_4^0 x_1^0 $a_{23} * x_3^0$ $a_{42} * x_2^0$ $a_{14} * x_4^0$ y_3 y_2 y_4 y_1 8
x_2^0 x_3^0 x_3^0 a_{32} x_2^0 a_{24} x_4^0 a_{41} x_1^0 g_1 $a_{32} * x_2^0$ $a_{24} * x_4^0$ $a_{41} * x_1^0$ x_1^1 y_3 y_2 y_4 y_1 9
x_3^0 x_3^0 x_2^0 x_4^0 a_{31} x_1^0 g_4 $a_{31} * x_1^0$ x_4^1 y_3 y_2 y_4 x_1^1 10
x_3^0 x_2^0 x_4^0 x_1^0 g_2 x_2^1 y_3 y_2 x_4^1 11
x_2^0 x_4^0 x_1^0 g_3 x_3^1 y_3 x_2^1 12
x_4^0 x_1^0 13 x_3^1
$g_i = b_i, a_{ii}$ $\}i = 1, 2, \ldots, n$
$x_i^1 = (b_i - y_i)/a_{ii}$

Figure 3.17: The Jacobi iterative method on Evans and Bekakos array.



$$g_i = b_i, a_{ii} \qquad \Big\} i = 1, 2, \ldots, n$$
$$x_i^1 = (b_i - y_i)/a_{ii}$$

Figure 3.18: The Jacobi iterative method on Evans and Gusev array.

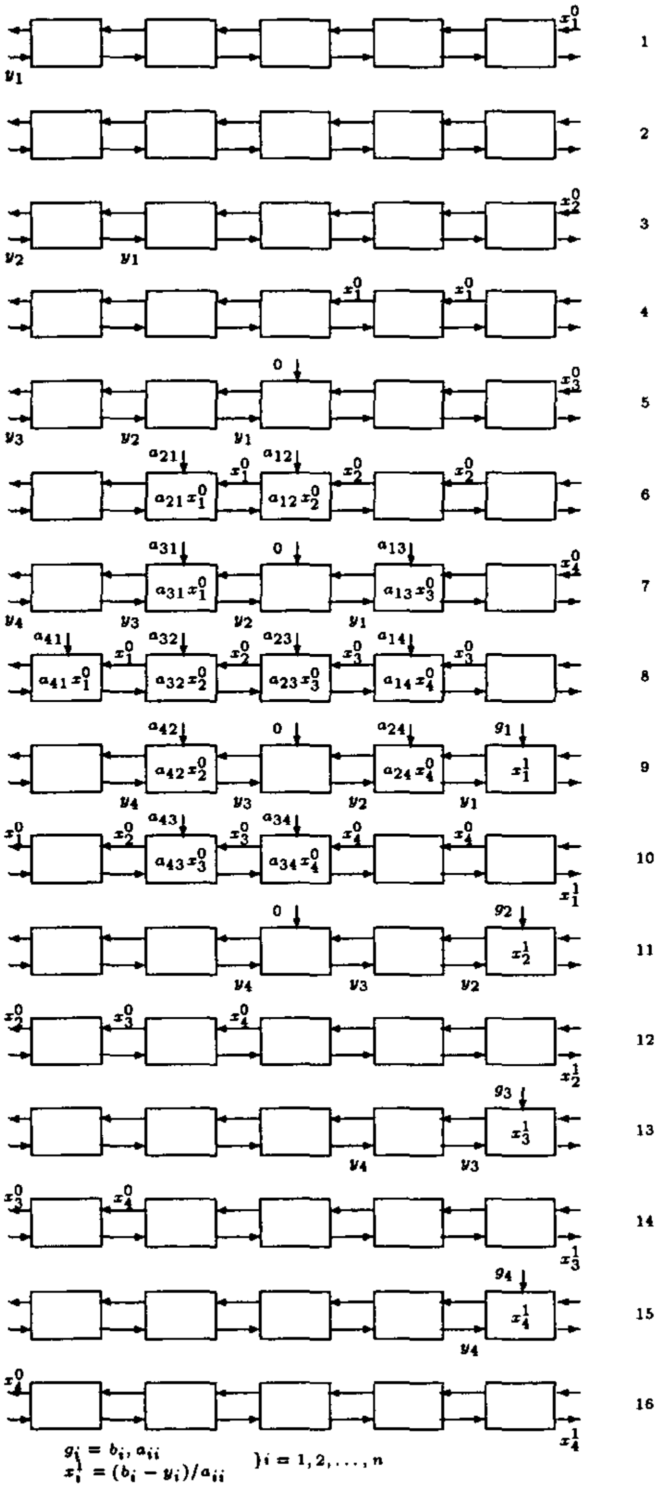

x_1^0
y_1
1
2
x_2^0
y_2 y_1
3
x_1^0 x_1^0
4
0
x_3^0
y_3 y_2 y_1
5
a_{21} x_1^0 a_{12} x_2^0 x_2^0
$a_{21}x_1^0$ $a_{12}x_2^0$
6
a_{31} 0 a_{13}
x_4^0
$a_{31}x_1^0$ $a_{13}x_3^0$
y_4 y_3 y_2 y_1
7
a_{41} x_1^0 a_{32} x_2^0 a_{23} x_3^0 a_{14} x_3^0
$a_{41}x_1^0$ $a_{32}x_2^0$ $a_{23}x_3^0$ $a_{14}x_4^0$
8
a_{42} 0 a_{24} g_1
$a_{42}x_2^0$ $a_{24}x_4^0$ x_1^1
y_4 y_3 y_2 y_1
9
x_1^0 x_2^0 a_{43} x_3^0 a_{34} x_4^0 x_4^0
$a_{43}x_3^0$ $a_{34}x_4^0$
x_1^1
10
0 g_2
x_2^1
y_4 y_3 y_2
11
x_2^0 x_3^0 x_4^0
x_2^1
12
g_3
x_3^1
y_4 y_3
13
x_3^0 x_4^0
x_3^1
14
g_4
x_4^1
y_4
15
x_4^0
x_4^1
16
$g_i = b_i, a_{ii}$
$x_i^1 = (b_i - y_i)/a_{ii}$
$\}i = 1, 2, \ldots, n$

Figure 3.19: The Jacobi iterative method on Suros and Montagne array.

3.8.2 Systolic designs for the Gauss-Seidel iterative method

The Jacobi iterative method is greatly improved if the method uses the most recent values of the variables x_{i-1} to evaluate the new approximation of x_i for the value $1 \leq i \leq n$. The method thus obtained is known as the Gauss-Seidel iterative method and is written as,

$$\underline{x}^{(k+1)} = D^{-1}(\underline{b} - L\underline{x}^{(k+1)} - U\underline{x}^{(k)}). \qquad (3.8.2)$$

This method suggests that an order is imposed on calculating the approximations, i.e. ordering is necessary (x_{i-1} is to be evaluated before x_i). This means that the idea of Evans and Bekakos does not allow the Gauss-Seidel iterative method to be implemented on the systolic array. The method of fitted diagonals proposed by Suros and Montagne can not be used to implement the Gauss-Seidel iterative method, due to synchronisation of data for the lower part of the algorithm. However, the design proposed by Kung, Evans and Gusev can be implemented to perform the Gauss-Seidel iterative method. Figures 3.20 and 3.21 show the Gauss-Seidel iterative method implemented on the Kung's and, Evans and Gusev array.

The area requirement of the design using the Evans and Gusev [17] matrix vector multiplication is less compared to the Kung's design. The time required to compute each iteration step is the same. The IPS cell used by Evans and Gusev (SIPS) has a complex control and switching logic, which enables the two computational streams to be accomplished.

One fact to note here is that if the matrix is dense then the processor utilisation is quite low. So the systolic algorithms are not well suited for these types of matrices. If the matrix is banded but dense, i.e. the number of non-zeros on the sub and super diagonals of the matrix is much less compared to the zeros then the systolic array has best utilisation. Table (3.3) gives area, time and utilisation of the linear systolic array implementing the Gauss-Seidel iterative

Architecture	Time	Area ((IPS)+Boundary cell)	Utilisation
Kung	$4n-3$	$(2n-2)+1$	$n^2/(4n-3)(2n-1)$
Evans and Bekakos		N.A.	
Evans and Gusev	$4n-3$	$(n)+1$	$n^2/(4n+3)(n+1)$
Suros and Montagne		N.A.	

Table 3.3: Time, area and utilisation of Guass-Seidel iterative method.

method.

The matrix vector multiplication design proposed by Evans and Bekakos [5], Suros and Montagne [75] and, Evans and Gusev [17] are well suited for dense matrices. The problems originating from the Engineering and scientific fields, when discretised by using the finite difference approximation result into sparse matrices, i.e. the one dimensional problem produces a tridiagonal matrix while the higher dimensional problems produce large sparse matrices e.g. a 2 dimensional discretised problem looks like,

$$\begin{bmatrix} 4 & -1 & 0 & -1 & & & & & \\ -1 & 4 & -1 & 0 & -1 & & & & \mathbf{0} \\ 0 & -1 & 4 & 0 & 0 & -1 & & & \\ -1 & 0 & 0 & 4 & -1 & 0 & -1 & & \\ & -1 & 0 & -1 & 4 & -1 & 0 & -1 & \\ & & -1 & 0 & -1 & 4 & 0 & 0 & -1 \\ & & & -1 & 0 & 0 & 4 & -1 & 0 \\ \mathbf{0} & & & & -1 & 0 & -1 & 4 & -1 \\ & & & & & -1 & 0 & -1 & 4 \end{bmatrix} * \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \\ b_8 \\ b_9 \end{bmatrix} \qquad (3.8.3)$$

as the order of the system is increased the gap between the sub diagonals is increased. This suggests that the concept of folding does not give any advantage as the number of non-zero diagonals is small and hence the system under consideration becomes more complex and expensive, as compared to Kung's original design. Later in this thesis we propose a new design to implement such problems on a linear systolic array.

Note that in the case of the Jacobi and Gauss-Seidel iterative methods the diagonals are pumped to the boundary cells. However their original position in the matrix vector multiply is changed with dummy data, i.e. there is some extra overhead for the host computer, in all the implementations discussed.

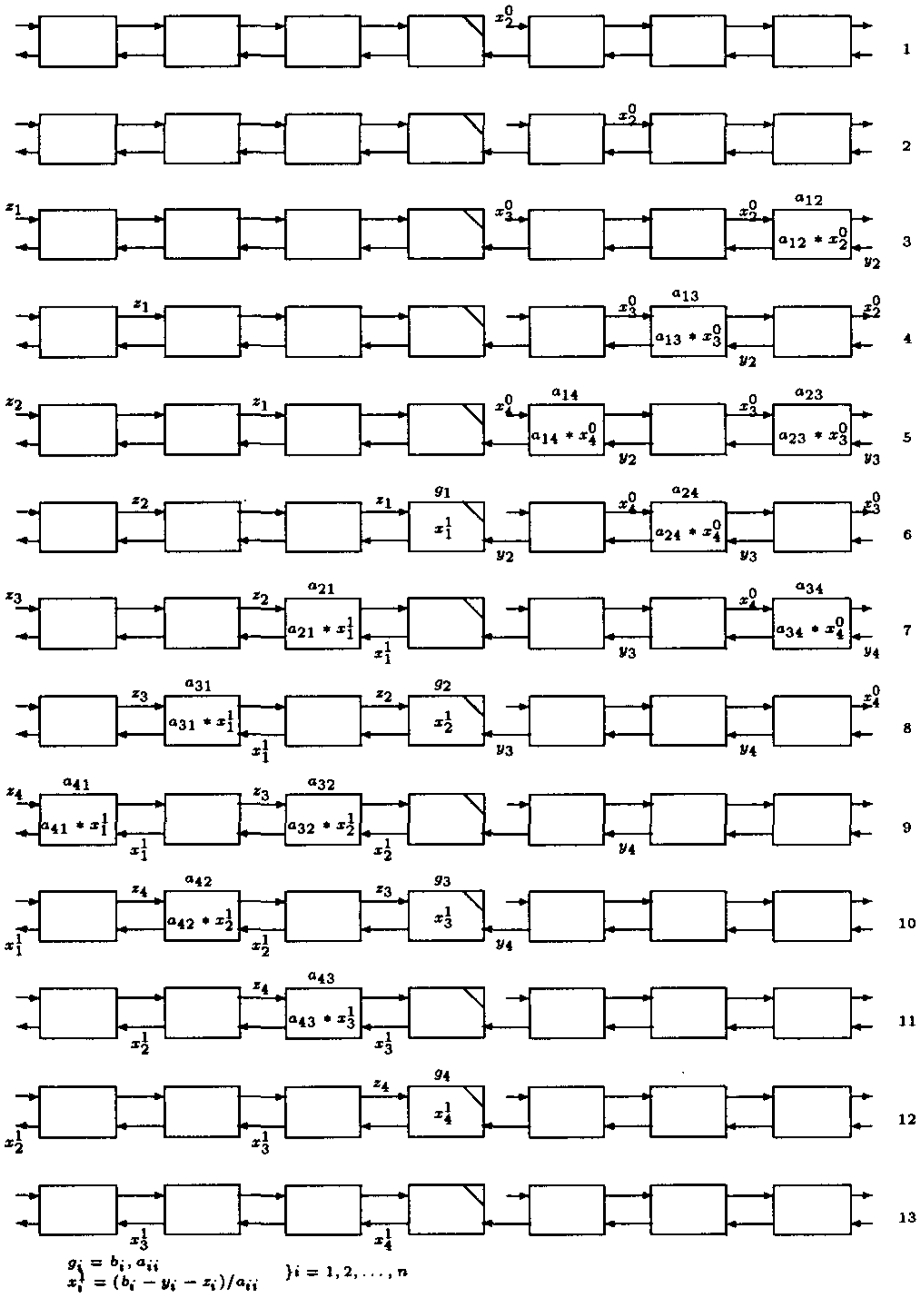


Figure 3.20: The Gauss-Seidel iterative method on Kung's array.

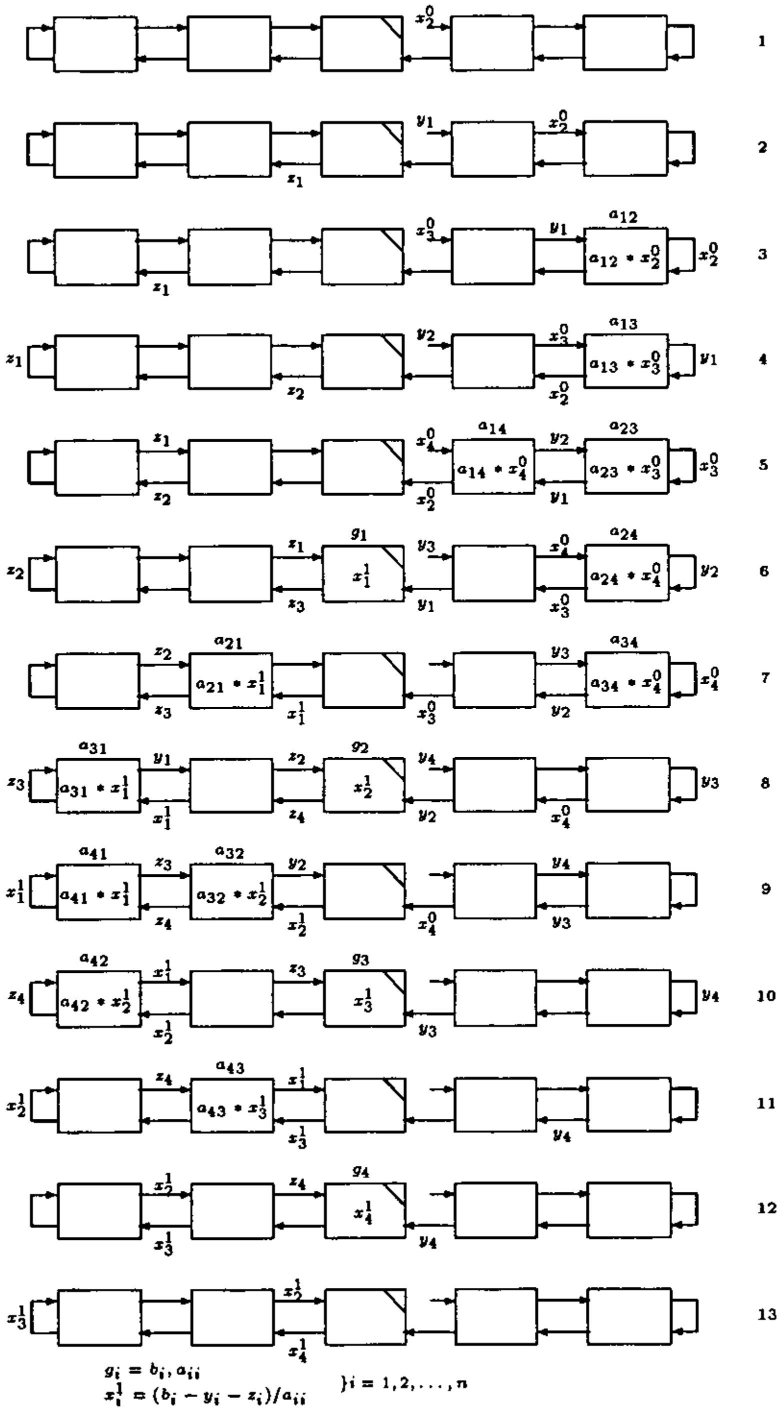


Figure 3.21: The Gauss-Seidel iterative method Evans and Gusev array.

Chapter 4

Systolic Arrays for the Symmetric S.O.R. and Related Methods

This chapter presents the systolic designs for the Jacobi overrelaxation (J.O.R.), Successive Overrelaxation (S.O.R.), Modified Successive Overrelaxation (M.S.O.R.) and Symmetric Successive Overrelaxation (S.S.O.R.) methods for the iterative solution of linear systems such that the computational time is decreased. The Conrad-Wallach technique [11] is incorporated in the S.S.O.R. systolic design so that the overall work is comparable to that of the S.O.R. iterative method. The systolic array for matrix vector multiplication (mvm) is used to compute the iterations.

4.1 Introduction

Several iterative solutions to the large sparse linear systems obtained from the finite difference/element discretisation of boundary value problems are well known like the Jacobi, Gauss-Seidel, J.O.R. and S.O.R. methods (see Young [82], Hageman and Young [34]). Further several acceleration and overrelaxation strategies have been developed and incorporated into the methods (in order to decrease the computation time). For these computational methods several systolic designs have also been developed (see Evans and Margaritis [20], Berzins, Buckley and Dew [7], Margaritis [51], Casasent [10], Quinton, Jannault and Gachet [63]).

The systolic array processor was introduced by H. T. Kung [45] and is a linearly connected array of simple processors. The array is highly pipelined with few memory references. The data is fed into the array and each processor utilises the data as it passes through the array and at the end of the computation the result is sent to the memory. The systolic array is best suited for computation bound problems. Thus it is well suited for the iterative solvers of large linear banded systems.

The systolic solution for the Jacobi, Gauss-Seidel and S.O.R. iterative methods have been presented in Berzins, Buckley and Dew [7] and Margaritis [51]. The solutions consist of pipeline designs both inside each iteration as well as in successive iterations. In Quinton et al. [63] and, Evans and Margaritis [20] a reusable systolic design for matrix-matrix-multiplication (mmm) and matrix vector-multiplication (mvm) are presented. Using these concepts an improved systolic design for the S.S.O.R. iterative method is presented.

4.2 Basic iterative methods

The equation,

$$A\underline{x} = \underline{b}, \tag{4.2.1}$$

represents a system of n linear equations with n unknowns. Here A is the $(n \times n)$ coefficient matrix, $\underline{x}$ are the n unknowns and $\underline{b}$ represents the n known constants. If matrix A is split such that,

$$A = D + J = D + L + U,$$

where D is the diagonal matrix with non-zero elements, J has zeros on its diagonals and L and U are strictly lower and upper triangular matrices, then the Jacobi iterative method is defined as

$$\underline{x}^{(k+1)} = D^{-1}(\underline{b} - J\underline{x}^{(k)}), \tag{4.2.2}$$

and is independent of the ordering of the equations in (4.2.2). The matrix-$D^{-1}J$ is called the Jacobi iteration matrix. It can be seen from equation (4.2.2) that the operations involved in each iteration are a matrix vector multiplication, one addition and a scalar multiplication. Equation (4.2.2) representing the Jacobi iterative method can also be written in the iterative form as follows,

$$\underline{x}^{(k+1)} = D^{-1}(\underline{b} - L\underline{x}^{(k)} - U\underline{x}^{(k)}). \tag{4.2.3}$$

Figure (4.1) represents a simple systolic implementation of the Jacobi iterative method on a linear array.

The boundary processor on the right hand side performs the addition and the scalar multiplication whereas the remaining processors perform the matrix vector multiplication. The Jacobi iterative method can be improved greatly by the Gauss-Seidel and S.O.R. iterative methods, where the most recent iterate is

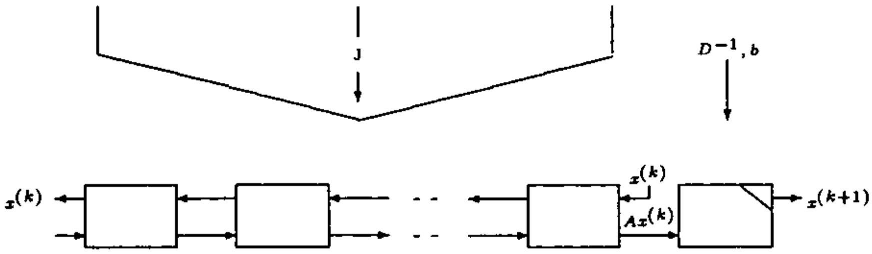


Figure 4.1: Systolic design for the Jacobi iterative method.

used in the computation. This is the basic difference between the Gauss-Seidel and the Jacobi iterative methods, and explains the ordering requirements for the Gauss-Seidel iterative method. The Gauss-Seidel method does not require the simultaneous storage of the new and old approximations in the course of computation.

4.2.1 The Jacobi Overrelaxation method

The equation (4.2.2) can be written as,

$$\underline{x}^{(k+1)} = \underline{x}^{(k)} + [D^{-1}(\underline{b} - J\underline{x}^{(k)}) - \underline{x}^{(k)}] \tag{4.2.4}$$

where the term $[D^{-1}(\underline{b} - J\underline{x}^{(k)}) - \underline{x}^{(k)}]$ is the correction factor. If we multiply this correction factor by a constant ω, then we get,

$$\underline{x}^{(k+1)} = \underline{x}^{(k)} + \omega[D^{-1}(\underline{b} - J\underline{x}^{(k)}) - \underline{x}^{(k)}], \tag{4.2.5}$$

which can be re-written as,

$$\underline{x}^{(k+1)} = \omega[D^{-1}(\underline{b} - J\underline{x}^{(k)})] + (1 - \omega)\underline{x}^{(k)}. \tag{4.2.6}$$

The constant ω is called the acceleration factor. When $\omega > 1$, then equation (4.2.6) is called the Jacobi overrelaxation (J.O.R.) method. If $\omega < 1$ then the method is known as the Jacobi underrelaxation method. Expanding equation (4.2.6) we get,

$$\underline{x}^{(k+1)} = \omega D^{-1}\underline{b} - \omega D^{-1}J\underline{x}^{(k)} + (1-\omega)\underline{x}^{(k)},$$

$$\underline{x}^{(k+1)} = \omega D^{-1}\underline{b} + \left[(1-\omega)I - \omega D^{-1}J\right]\underline{x}^{(k)},$$

$$\underline{x}^{(k+1)} = \omega D^{-1}\underline{b} + \left[(1-\omega)I - \omega D^{-1}(L+U)\right]\underline{x}^{(k)},$$

where $[(1-\omega)I - \omega D^{-1}(L+U)]$ is the J.O.R. iteration matrix.

4.2.2 The Successive Overrelaxation method

For the standard Gauss-Seidel iterative method the new vector approximation is written as,

$$\underline{x}^{(k+1)} = D^{-1}(\underline{b} - L\underline{x}^{(k+1)} - U\underline{x}^{(k)}). \tag{4.2.7}$$

The equation (4.2.7) can be written as

$$(I + D^{-1}L)\underline{x}^{(k+1)} = D^{-1}(\underline{b} - U\underline{x}^{(k)}), \tag{4.2.8}$$

$$\underline{x}^{(k+1)} = (I + D^{-1}L)^{-1}[D^{-1}(\underline{b} - U\underline{x}^{(k)})], \tag{4.2.9}$$

where $-(I + D^{-1}L)^{-1}D^{-1}U$ is the Gauss-Seidel iteration matrix. The correction vector in the Gauss-Seidel iterative method is given by $[D^{-1}(\underline{b} - L\underline{x}^{(k+1)} - U\underline{x}^{(k)}) - \underline{x}^{(k)}]$. Now the equation (4.2.7) can be written as,

$$\underline{x}^{(k+1)} = \underline{x}^{(k)} + [D^{-1}(\underline{b} - L\underline{x}^{(k+1)} - U\underline{x}^{(k)}) - \underline{x}^{(k)}]. \tag{4.2.10}$$

If an overrelaxation factor ω is applied to the equation (4.2.10) then we get,

$$\underline{x}^{(k+1)} = \underline{x}^{(k)} + \omega[D^{-1}(\underline{b} - L\underline{x}^{(k+1)} - U\underline{x}^{(k)}) - \underline{x}^{(k)}] \qquad (4.2.11)$$

$$\underline{x}^{(k+1)} = \omega D^{-1}[(\underline{b} - U\underline{x}^{(k)})] - \omega D^{-1}L\underline{x}^{(k+1)} + (1-\omega)\underline{x}^{(k)},$$

$$(I + \omega D^{-1}L)\underline{x}^{(k+1)} = \omega D^{-1}(\underline{b} - U\underline{x}^{(k)}) + (1-\omega)\underline{x}^{(k)},$$

$$(I + \omega D^{-1}L)\underline{x}^{(k+1)} = \omega D^{-1}\underline{b} + [-\omega D^{-1}U + (1-\omega)I]\underline{x}^{(k)},$$

$$\underline{x}^{(k+1)} = (I + \omega D^{-1}L)^{-1}\left\{\omega D^{-1}\underline{b} + \left[-\omega D^{-1}U + (1-\omega)I\right]\underline{x}^{(k)}\right\}. (4.2.12)$$

Equation (4.2.12) represents the successive overrelaxation (S.O.R.) iterative method. The matrix $S = (I + \omega D^{-1}L)^{-1}[-\omega D^{-1}U + (1-\omega)I]$ is called the S.O.R. iteration matrix.

The overrelaxation strategy is used to achieve a faster rate of convergence for the iterative methods. The standard S.O.R. iterative method is written as,

$$\underline{x}^{(k+1)} = D^{-1}(\omega(\underline{b} - L\underline{x}^{(k+1)} - U\underline{x}^{(k)})) + (1-\omega)\underline{x}^{(k)}. \qquad (4.2.13)$$

Young, [82], has shown that this method for solving equations such as ($A\underline{x} = \underline{b}$) converges for $0 < \omega < 2$ if A is symmetric positive definite matrix. Also the optimum value "ω_o" of ω for maximum convergence is given by,

$$\omega_o = \frac{2}{1 + \sqrt{(1-\mu^2)}},$$

where μ is the spectral radius of the Jacobi iteration matrix.

4.2.3 The Modified Successive Overrelaxation method

The equation (4.2.1), represents the system of n linear equations with n unknowns. Here if A represents a block matrix of the form,

$$A = \begin{bmatrix} D_1 & A_1 \\ A_2 & D_2 \end{bmatrix},$$

where D_1, D_2, A_1, A_2, are block sub matrices, and D_1 and D_2 contain only the diagonal elements. If we consider a red/black ordering of the equations then A_1 contains all the "black" points of matrix A and A_2 contains all the "red" points of A. As before $\underline{x}$ are the n unknowns and $\underline{b}$ represents the n known constants. $\underline{x}$ and $\underline{b}$ are partitioned to match the red/black partitioning of matrix A. The matrix A is known as a 2-cyclic matrix. The matrix A can also be scaled in the form,

$$A = \begin{bmatrix} I_1 & D_1^{-1}A_1 \\ D_2^{-1}A_2 & I_2 \end{bmatrix} = \begin{bmatrix} I_1 & F_1 \\ F_2 & I_2 \end{bmatrix}, \tag{4.2.14}$$

The sub-matrices F_1 and F_2 have the following properties.

1. If n is even the F_1 and F_2 are $\left(\frac{n}{2}, \frac{n}{2}\right)$ matrices. If n is odd then F_1 and F_2 are $\left(\frac{(n+1)}{2}, \frac{(n-1)}{2}\right)$ and $\left(\frac{(n-1)}{2}, \frac{(n+1)}{2}\right)$ matrices respectively.
2. In case of even n, F_1 is lower triangular bidiagonal and F_2 is upper triangular bidiagonal matrix. For odd n, F_1 and F_2 can be transformed to lower and upper triangular bidiagonal matrices, by adding a dummy column and row respectively.

Similarly for even n, $\underline{x}_1, \underline{x}_2, \underline{g}_1, \underline{g}_2$ are $\left(\frac{n}{2}, 1\right)$ matrices and for n odd $\underline{x}_1, \underline{g}_1$ are $\left(\frac{(n+1)}{2}, 1\right)$ and $\underline{x}_2$, and $\underline{g}_2$ are $\left(\frac{(n-1)}{2}, 1\right)$ matrices.

The b_i for $i = 1, 2, \ldots, n$ is also modified accordingly and represented as $g_i = b_i/a_{ii}$ for $i = 1, 2, \ldots, n$.

The S.O.R. iterative method can be written for A in the form (4.2.14) as (see Varga [78]),

$$\underline{x}_1^{(k+1)} = \omega(F_1\underline{x}_2^{(k)} + \underline{g}_1) + (1-\omega)\underline{x}_1^{(k)}, \tag{4.2.15}$$

$$\underline{x}_2^{(k+1)} = \omega(F_2\underline{x}_1^{(k+1)} + \underline{g}_2) + (1-\omega)\underline{x}_2^{(k)}. \tag{4.2.16}$$

If two overrelaxation factors "ω" and "ω'" are applied to the S.O.R. iterative method, such that one overrelaxation factor is used for the red ordered equations and the other for the black ordered equations the iterative method obtained is known as modified S.O.R. (see Young [82], Martins [54], Evans, Haider and Martin [19]). The following equations represent the M.S.O.R. iterative method,

$$\underline{x}_1^{(k+1)} = \omega(F_1\underline{x}_2^{(k)} + \underline{g}_1) + (1-\omega)\underline{x}_1^{(k)}, \tag{4.2.17}$$

$$\underline{x}_2^{(k+1)} = \omega'(F_2\underline{x}_1^{(k+1)} + \underline{g}_2) + (1-\omega')\underline{x}_2^{(k)}. \tag{4.2.18}$$

If $\omega = \omega'$ then the M.S.O.R. iterative method reduces to the S.O.R. iterative method.

4.2.4 The Symmetric Successive Overrelaxation method

A variant to the S.O.R. iterative method called the Symmetric Successive Overrelaxation (S.S.O.R.) was proposed by Sheldon [67]. In this iterative method the S.O.R. iterative method is applied first by using the forward ordering of the equations and then to the backward ordering of the equations. So the 2^{nd} half iteration of the S.S.O.R. iterative method will be given by,

$$\underline{x}^{(k+2)} = D^{-1}(\omega(\underline{b} - L\underline{x}^{(k+1)} - U\underline{x}^{(k+2)})) + (1-\omega)\underline{x}^{(k+1)} \tag{4.2.19}$$

$$\underline{x}^{(k+2)} = (I+\omega D^{-1}U)^{-1}\left\{\omega D^{-1}\underline{b} + \left[-\omega D^{-1}L + (1-\omega)I\right]\underline{x}^{(k+1)}\right\}. \tag{4.2.20}$$

The iteration matrix for the first half iteration is as that of the S.O.R. iterative method and the iteration matrix for the second half is given by $G = (I + \omega D^{-1}U)^{-1}[-\omega D^{-1}L + (1-\omega)I]$. By applying the iterations in this order it can be shown that the iteration matrix K is symmetric and positive definite for $0 < \omega < 2$. Consequently the eigenvalues of K are all real.

Since $\underline{x}^{(k+1)}$ and $\underline{x}^{(k+2)}$ are the 1^{st} and 2^{nd} half of the S.S.O.R. iterative method, they can be written as $\underline{x}^{(k+\frac{1}{2})}$ and $\underline{x}^{(k+1)}$ notation. So equation (4.2.12) and equation (4.2.19) can be rewritten as,

$$\underline{x}^{(k+\frac{1}{2})} = S\underline{x}^{(k)} + (I+\omega D^{-1}L)^{-1}(\omega D^{-1}\underline{b}) \tag{4.2.21}$$

$$\underline{x}^{(k+1)} = G\underline{x}^{(k+\frac{1}{2})} + (I+\omega D^{-1}U)^{-1}(\omega D^{-1}\underline{b}). \tag{4.2.22}$$

Eliminating $\underline{x}^{(k+\frac{1}{2})}$ in equation (4.2.22) by using equation (4.2.21) we get,

$$\underline{x}^{(k+1)} = K\underline{x}^{(k)} + \underline{g} \tag{4.2.23}$$

where $\underline{g}$ is evaluated as follows,

$$\underline{g} = G\left[(I+\omega D^{-1}L)^{-1}(\omega D^{-1}\underline{b})\right] + (I+\omega D^{-1}U)^{-1}(\omega D^{-1}\underline{b})$$

$$\underline{g} = \left[G(I+\omega D^{-1}L)^{-1} + (I+\omega D^{-1}U)^{-1}\right](\omega D^{-1}\underline{b})$$

substituting the value of G we get,

$$\begin{aligned}\underline{g} &= \left\{(I+\omega D^{-1}U)^{-1}\left[(1-\omega)I - \omega D^{-1}L\right]\right.\\ &\quad \left.(I+\omega D^{-1}L)^{-1} + (I+\omega D^{-1}U)^{-1}\right\}(\omega D^{-1}\underline{b})\end{aligned}$$

$$\underline{g} = \left\{\left[(1-\omega)I - \omega D^{-1}L\right](I+\omega D^{-1}L)^{-1} + I\right\}(I+\omega D^{-1}U)^{-1}(\omega D^{-1}\underline{b})$$

$$\begin{aligned}\underline{g} &= \left[(1-\omega)I - \omega D^{-1}L + I + \omega D^{-1}L\right]\\ &\quad (I+\omega D^{-1}L)^{-1}(I+\omega D^{-1}U)^{-1}(\omega D^{-1}\underline{b})\end{aligned}$$

$$\underline{g} = \omega(2-\omega)(I+\omega D^{-1}L)^{-1}(I+\omega D^{-1}U)^{-1}(D^{-1}\underline{b}) \tag{4.2.24}$$

$$K = GS$$

$$\begin{aligned}K &= (I+\omega D^{-1}U)^{-1}\left[-\omega D^{-1}L + (1-\omega)I\right]\\ &\quad (I+\omega D^{-1}L)^{-1}\left[-\omega D^{-1}U + (1-\omega)I\right]\end{aligned}$$

$$\begin{aligned}K &= (I+\omega D^{-1}U)^{-1}(I+\omega D^{-1}L)^{-1}\\ &\quad \left[(1-\omega)I - \omega D^{-1}L\right]\left[(1-\omega)I - \omega D^{-1}U\right].\end{aligned}$$

By observing equations (4.2.13) and (4.2.19) it can be seen that the $L\underline{x}^{(k+1)}$ factor is common, so there is no need for it to be calculated twice. Similarly for the factor $U\underline{x}^{(k+2)}$ which becomes $U\underline{x}$ for the next iteration (see equation (4.2.25)). This increases the storage requirement although the computational work load is now similar to the S.O.R. iterative method.

This strategy is known as the Conrad-Wallach technique (see Conrad and Wallach [11]). The area requirements for the S.S.O.R. iterative method with and without the Conrad-Wallach technique remain the same but the time requirements are different. The next iteration for the S.S.O.R. iterative method after (4.2.19) will be,

$$\underline{x}^{(k+3)} = D^{-1}(\omega(\underline{b} - L\underline{x}^{(k+3)} - U\underline{x}^{(k+2)})) + (1-\omega)\underline{x}^{(k+2)}. \tag{4.2.25}$$

So if successive iterations for the S.S.O.R. iterative method are written, it can be observed that $U\underline{x}^{(0)}$ is the only computation required to be done. The $L\underline{x}^{(k+1)}$ and $U\underline{x}^{(k+2)}$ terms for the remainder of the iterations are precomputed in the most recent previous iteration respectively.

4.2.5 The U.S.S.O.R. iterative method

If two overrelaxation parameters "ω" and "$\hat{\omega}$" are cyclically used alternatively in successive iterations of the S.S.O.R. iterative method, the method obtained is known as Unsymmetric Successive Overrelaxation (U.S.S.O.R.). The equations (4.2.13), (4.2.19) and (4.2.25) can be rewritten to represent the U.S.S.O.R. iterative method as follows:

$$\underline{x}^{(k+1)} = D^{-1}(\omega(\underline{b} - L\underline{x}^{(k+1)} - U\underline{x}^{(k)})) + (1-\omega)\underline{x}^{(k)}, \tag{4.2.26}$$

$$\underline{x}^{(k+2)} = D^{-1}(\hat{\omega}(\underline{b} - L\underline{x}^{(k+1)} - U\underline{x}^{(k+2)})) + (1-\hat{\omega})\underline{x}^{(k+1)}, \tag{4.2.27}$$

$$\underline{x}^{(k+3)} = D^{-1}(\omega(\underline{b} - L\underline{x}^{(k+3)} - U\underline{x}^{(k+2)})) + (1-\omega)\underline{x}^{(k+2)}. \tag{4.2.28}$$

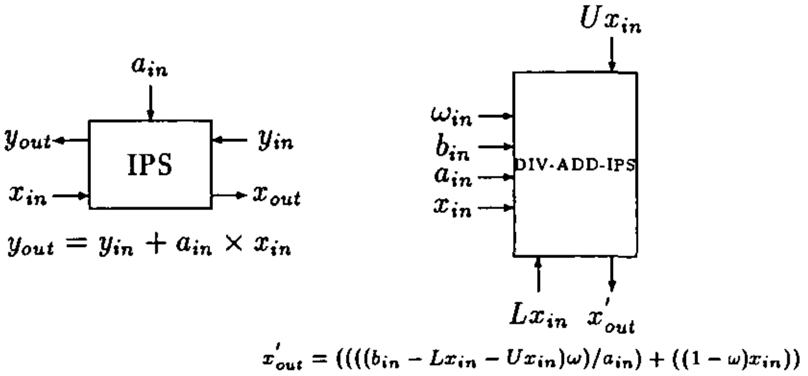

a_{in}
y_{out}
IPS
y_{in}
x_{in}
x_{out}
$y_{out} = y_{in} + a_{in} \times x_{in}$
Ux_{in}
ω_{in}
b_{in}
a_{in}
x_{in}
DIV-ADD-IPS
Lx_{in}
x'_{out}
$x'_{out} = ((((b_{in} - Lx_{in} - Ux_{in})\omega)/a_{in}) + ((1-\omega)x_{in}))$

Figure 4.2: IPS and DIV ADD IPS cells.

4.3 Systolic designs for the iterative solvers

The following sections present the systolic designs for the S.O.R., J.O.R., M.S.O.R. and S.S.O.R. iterative methods. The designs are simulated soft systolically using OCCAM as a hardware description language.

4.3.1 The S.O.R. systolic design

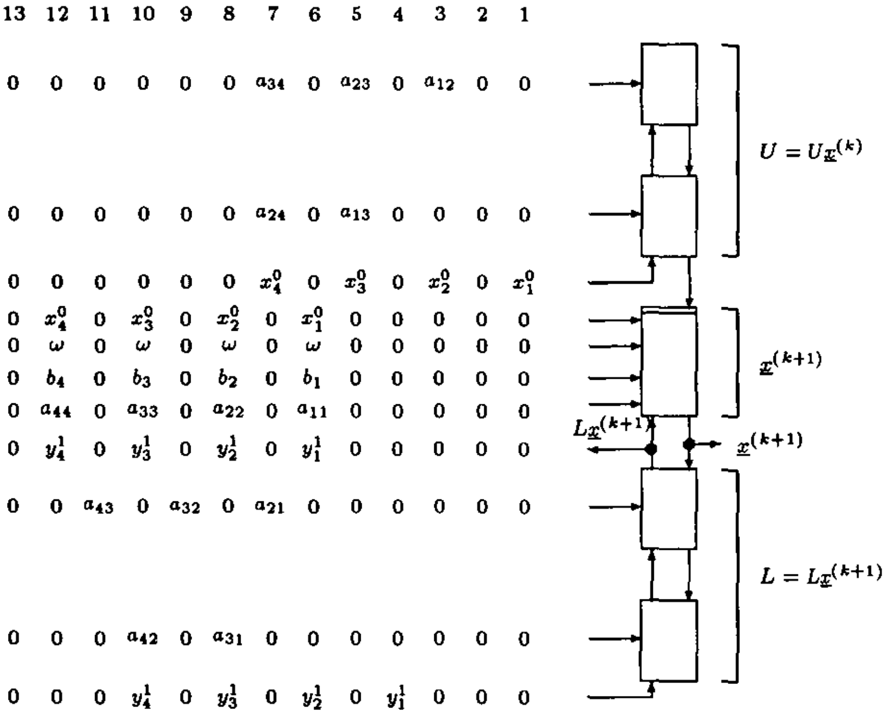
The S.O.R. systolic design is shown in figure (4.3) and figure (4.4) shows the snap shots of the 1^{st} iteration. The design consists of three parts. These parts compute the $L\underline{x}_i^{(k+1)}$, $U\underline{x}_i^{(k)}$ and $x_i^{(k+1)}$ (i.e. the new approximation to the solution vector) for $i = 0, 1 \ldots, n$. Here n is the size of the matrix and k is the number of iterations performed. The banded system with bandwidth $w = 5$ is chosen and a $(4 * 4)$ system is simulated and given by the following equation,

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & 0 \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ 0 & a_{42} & a_{43} & a_{44} \end{bmatrix} * \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix} \tag{4.3.1}$$

The S.O.R. linear systolic array works as follows, Let us compute the new approximation x_3^1. The Lx part consists of computation $(a_{31}x_1^1 + a_{32}x_2^1)$ and the Ux part consists of computation $(a_{34}x_4^0)$. The overall computation can be written as,

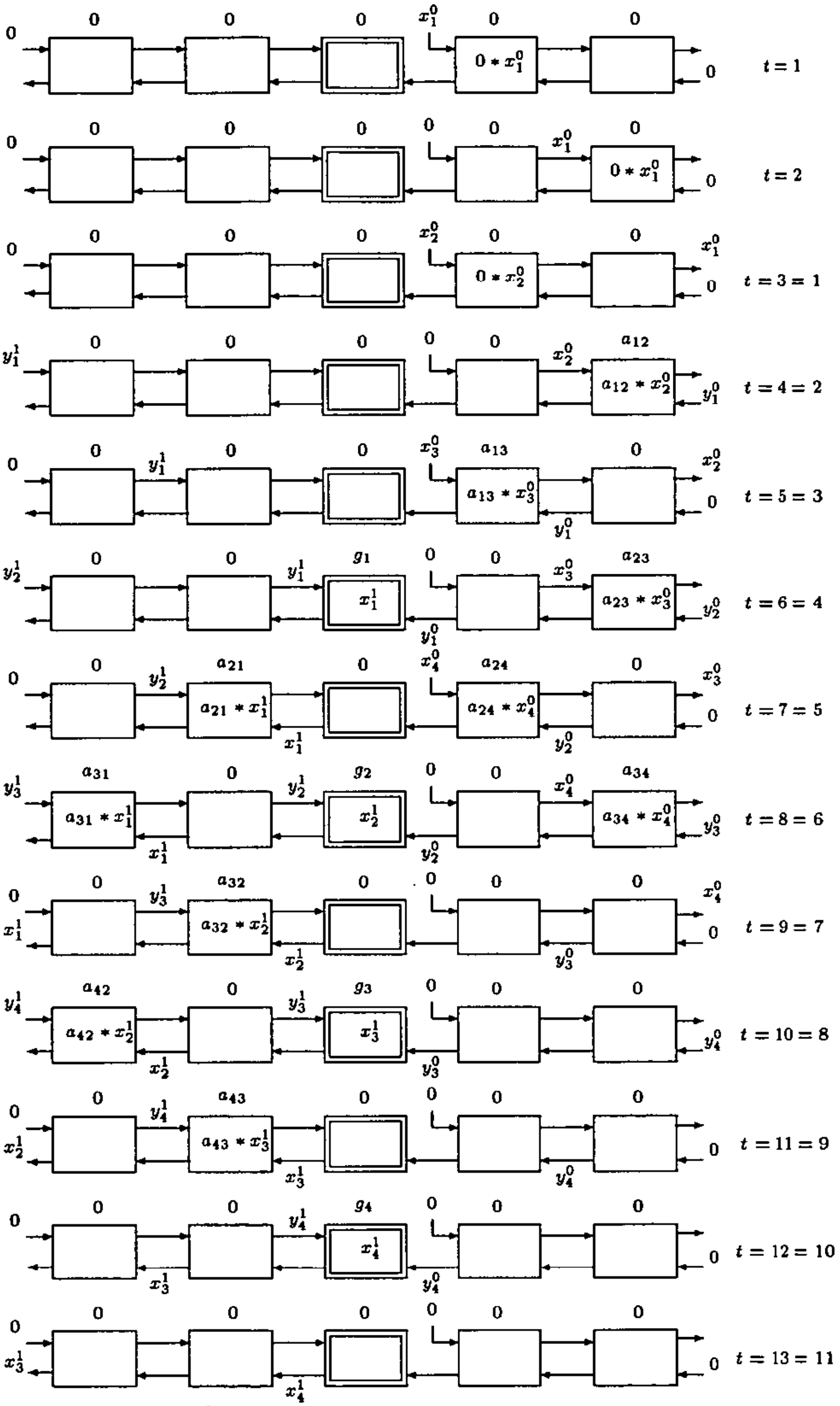
$$x_3^1 = (((b_3 - Lx - Ux)\omega)/a_{33}) + (1-\omega)x_3^0.$$

In figure (4.4) Lx and Ux computation is represented by y_3^1 and y_3^0 respectively. The processors in the array are marked from left to right (irrespective of the type) with processor 1 being the leftmost processor and processor 5 being the rightmost processor. At time step 8, x_1^1 moves into processor 1. The computation $(a_{31}x_1^1)$ is performed and $y_3^1 = a_{31}x_1^1$. x_4^0 enters the processor 5 and $(a_{34}x_4^0)$ is performed and $y_3^0 = a_{34}x_4^0$. The processor 3 computes x_2^1, this processor is known as the DIV ADD IPS processor and is shown in figure (4.2). At time step 9, y_3^1 moves to processor 2 and $(a_{32}x_2^1)$ is performed and $y_3^1 = a_{31}x_1^1 + a_{32}x_2^1$. The processor 4 passes the value y_3^0 unaltered. At time step 10, the y_3^0 and y_3^1 enter the DIV ADD IPS cell along with b_3, ω, a_{33} and x_3^0, and x_3^1 is computed.



13	12	11	10	9	8	7	6	5	4	3	2	1
0	0	0	0	0	0	a_{34}	0	a_{23}	0	a_{12}	0	0
0	0	0	0	0	0	a_{24}	0	a_{13}	0	0	0	0
0	0	0	0	0	0	x_4^0	0	x_3^0	0	x_2^0	0	x_1^0
0	x_4^0	0	x_3^0	0	x_2^0	0	x_1^0	0	0	0	0	0
0	ω	0	ω	0	ω	0	ω	0	0	0	0	0
0	b_4	0	b_3	0	b_2	0	b_1	0	0	0	0	0
0	a_{44}	0	a_{33}	0	a_{22}	0	a_{11}	0	0	0	0	0
0	y_4^1	0	y_3^1	0	y_2^1	0	y_1^1	0	0	0	0	0
0	0	a_{43}	0	a_{32}	0	a_{21}	0	0	0	0	0	0
0	0	0	a_{42}	0	a_{31}	0	0	0	0	0	0	0
0	0	0	y_4^1	0	y_3^1	0	y_2^1	0	y_1^1	0	0	0

Figure 4.3: Systolic design for the 1^{st} iteration of the S.O.R. iterative method.



$$\left.\begin{array}{l} g_i = (\omega, b_i, a_{ii}, x_i^0) \\ x_i^1 = ((((b_i - y_i^0 - y_i^1)\omega)/a_{ii}) + (1-\omega)x_i^0)) \end{array}\right\} i = 1, 2, \ldots, n$$

Figure 4.4: Snap shots of the 1st iteration for the S.O.R. iterative method.

4.3.2 The J.O.R. systolic design

The J.O.R. iterative method expressed by the equation (4.2.6) is very similar to the S.O.R. iterative method (with $J = L+U$). Comparing equations (4.2.6) and (4.2.13) it can be seen that the J.O.R. iterative method does not use the most recent approximations to compute the new approximation. If in the S.O.R. systolic design the old values of $\underline{x}^{(k)}$ are fed in by the host computer to the section computing Lx part then the design will work as the J.O.R. iterative method. Figure (4.5) shows the snap shots for the 1st iteration of the J.O.R. systolic design. The working of the design is the same as that of the S.O.R. systolic design.

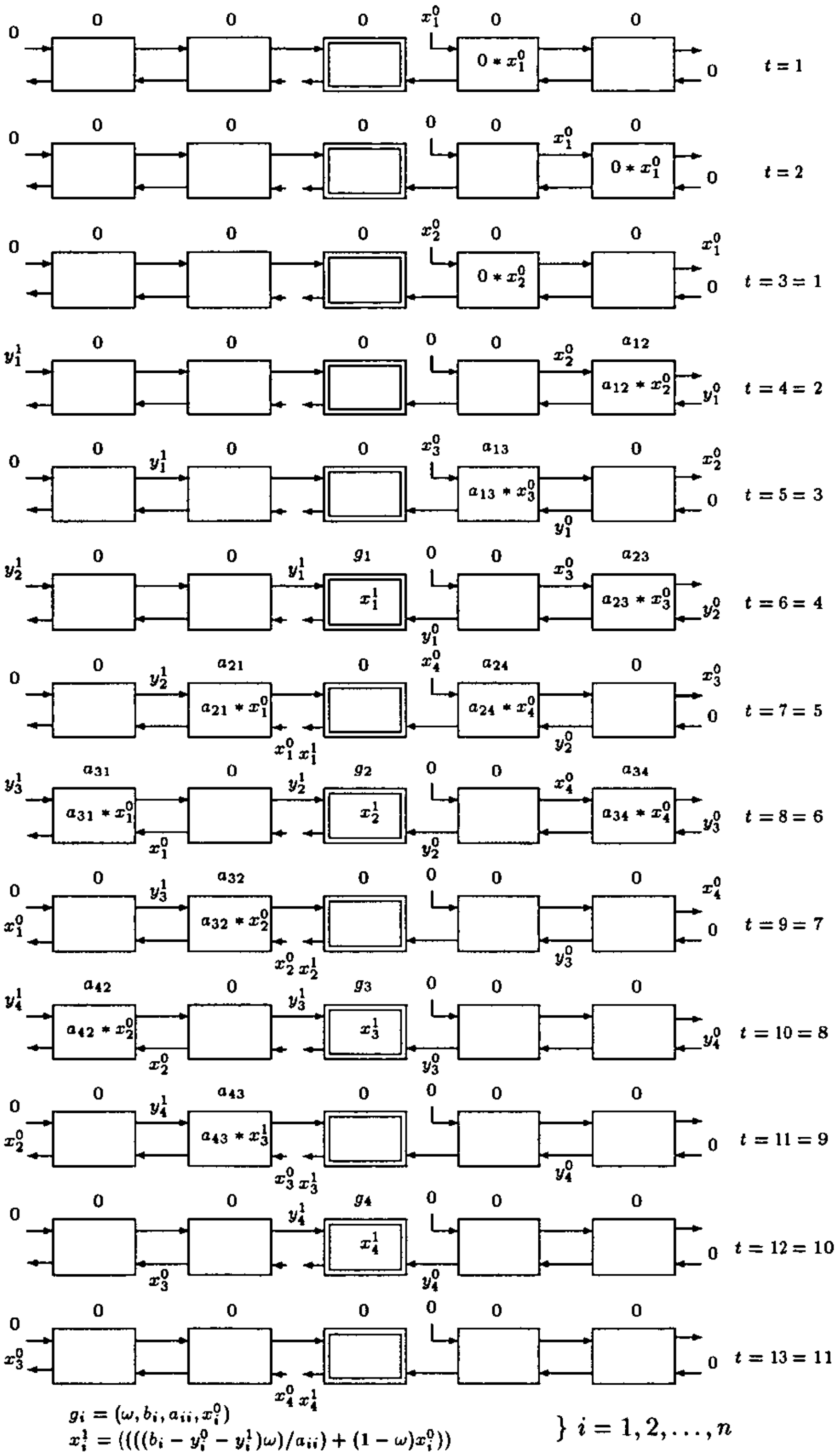

t = 1
t = 2
t = 3 = 1
t = 4 = 2
t = 5 = 3
t = 6 = 4
t = 7 = 5
t = 8 = 6
t = 9 = 7
t = 10 = 8
t = 11 = 9
t = 12 = 10
t = 13 = 11
$g_i = (\omega, b_i, a_{ii}, x_i^0)$
$x_i^1 = ((((b_i - y_i^0 - y_i^1)\omega)/a_{ii}) + (1-\omega)x_i^0))$
$\} \; i = 1, 2, \ldots, n$

Figure 4.5: Snap shots of the 1^{st} iteration for the J.O.R. iterative method.

4.3.3 The M.S.O.R. systolic design

The systolic design for the M.S.O.R. iterative method is shown in figure (4.6) and figure (4.7) shows the snap shots. The systolic designs for the Jacobi, Gauss-Seidel and S.O.R. iterative methods using cyclic reduction are presented in Margaritis and Evans [52]. The design is simulated for a $(5 * 5)$ tridiagonal matrix $(w = 3)$. The system to be solved is represented by equation (4.3.2).

$$\begin{bmatrix} a_{11} & a_{12} & & & \\ a_{21} & a_{22} & a_{23} & & 0 \\ & a_{32} & a_{33} & a_{34} & \\ 0 & & a_{43} & a_{44} & a_{45} \\ & & & a_{54} & a_{55} \end{bmatrix} * \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \end{bmatrix} \tag{4.3.2}$$

The scaled and block representation of the system is represented by equation (4.3.3).

$$\begin{bmatrix} m_{11} & & & \vdots & m_{12} & \\ & m_{33} & & \vdots & m_{32} & m_{34} \\ & & m_{55} & \vdots & & m_{54} \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ m_{21} & m_{23} & & \vdots & m_{22} & \\ & m_{43} & m_{45} & \vdots & & m_{44} \end{bmatrix} * \begin{bmatrix} x_1 \\ x_3 \\ x_5 \\ \cdots \\ x_2 \\ x_4 \end{bmatrix} = \begin{bmatrix} g_1 \\ g_3 \\ g_5 \\ \cdots \\ g_2 \\ g_4 \end{bmatrix} \tag{4.3.3}$$

The given tridiagonal matrix A, and b vector are converted in the form of equation (4.2.14) by the host computer. A simple systolic preprocessor can be designed to do so but no special gain will be achieved. The design works as following.

The "red" part consists of x_1, x_3 and x_5 and the "black" part consists of x_2 and x_4. Since the algorithm is divided into two parts hence, the hardware to implement consists of two (physically) identical parts also, one to solve the red part and the other to solve the black part. Each part consists of two IPS and a ADD IPS cell (this cell is similar to the DIV ADD IPS cell shown in figure (4.2) except that the DIV operation has been removed due to prescaling of the matrix A). The working of each part is same as the S.O.R. systolic design. The left hand side part of the hardware computes the successive red new approximations. The right hand side part uses the newly evaluated approximations (produced by left hand side part) to produce new black approximations. According to the algorithm the red part should be computed first as the black part is dependent on the results of the red part. Though this dependency exists by expanding equations (4.2.17) and (4.2.18) and observing equation (4.3.3) it can be seen that the computation of the black part can be started as soon as the partial dependent part has been solved. To understand this fact and the working of the array consider the computation of x_2^1 (black part) and x_5^1 (red part) which are given by,

$$x_2^1 = \omega(m_{21}x_1^1 + m_{23}x_3^1 + g_2) + (1-\omega)x_2^0$$

and $x_5^1 = \omega(m_{54}x_4^0 + g_5) + (1-\omega)x_5^0$ respectively.

Note that host provides $m_{ij} = -a_{ij}/a_{ii}$ for $i, j = 1, 2, \ldots, n; i \neq j$.

For simplicity consider that the whole system (array) is numbered from left to right with processor 1 being the left most cell and processor 6 being the right most cell. At time step 4, x_1^1 is available, indicating that the partial computation for the black portion can be started. At time step 5, x_1^1 moves to processor 6 which computes $y_2^1 = m_{21}x_1^1$. Also note that processor 3 computes x_3^1, which is needed by the processor 5 in the next step. Processor 1 computes $y_5^1 = m_{54}x_4^0$. At time step 6, processor 5 computes $m_{23}x_3^1$ and $y_3^1 = m_{21}x_1^1 + m_{23}x_3^1$ and y_5^1 moves through processor 2 unaltered. At time step 7, processor 4 computes x_2^1 and processor 3 computes x_5^1.

The area and time requirements for a single iteration are shown in table (4.1). Table (4.1) is valid only for the tridiagonal system of linear equations. Note that as the size of the matrix is increased the utilisation of the array is im-

Method	Order	Time	Area
M.S.O.R.	n-odd	$(n+5)$	4 IPS + 2 ADD cells
M.S.O.R.	n-even	$(n+6)$	4 IPS + 2 ADD cells

Table 4.1: Area and time requirements for M.S.O.R. design 1.

Method	Order	Time	Area
M.S.O.R.	n-odd	$(2n+2)$	2 IPS + 1 ADD cells
M.S.O.R.	n-even	$(2n+4)$	2 IPS + 1 ADD cells

Table 4.2: Area and time requirements for M.S.O.R. design 2.

proved, for large n (i.e. $n > 100$) the utilisation of the M.S.O.R. linear systolic array is about 50%.

The other possible way to design the systolic array for the M.S.O.R. iterative method is to let the red part be completed and then to start the black part on the same hardware. Such a systolic design can be implemented using half the array, compared to the systolic design represented by figure (4.6). The new design is represented by figure (4.8). The area is half but it takes more time to complete the single iteration. The utilisation is better for small n as compared to the previous design (design 1) and is same for large n (i.e. $n > 100$). The time taken by design 2 to complete one iteration is almost twice. So if a fast response time is required then one must use design 1 and if it is not important then the design 2 is preferable due to low area, hence cost. The area and time requirements for design 2 are shown in table (4.2).

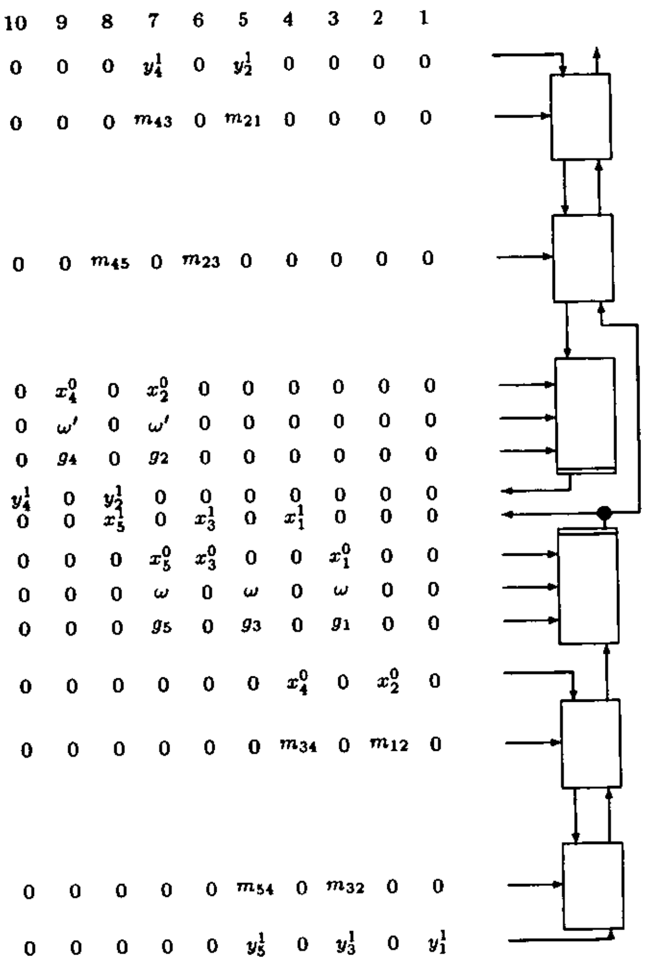
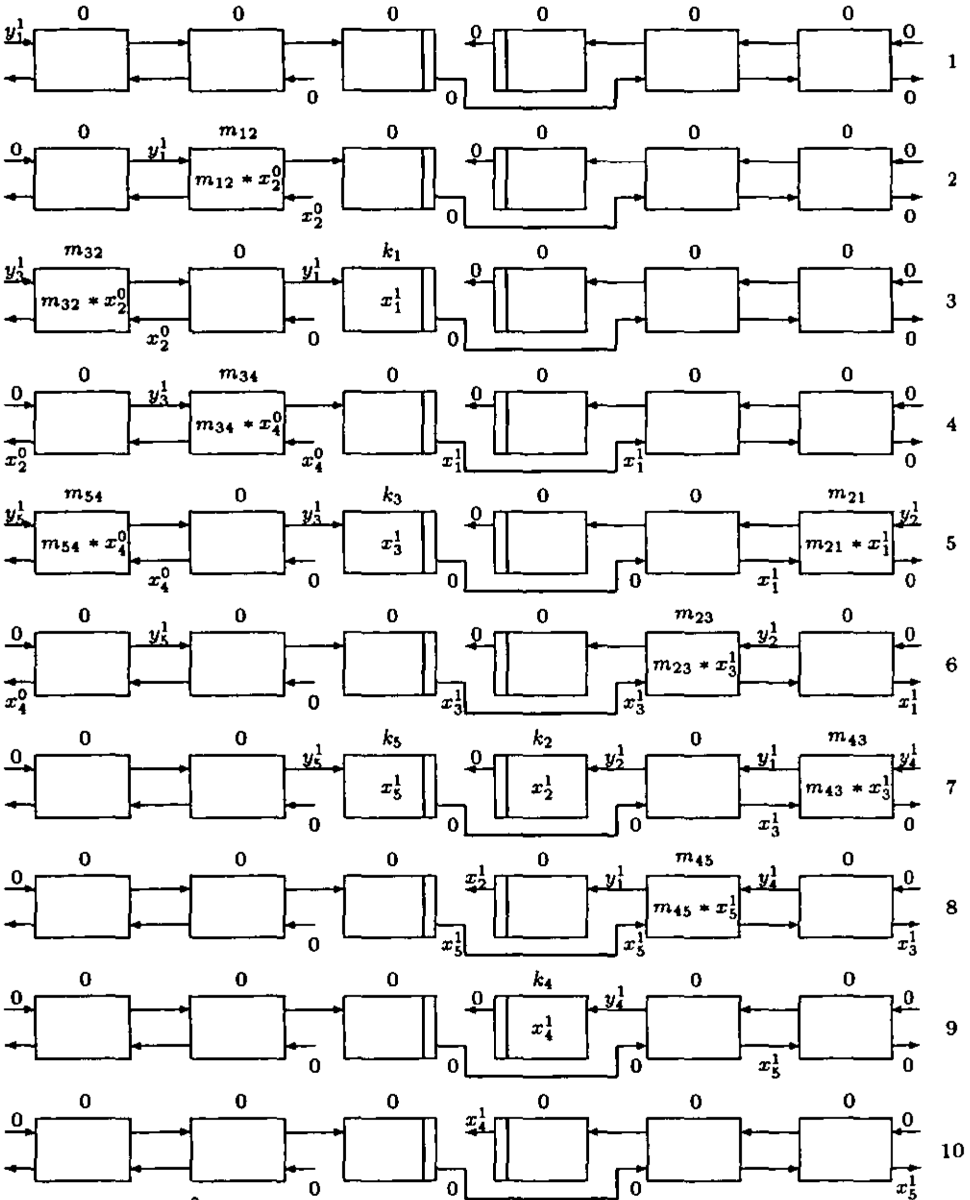

10 9 8 7 6 5 4 3 2 1
0 0 0 y_4^1 0 y_2^1 0 0 0 0
0 0 0 m_{43} 0 m_{21} 0 0 0 0
0 0 m_{45} 0 m_{23} 0 0 0 0 0
0 x_4^0 0 x_2^0 0 0 0 0 0 0
0 ω' 0 ω' 0 0 0 0 0 0
0 g_4 0 g_2 0 0 0 0 0 0
y_4^1 0 y_2^1 0 0 0 0 0 0 0
0 0 x_5^1 0 x_3^1 0 x_1^1 0 0 0
0 0 0 x_5^0 x_3^0 0 0 x_1^0 0 0
0 0 0 ω 0 ω 0 ω 0 0
0 0 0 g_5 0 g_3 0 g_1 0 0
0 0 0 0 0 0 x_4^0 0 x_2^0 0
0 0 0 0 0 0 m_{34} 0 m_{12} 0
0 0 0 0 0 m_{54} 0 m_{32} 0 0
0 0 0 0 0 y_5^1 0 y_3^1 0 y_1^1

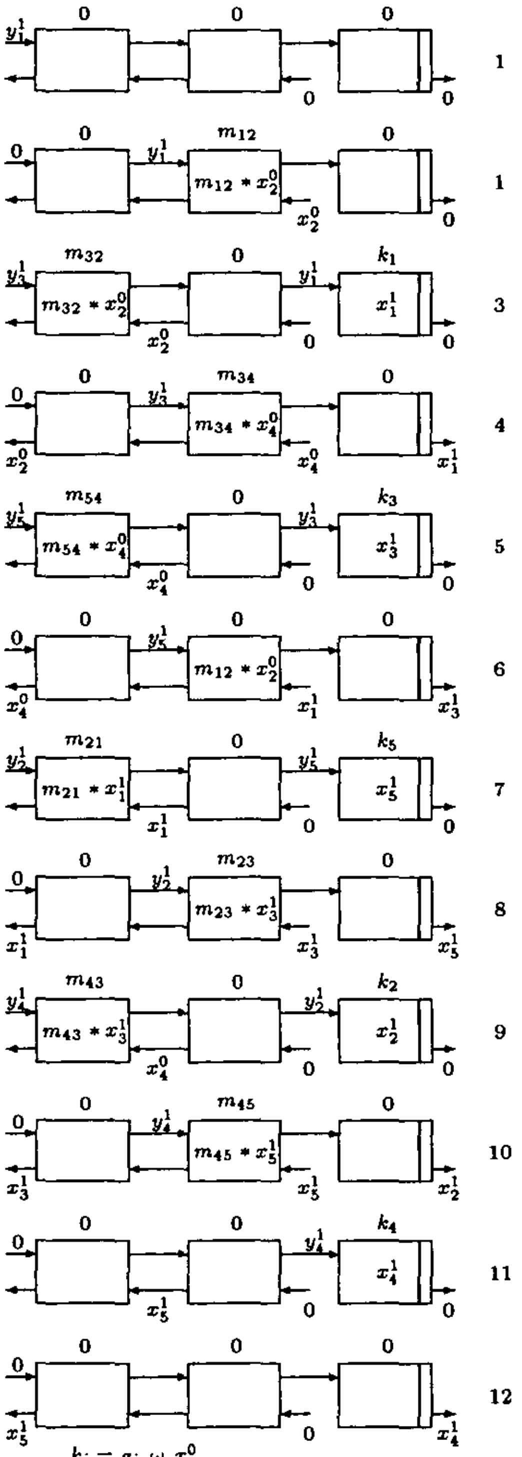
Figure 4.6: Systolic design for the M.S.O.R. iterative method.



$k_i = g_i, \omega, x_i^0$
$x_i^1 = \omega(y_i^1 + g_i) + (1 - \omega)x_i^0$ $\}\ i = 1, 3, \ldots, n$

$k_i = g_i, \omega', x_i^0$
$x_i^1 = \omega'(y_i^1 + g_i) + (1 - \omega')x_i^0$ $\}\ i = 2, 4, \ldots, n$

Figure 4.7: Snap shots of the M.S.O.R. systolic design 1.



$$k_i = g_i, \omega, x_i^0$$
$$x_i^1 = \omega(y_i^1 + g_i) + (1 - \omega)x_i^0 \quad \} \; i = 1,3,\ldots,n$$
$$k_i = g_i, \omega', x_i^0$$
$$x_i^1 = \omega'(y_i^1 + g_i) + (1 - \omega')x_i^0 \quad \} \; i = 2,4,\ldots,n$$

Figure 4.8: Snap shots of the M.S.O.R. systolic design 2.

4.3.4 The S.S.O.R. systolic design

The systolic design for the S.S.O.R. iterative method with and without the Conrad-Wallach technique has been simulated as follows. The systolic design is shown in figures (4.3) and (4.9). Figure (4.3) represents the very first iteration and figure (4.9) shows the successive iterations of the method. One fact to note here is that the extra hardware required for computing $U\underline{x}^{(0)}$ is used only once in the first iteration. If the initial guess is always taken to be zero then this computation is not needed as the $U\underline{x}^{(0)}$ vector will be zero, however this may not always be the best assumption. The systolic design shown in figures (4.3) and (4.9) is for a (4×4) banded matrix with q less than or equal to p. This is required due to the nature of the problem. The snap shots are shown in figures (4.4) and (4.10) for the first and the successive iterations respectively. The area requirements for the S.S.O.R. with and without the Conrad-Wallach technique are the same, but the time requirements are different, as shown in table (4.3) (w is the bandwidth of the system and is 5 in our design, k represents the number of iterations performed). The DIV ADD IPS and IPS cells are shown in figure (4.2). In the second and successive iterations as stated above the upper portion of the hardware is not used (see figure (4.9)). The host computer feeds the $L\underline{x}^{(k+1)}$ and $U\underline{x}^{(k+2)}$ and so on, which are precomputed in the most recent previous iteration. The working of the array is similar to that of the S.O.R. linear systolic array.

Method	Architecture	Time	Area (IPS) + DIV-ADD-IPS
S.S.O.R.	Normal	$k(2n+w)$	$(w-1)+1$
S.S.O.R.	Conrad-Wallach	$(2n+w)+(k-1)(w+2n-3)$	$(w-1)+1$

Table 4.3: Area and time requirements for S.S.O.R. iterative method.

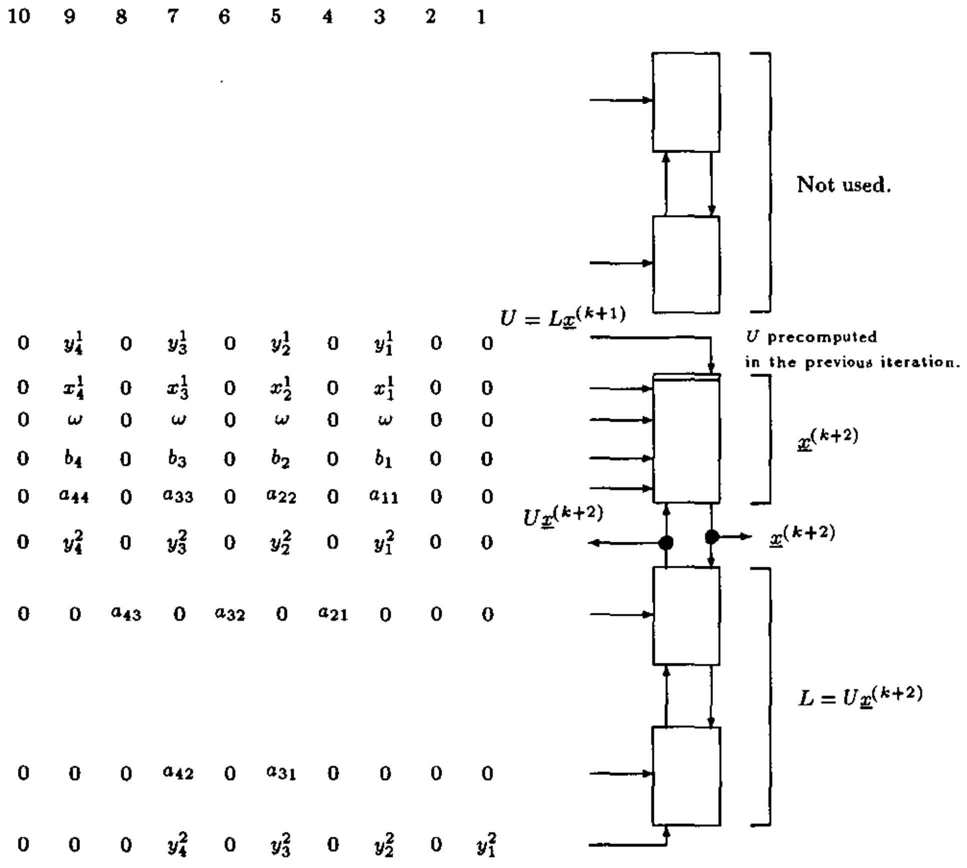

10 9 8 7 6 5 4 3 2 1
Not used.
U = Lx^(k+1)
U precomputed in the previous iteration.
0 y41 0 y31 0 y21 0 y11 0 0
0 x41 0 x31 0 x21 0 x11 0 0
0 ω 0 ω 0 ω 0 ω 0 0
0 b4 0 b3 0 b2 0 b1 0 0
0 a44 0 a33 0 a22 0 a11 0 0
x^(k+2)
Ux^(k+2)
0 y42 0 y32 0 y22 0 y12 0 0
x^(k+2)
0 0 a43 0 a32 0 a21 0 0 0
L = Ux^(k+2)
0 0 0 a42 0 a31 0 0 0 0
0 0 0 y42 0 y32 0 y22 0 y12

Figure 4.9: Systolic design for the 2^{nd} iteration of S.S.O.R. iterative method.

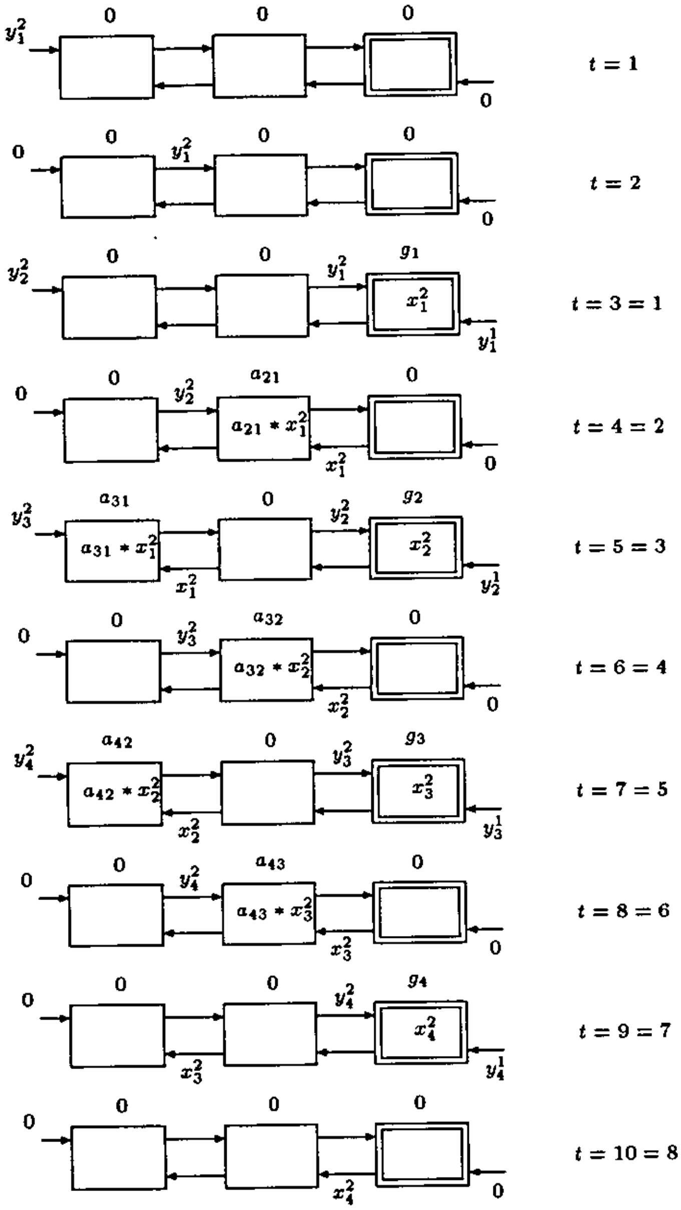


Figure 4.10: Snap shots for the 2^{nd} iteration of the S.S.O.R. iterative method.

4.3.5 The U.S.S.O.R. systolic design

The systolic design remains the same except that in figure (4.9) and figure (4.10) ω is replaced by $\hat{\omega}$ i.e. the second overrelaxation parameter. The time and area requirements remain the same but the host computer needs to do a little extra work to take care of two overrelaxation factors.

4.4 Further design improvements

The J.O.R., S.O.R. and S.S.O.R. systolic designs can be improved by observing the following facts:

a) In figure (4.3) the factor marked x_1 in the upper triangular matrix need not be implemented in the design since this factor will always be zero. This can be seen by expanding equation (4.2.13).

b) In figure (4.9) the factor marked y_1 is not required to be computed as it will always be zero as is apparent from figure (4.10).

c) The DIV ADD IPS cell can be simplified by using the systolic preprocessor. The preprocessor divides all the rows by their respective diagonals and multiplies the whole matrix by ω, (also the $\underline{b}$ is modified) as this is done in iterations.

Table (4.4) presents the area and time requirements for the improved design (k represents the number of iterations performed and w the bandwidth of the system, 5 in the example discussed).

4.4.1 Pipelining

Each iteration in the design is pipelined within itself but successive iterations cannot be pipelined (see Berzins, Buckley and Dew [7]) due to the reversal of the equations in the 2^{nd} half iterations. This makes the systolic design

Method	Architecture	Time	Area(IPS)+DIV-ADD-IPS
S.S.O.R.	Normal	$k(w+2n-2)$	$w-1+1$
S.S.O.R.	Conrad-Wallach	$(w+2n-2)+(k-1)(w+2n-5)$	$w-1+1$

Table 4.4: Area and time requirements for the improved S.S.O.R. design.

simple, with no delays and extra area requirements. The price paid here is the repumping of data by the host computer. The systolic array is re-used in this design for the computation of successive iterations.

4.4.2 Systolic design for pumping data

The architectural consideration about the pumping of data is also important. As the host computer is repumping data repeatedly for the iterations, there are advantages and disadvantages which are outlined below.

The host computer cannot do any other useful work and the memory used to store the matrix and vectors cannot be overwritten or reclaimed by the host computer.

There are two ways by which data can be pumped from the host computer to the linear systolic array:

1. By saving the matrix and vectors in the memory and after each iteration reverse the data in these locations. This strategy degrades the performance of the design.
2. By saving the matrix and vectors in the memory along with their reverse images. This increases the memory requirements for the host computer but is worth consideration.

Method 1 requires less memory and hence more host computing time. Method 2 on the other hand requires more memory but less host computing time.

If the data is pumped every iteration from the host computer then the performance of the array will be degraded due to a slow host computer (communication overheads). If the data is permanently stored in the array then a large resident memory is required by each processor. An example of this architecture is WARP (Tseng [77]). Also, the reverse image technique to be implemented needs attention so that the overheads are reduced to a minimum.

4.5 Conclusions

The methods simulated above are faster, i.e. the computational time is less and so is the area requirement as compared to fully pipelined designs. The S.S.O.R. iterative method using the Conrad-Wallach technique is 23.08% faster than the usual S.S.O.R. iterative method. The improved design is 20.00% faster than the improved normal S.S.O.R. iterative method and is 38.46% faster than the unimproved normal S.S.O.R. iterative method. This speedup is calculated for $n = 4$.

The advantage of ordering the given equations into red, black points and using the concept of block matrices results in a speedup to a certain degree which can be achieved. However, the area requirement is definitely increased. The speedup will be less than double due to the dependencies. As can be seen from figure (4.7) the computation x_2^1 cannot be started until x_1^1 and x_3^1 have been computed. The area requirement also depends on the order of the matrix and also on the even and odd orderings of the equations. The same will be true for the time requirements.

Since a very special case has been discussed in this chapter no general comparisons can be made. However a comparison can be made for the area and time requirements between the M.S.O.R. and S.O.R. (using two overrelaxation parameters alternately) iterative methods. Also, this comparison can be extended for the overall methods, i.e. which one converges faster for the special case discussed above.

Chapter 5

Further Matrix Iterative Methods

In this chapter we present the systolic designs for the stationary and non-stationary second order Richardson iterative methods for solving large linear systems generated from discretisation of boundary value problems. It is well known that the Chebyshev polynomials have the following "Equal ripple" property, i.e. the Chebyshev polynomial $T_k(x)$ oscillates between the interval [-1,1] exactly k times as x goes from -1 to 1. This property of the Chebyshev polynomials is used to reduce the error norm of the iterative method and hence achieve fast convergence. The systolic array is reusable and each cell has its own local memory. The design is highly parallel and pipelined.

Later in the chapter the systolic design for the Accelerated Overrelaxation iterative method (see Hadjidimos [33]) for solving large systems of linear equations is presented based on the VLSI techniques discussed earlier, in Evans and Haider [18], for the Jacobi, Gauss-Seidel, S.O.R. and S.S.O.R. iterative methods. Finally, a similar strategy (see Conrad and Wallach [11], Evans and Haider [18]) is used to accomplish the S.A.O.R. iterative method involving no extra computational work.

5.1 Introduction

The equation,

$$A\underline{x} = \underline{b}, \tag{5.1.1}$$

represents a system of n linear equations in n unknowns, where A is a $(n * n)$ coefficient matrix, $\underline{x}$ and $\underline{b}$ represent the n unknown and known constants respectively. If the matrix A is split such that,

$$A = D + L + U, \tag{5.1.2}$$

where D, L, U are $(n * n)$ matrices, with D containing only the diagonal elements of A, U and L are the strictly upper and lower triangular matrices with zero diagonals.

5.1.1 The first order Richardson iterative method

The first order Richardson iterative method is defined as,

$$\underline{x}^{(k+1)} = \underline{x}^{(k)} + \alpha(\underline{b} - A\underline{x}^{(k)}). \tag{5.1.3}$$

The residual vector is given by,

$$\underline{r}^{(k)} = \underline{b} - A\underline{x}^{(k)}$$

and is the measure of the error in the method. The factor α is used to get a faster convergence (accelerate the iterative process). The error $\underline{e}^k$ is defined to be of the form $\underline{x}^k - A^{-1}\underline{b}$ (where $A^{-1}\underline{b}$ is the exact solution of the equation (5.1.3)). The equation (5.1.3) can also be written in the form

$$\underline{x}^{(k)} = \underline{x}^{(k-1)} + \alpha(\underline{b} - A\underline{x}^{(k-1)}). \tag{5.1.4}$$

A recurrence relation for the error vectors can be defined from equations (5.1.3) and (5.1.4) and is of the form

$$\underline{e}^{(k+1)} = \underline{e}^{(k)} + \alpha(-A\underline{e}^{(k)}),$$

where

$$\underline{e}^{(k+1)} = \underline{x}^{(k+1)} - \underline{x}^{(k)}.$$

The method is said to be a stationary iterative method if α remains constant (the error operator remains constant through out the iterative process). Now the above recurrence relation can be written in the form

$$\underline{e}^{(k+1)} = (I - \alpha A)\underline{e}^{(k)}.$$

Let μ_i be the eigenvalues of the matrix A, which implies that $A\underline{x}_i = \mu_i\underline{x}_i$ corresponding to its eigenvectors $\underline{x}_i$, $i = 1, 2, \ldots, n$. Since A is positive definite, all $\mu_i > 0$. If λ_i be the eigenvalues of $I - \alpha A$ corresponding to its eigenvectors $\underline{x}_i$ implies that $(I - \alpha A)\underline{x}_i = \lambda_i\underline{x}_i$ or,

$$I\underline{x}_i - \alpha A\underline{x}_i = \lambda_i\underline{x}_i.$$

Therefore we have for each value of i,

$$\underline{x}_i - \alpha(\mu_i\underline{x}_i) = \lambda_i\underline{x}_i,$$

$$(1 - \alpha\mu_i)\underline{x}_i = \lambda_i\underline{x}_i,$$

$$\lambda_i = (1 - \alpha\mu_i) \quad i = 1, 2, \ldots, n.$$

Since for convergence we require $|\lambda_i| < 1$ then,

$$|(1 - \alpha\mu_i)| < 1.$$

Now using the modulus property i.e. $|z| < a \Rightarrow -a < z < a$ we get,

$$-1 < 1 - \alpha\mu_i < 1$$

$$(-1) - 1 < (1 - \alpha\mu_i) - 1 < (1) - 1$$

$$-2 < -\alpha\mu_i < 0.$$

Now for convergence

1. we can establish that

   $$2 > \alpha\mu_i.$$

   Since $\mu_i > 0$, then $\frac{2}{\mu_i} > \alpha$ or

   $$\alpha < \frac{2}{\mu_i} \quad \forall i = 1, 2, \ldots, n.$$

   As this is true for $\forall i$, then it is also true for some i such that,

   $$\alpha < \frac{2}{\max_i \mu_i}. \tag{5.1.5}$$

2. $-\alpha\mu_i < 0$ implies that

   $$\alpha\mu_i > 0$$

   therefore,

   $$\alpha > 0 \text{ since } \mu_i > 0.$$

   Hence it is established that, the convergence range of the Richardson iterative method is,

   $$0 < \alpha < \frac{2}{max_i \mu_i}. \tag{5.1.6}$$

   This means for the first order Richardson iterative method to achieve accelerated convergence, α should be in the bound defined by equation (5.1.6).

Suppose all we know about the μ_i is that they lie in an interval $[a, b]$ and $0 < a < b < \infty$. For every function $\mid 1 - \alpha\mu \mid$ assumes its maximum at one of the end points $\mu = a$ or $\mu = b$. The best choice of α is the one for which,

$$1 - \alpha a = -1(1 - \alpha b)$$

to give

$$\alpha = \frac{2}{a + b}$$

with this choice of α, the maximum value of $1 - \alpha\mu_i$ at the end point a is given by,

$$1 - \frac{2}{a + b}a$$

which gives

$$\frac{b - a}{a + b}$$

and it must be true that

$$\mid 1 - \alpha\mu_i \mid < \frac{b - a}{b + a} < 1.$$

This means that the convergence factor for this method with a chosen constant α is bounded by a certain function of $\frac{b}{a}$ or P the condition number. The rate of convergence is given by,

$$R = -\log\left[\frac{-1 + \frac{b}{a}}{1 + \frac{b}{a}}\right],$$

$$R = -\log\left[\frac{-1 + P}{1 + P}\right],$$

$$R = \log\left[\frac{-1 + P}{1 + P}\right]^{-1},$$

$$R = \log\left[\frac{1+P}{-1+P}\right],$$

$$R = \log\left[\frac{1+\frac{1}{P}}{1-\frac{1}{P}}\right]$$

or,

$$R = \log(1+\tfrac{1}{P}) - \log(1-\tfrac{1}{P}) \ .$$

Expanding the log series we get,

$$R = (\frac{1}{P} - \frac{1}{2P^2} + \frac{1}{3P^3} - \cdots) - (-\frac{1}{P} + \frac{1}{2P^2} - \frac{1}{3P^3} + \cdots)$$

$$R = \frac{2}{P} + \frac{2}{3P^3} + \cdots$$

Neglecting the higher powers of P we get,

$$R \simeq \frac{2}{P}.$$

5.1.2 The first order Richardson iterative method with Chebyshev acceleration

If the parameter α is varied with each iteration then the iterative method obtained is known as a nonstationary method. Now the equation (5.1.3) can be written as,

$$\underline{x}^{(k+1)} = \underline{x}^{(k)} + \alpha_k(\underline{b} - A\underline{x}^{(k)}) \qquad (5.1.7)$$

to represent the first order Richardson iterative method with Chebyshev acceleration. Also,

$$\underline{e}^{(k)} = \underline{e}^{(k-1)} + \alpha_{k-1}(-A\underline{e}^{(k-1)})$$

which is equivalent to a nonstationary method

$$\underline{e}^{(k)} = (I - \alpha_0 A)(I - \alpha_1 A)(I - \alpha_2 A)\ldots(I - \alpha_{k-1} A)\underline{e}^{(0)}.$$

or can be written in the form

$$\underline{e}^{(k)} = P_k(A)\underline{e}^{(0)},$$

where

$$P_k(x) = \prod_{i=0}^{k-1}(1 - \alpha_i x)$$

is a polynomial in x with the property $P_k(0) = 1$. The numbers $\frac{1}{\alpha_i}$ are the zeros of the polynomial P_k and $P_k(\frac{1}{\alpha_i}) = 0$, can cause round off difficulties. It is desired that $\underline{e}^k$ should be made as small as possible, hence that $P_k(A)$ be made as small as possible. The obvious choice is $\alpha_i = \lambda_i^{-1}$, where λ_i are the eigenvalues of A. This is not possible since we have no predetermined knowledge of the λ_i. So an alternative procedure of making $P_k(x)$ small over the interval [c,d] is adopted. If the eigenvalues of A lie in the range, $c \leq x \leq d < 1$ where c and d represent the minimum and maximum eigenvalues of A and are real we have $\| \underline{e}^{(k)} \| \leq \| P_k(A) \| \| \underline{e}^0 \|$, and since

$$\| P_k(A) \| = \rho(P_k(A)) = \max_{c \leq x \leq d} | P_k(x) |$$

A polynomial of degree k is required such that

$$\max_{c \leq x \leq d} | P_k(x) |$$

is a minimum under the constraint $P_k(0) = 1$. Such a polynomial was given by W. Markoff (in 1892) and is defined by,

$$P_k(x) = \frac{T_k(x)}{T_k(d)} \tag{5.1.8}$$

where $T_k(x)$ is the Chebyshev polynomial and $P_k(x) = 1$ and

$$\max_{-1 \leq x \leq 1} | P_k(x) | = \frac{1}{T_k(d)}.$$

Chebyshev polynomials of degree k are represented by $\{T_k\}$. For $x \in [1, -1]$, define

$$T_k(x) = \cos[k \arccos(x)] \text{ for each } k \geq 0$$

Let $\arccos(x) = \theta \Rightarrow$ x $= \cos(\theta)$

$$\Rightarrow \quad T_k(\theta) = \cos(k\theta), \quad \theta \in [0, \pi]$$

A recurrence relation can be derived by noting that

$$\begin{aligned} T_{k+1}(\theta) &= \cos((k+1)\theta) \\ &= \cos(k\theta)\cos(\theta) - \sin(k\theta)\sin(\theta) \end{aligned}$$

and

$$\begin{aligned} T_{k-1}(\theta) &= \cos((k-1)\theta) \\ &= \cos(k\theta)\cos(\theta) + \sin(k\theta)\sin(\theta) \end{aligned}$$

So $T_{k+1}(\theta) = 2\cos(k\theta)\cos(\theta) - T_{k-1}(\theta)$.

Returning to the variable x, we obtain

$$T_{k+1}(x) = 2xT_k(x) - T_{k-1}(x) \tag{5.1.9}$$

The Chebyshev polynomials are now easily obtained in a sequential manner by using the three term recurrence relation obtained from equation (5.1.9),

$$\begin{aligned} &T_0(x) = 1 \\ &T_1(x) = x \\ &T_{k+1}(x) = 2xT_k(x) - T_{k-1}(x) \end{aligned}$$

So by using these concepts we can write

$$P_k(x) = \frac{T_k\left[\frac{b+a-2x}{b-a}\right]}{T_k(y_0)}$$

where $y_0 = \frac{b+a}{b-a}$ and $T_k(y)$ is a Chebyshev polynomial of degree k adjusted in the interval $-1 \leq y \leq 1$. Then, $P_k(x)$ is a Chebyshev polynomial of degree k adjusted to the interval $a \leq x \leq b$ and scaled so that $P_k(0) = 1$. Let $y = \frac{b+a-2x}{b-a}$ then y_0 is the value of y for $x = 0$, i.e. $y_0 = \frac{b+a}{b-a}$ for $x = a$, $y = 1$

and for $x = b$, $y = -1$. The $\mid P_k(y) \mid$ has a maximum value $\frac{1}{T_k(y_0)}$. The average rate of convergence per iteration, defined as the lower bound is not less than,

$$R = -\frac{1}{(k+1)} \log \left[\frac{T_{k+1}\left(\frac{b+a-2x}{b-a}\right)}{T_{k+1}\left(\frac{b+a}{b-a}\right)} \right]. \tag{5.1.10}$$

Since the maximum absolute value of the numerator is unity hence

$$R = -\frac{1}{(k+1)} \log \left[\frac{1}{T_{k+1}\left(\frac{b+a}{b-a}\right)} \right]. \tag{5.1.11}$$

As $b/a = P$ then equation (5.1.11) can be written as,

$$R = \frac{-1}{(k+1)} \log \left[T_{k+1} \frac{P+1}{P-1} \right]^{-1}.$$

Since $\frac{P+1}{P-1} > 1$ then $T_k(x)$ is defined as,

$$T_k(x) = \cosh(k \cosh^{-1} x)$$

and for large k we have,

$$\lim_{k \to \infty} \log \left[\cosh(k \cosh^{-1} x) \right]^k = \cosh^{-1} x.$$

Let $m = \cosh^{-1} x$, then

$$\cosh m = x = \frac{e^m + e^{-m}}{2}$$

so $e^{2m} - 2xe^m + 1 = 0$

or $e^m = x \pm \sqrt{x^2 - 1}$ since $e^m > 0$ hence,

$e^m = x + \sqrt{x^2 - 1}$ or,

$m = \log(x + \sqrt{x^2 - 1})$ or,

$$\cosh^{-1} x = \log \left[x + \sqrt{x^2 - 1} \right]$$

$$= \log\left[\frac{P+1}{P-1} + \sqrt{\frac{(P+1)^2}{(P-1)^2} - 1}\right]$$

$$= \log\left[\frac{P+1}{P-1} + \sqrt{\frac{(P+1)^2 - (P-1)^2}{(P-1)^2}}\right]$$

$$= \log\left[\frac{P+1}{P-1} + \frac{\sqrt{4P}}{(P-1)}\right]$$

$$= \log\left[\frac{P+1+2\sqrt{P}}{(P-1)}\right]$$

$$= \log\left[\frac{(\sqrt{P}+1)^2}{(P-1)}\right]$$

$$= \log\left[\frac{(\sqrt{P}+1)^2}{(\sqrt{P}-1)(\sqrt{P}+1)}\right]$$

$$= \log\left[\frac{(\sqrt{P}+1)}{(\sqrt{P}-1)}\right]$$

expanding the series and neglecting higher powers of P we get

$$R = \frac{2}{\sqrt{P}}.$$

The operational aspects of this method are as follows. One must choose in advance the k parameters α_i, given only a, b, k and the value of the ratio $\frac{|r^{(0)}|}{|r^{(k)}|}$ at which the iteration is to be terminated. The best choice is given by

$$\alpha_i = \frac{2}{\left[(a+b) - (b-a)\cos\frac{(2i-1)}{2k}\right]} \quad i = 1, 2, \ldots, k.$$

This choice would ensure fastest convergence in the absence of rounding errors but in practice the iteration is very sensitive to rounding errors when $\frac{a}{b}$ is very small. This sensitivity, often a feature of nonstationary iterations, is due to the fact that for some values of α_i the later factors of

$$P_k(x) = \prod_{i=0}^{k-1}(1 - \alpha_i x)$$

are large and the earlier factors small. Therefore, some of the rounding errors committed in the earlier iterations are often accentuated instead of attenuated by the later iterations.

An improved version which is less sensitive to round off error was suggested by Stiefel, in [73], and is given below.

5.1.3 The second order Richardson iterative method

The equation,

$$\underline{x}^{(k+1)} = \underline{x}^{(k)} + \alpha(\underline{b} - A\underline{x}^{(k)}) + \beta(\underline{x}^{(k)} - \underline{x}^{(k-1)}) \tag{5.1.12}$$

represents the second order Richardson iterative method (see Frankel [29], Young [82], Golub and Varga [31], Hageman and Young [34], Haider and Evans [35]). The second order methods involve two previous iterates. For the first iteration the well known first order form of the Richardson method is used, i.e.

$$\underline{x}^{(1)} = \underline{x}^{(0)} + \alpha(\underline{b} - A\underline{x}^{(0)}).$$

For later iterations both $\underline{x}^{(k-1)}$ and $\underline{x}^{(k)}$ are used with two parameters α and β chosen to speed up the convergence. These parameters remain constant throughout the iteration process. The error at the $(k+1)^{th}$ iterate is defined as,

$$\underline{e}_i^{(k+1)} = [1 + \beta - \alpha(1 - \lambda_i)]\,\underline{e}_i^{(k)} - \beta\underline{e}_i^{(k-1)},$$

where $(1 - \lambda_i)$ are the eigenvalues of A. Let γ_i be the eigenvalues of the matrix associated with the iteration process. Then

$$\underline{e}_i^{(k+1)} = \gamma_i\underline{e}_i^{(k)} = \gamma_i^2\underline{e}_i^{(k-1)}.$$

Now we can write,

$$\underline{e}_i^{(k+1)} = \left[1 + \beta - \alpha(1 - \lambda_i) - \frac{\beta}{\gamma_i}\right]\underline{e}_i^{(k)}$$

or

$$\gamma_i^2 - (1 + \beta - \alpha + \alpha\lambda_i)\,\gamma_i + \beta = 0.$$

Let $k_i = (1 + \beta - \alpha + \alpha\lambda_i)$ in the above equation, which represents a quadratic equation in γ, which gives,

$$\gamma_i^2 - (k_i)\,\gamma_i + \beta = 0.$$

The above equation gives the solution,

$$\gamma_i = \frac{k_i \pm \sqrt{k_i^2 - 4\beta}}{2}. \tag{5.1.13}$$

If α and β are chosen so that $(k_i^2 - 4\beta)$ is negative for all λ_i, then all the γ_i will be complex and all $\mid \gamma_i \mid$ identical. If γ_i are real and identical then,

$$k_1^2 - 4\beta = 0 \Rightarrow k_1 = \pm 2\sqrt{\beta}$$

$$k_n^2 - 4\beta = 0 \Rightarrow k_n = - \pm 2\sqrt{\beta}$$

$$1 + \beta - \alpha + \alpha\lambda_1 = 2\sqrt{\beta}$$

$$1 + \beta - \alpha + \alpha\lambda_n = -2\sqrt{\beta}.$$

Let $\mu_i = 1 - \lambda_i$ in the above two equations we get,

$$1 + \beta - \alpha\mu_1 = 2\sqrt{\beta} \tag{5.1.14}$$

$$1 + \beta - \alpha\mu_n = -2\sqrt{\beta}. \tag{5.1.15}$$

Solving for α, and adding equations (5.1.14) and (5.1.15) we get

$$2 + 2\beta - \alpha\,(\mu_1 + \mu_n) = 0$$

or

$$\alpha = 2\left(\frac{1 + \beta}{\mu_1 + \mu_n}\right). \tag{5.1.16}$$

Substituting the value of α from equation (5.1.16) in equation (5.1.14) we get,

$$1+\beta-\mu_1\left(\frac{2+2\beta}{\mu_1+\mu_n}\right)=2\sqrt{\beta}$$

$$1+\beta-\frac{2\mu_1}{\mu_1+\mu_n}-\frac{2\mu_1}{\mu_1+\mu_n}\beta=2\sqrt{\beta}$$

$$\beta\left(1-\frac{2\mu_1}{\mu_1+\mu_n}\right)-2\sqrt{\beta}+\left(1-\frac{2\mu_1}{\mu_1+\mu_n}\right)=0$$

$$\beta\left(\frac{\mu_1+\mu_n-2\mu_1}{\mu_1+\mu_n}\right)-2\sqrt{\beta}+\left(\frac{\mu_1+\mu_n-2\mu_1}{\mu_1+\mu_n}\right)=0$$

$$\beta\left(\mu_n-\mu_1\right)-2\sqrt{\beta}\left(\mu_1+\mu_n\right)+\left(\mu_n-\mu_1\right)=0. \tag{5.1.17}$$

Equation (5.1.17) is a quadratic equation in $\sqrt{\beta}$ and can be solved for $\sqrt{\beta}$, and we get

$$\sqrt{\beta}=\frac{2\left(\mu_1+\mu_n\right)\pm\sqrt{4\left(\mu_1+\mu_n\right)^2-4\left(\mu_n-\mu_1\right)^2}}{2\left(\mu_n-\mu_1\right)}$$

$$\sqrt{\beta}=\frac{2(\mu_1+\mu_n)\pm2\sqrt{\mu_1^2+\mu_n^2+2\mu_1\mu_n-\mu_n^2-\mu_1^2+2\mu_1\mu_n}}{2(\mu_n-\mu_1)}$$

$$\sqrt{\beta}=\frac{(\mu_1+\mu_n)\pm\sqrt{4\mu_1\mu_n}}{(\mu_n-\mu_1)}$$

$$\sqrt{\beta}=\frac{(\mu_1+\mu_n)\pm2\sqrt{\mu_1\mu_n}}{(\mu_n-\mu_1)}$$

$$\sqrt{\beta}=\frac{(\sqrt{\mu_n}\pm\sqrt{\mu_1})^2}{(\mu_n-\mu_1)}$$

since we want β to be less than unity hence we take negative value,

$$\sqrt{\beta}=\frac{(\sqrt{\mu_n}-\sqrt{\mu_1})^2}{(\mu_n-\mu_1)}$$

$$\sqrt{\beta}=\frac{(\sqrt{\mu_n}-\sqrt{\mu_1})^2}{(\sqrt{\mu_n}-\sqrt{\mu_1})(\sqrt{\mu_n}+\sqrt{\mu_1})}$$

$$\sqrt{\beta} = \frac{(\sqrt{\mu_n} - \sqrt{\mu_1})}{(\sqrt{\mu_n} + \sqrt{\mu_1})}$$

$$\beta = \left[\frac{(\sqrt{\mu_n} - \sqrt{\mu_1})}{(\sqrt{\mu_n} + \sqrt{\mu_1})}\right]^2 . \tag{5.1.18}$$

Substituting the value of β in (5.1.16) we get,

$$\alpha = 2\left[\frac{1 + \left(\frac{\sqrt{\mu_n} - \sqrt{\mu_1}}{\sqrt{\mu_n} + \sqrt{\mu_1}}\right)^2}{\mu_1 + \mu_n}\right]$$

$$\alpha = 2\frac{\left[(\sqrt{\mu_n} + \sqrt{\mu_1})^2 + (\sqrt{\mu_n} - \sqrt{\mu_1})^2\right]}{(\sqrt{\mu_n} + \sqrt{\mu_1})^2 (\mu_1 + \mu_n)}$$

$$\alpha = 2\frac{\left[\mu_n + \mu_1 + 2\sqrt{\mu_n}\sqrt{\mu_1} + \mu_n + \mu_1 - 2\sqrt{\mu_n}\sqrt{\mu_1}\right]}{(\sqrt{\mu_n} + \sqrt{\mu_1})^2 (\mu_1 + \mu_n)}$$

$$\alpha = 2\frac{2(\mu_n + \mu_1)}{(\sqrt{\mu_n} + \sqrt{\mu_1})^2 (\mu_1 + \mu_n)}$$

$$\alpha = \left[\frac{2}{\sqrt{\mu_n} + \sqrt{\mu_1}}\right]^2 .$$

The α is calculated by using the relation,

$$\alpha = \left[\frac{2}{\sqrt{a} + \sqrt{b}}\right]^2 \tag{5.1.19}$$

and β is calculated using the relation

$$\beta = \left[\frac{\sqrt{a} - \sqrt{b}}{\sqrt{a} + \sqrt{b}}\right]^2 , \tag{5.1.20}$$

where "a" and "b" are the lower and upper bounds to the eigenvalue spectrum of A. The spectral radius of this iteration is $\sqrt{\beta}$ and the rate of convergence is $\frac{2}{\sqrt{P}}$ (see Hageman and Young [34], Wachspress [79]).

5.1.4 The Chebyshev acceleration of the second order Richardson iterative method

The Chebyshev acceleration of equation (5.1.12) has the form

$$\underline{x}^{(k+1)} = \underline{x}^{(k)} + \alpha_k(\underline{b} - A\underline{x}^{(k)}) + \beta_k(\underline{x}^{(k)} - \underline{x}^{(k-1)}) \tag{5.1.21}$$

with the parameters α_k and β_k varying with each iteration. If $\underline{x}$ is the exact solution then (5.1.21) can be written as

$$\underline{x} = \underline{x} + \alpha_k(\underline{b} - A\underline{x}) + \beta_k(\underline{x} - \underline{x}) \tag{5.1.22}$$

Subtracting (5.1.22) from (5.1.21) we get,

$$(\underline{x}^{(k+1)} - \underline{x}) = (\underline{x}^{(k)} - \underline{x}) - \alpha_k A(\underline{x}^{(k)} - \underline{x}) + \beta_k((\underline{x}^{(k)} - \underline{x}) - (\underline{x}^{(k-1)} - \underline{x})) \tag{5.1.23}$$

As $\underline{x}^{(k)} - \underline{x} = \underline{e}^{(k)}$ represents the error after the k^{th} iteration, equation (5.1.23) can be written as

$$\underline{e}^{(k+1)} = \underline{e}^{(k)} - \alpha_k A(\underline{e}^{(k)}) + \beta_k(\underline{e}^{(k)} - \underline{e}^{(k-1)}) \tag{5.1.24}$$

where $A = \frac{(b-a)z}{2} + \frac{(b+a)}{2}$ matching of A and "a" and "b" are the lower and upper bounds to the eigenvalue spectrum of A respectively adjusted to the interval $(-1, 1)$. z is defined as

$$z = \frac{b+a}{b-a}. \tag{5.1.25}$$

Now by the definition we have $\underline{e}^{(k)}$ can be substituted for $\frac{T_k(z)}{T_k(\frac{1}{\mu})}$ so equation (5.1.24) becomes,

$$\begin{aligned} \frac{T_k(z)}{T_k(\frac{1}{\mu})} &= \frac{T_{k-1}(z)}{T_{k-1}(\frac{1}{\mu})} - \alpha_k \left[\frac{(b-a)z}{2} \frac{T_{k-1}(z)}{T_{k-1}(\frac{1}{\mu})}\right] - \alpha_k \left[\frac{(b+a)}{2} \frac{T_{k-1}(z)}{T_{k-1}(\frac{1}{\mu})}\right] \\ &\quad + \beta_k \left[\frac{T_{k-1}(z)}{T_{k-1}(\frac{1}{\mu})} + \frac{T_{k-2}(z)}{T_{k-2}(\frac{1}{\mu})}\right]. \end{aligned} \tag{5.1.26}$$

Now to generate the Chebyshev polynomial, we form the recurrence relation which the Chebyshev polynomials must satisfy,

$$\frac{T_k(z)}{T_k(\frac{1}{\mu})} = \frac{2zT_{k-1}(z) - T_{k-2}(z)}{T_k(\frac{1}{\mu})} \tag{5.1.27}$$

equating the co-efficients of z on the right hand side of equations (5.1.26) and (5.1.27) we obtain,

$$\frac{2T_{k-1}(z)}{T_k(\frac{1}{\mu})} = -\alpha_k \left[\frac{(b-a)}{2}\frac{T_{k-1}(z)}{T_{k-1}(\frac{1}{\mu})}\right]$$

$$-\frac{(b-a)}{2}\alpha_k = \frac{2T_{k-1}(\frac{1}{\mu})}{T_k(\frac{1}{\mu})}$$

or

$$\alpha_k = \frac{-4}{(b-a)}\frac{T_{k-1}(\frac{1}{\mu})}{T_k(\frac{1}{\mu})}. \tag{5.1.28}$$

Similarly, we obtain

$$\beta_k = \frac{T_{k-2}(\frac{1}{\mu})}{T_k(\frac{1}{\mu})} \tag{5.1.29}$$

where

$$\mu = \frac{(b-a)}{2-(b+a)}. \tag{5.1.30}$$

The error reduction is given by the equation

$$\underline{e}^{(k+1)} = \left[\frac{T_{k+1}\left(\frac{b+a-2x}{b-a}\right)}{T_{k+1}\left(\frac{b+a}{b-a}\right)}\right]\underline{e}^{(0)} \tag{5.1.31}$$

and the average rate of convergence is given by

$$R = -\frac{1}{(k+1)}\log\left[T_{k+1}\left(\frac{b+a}{b-a}\right)\right]^{-1} \tag{5.1.32}$$

which reduces to (for proof see section (5.1.1)),

$$R = \frac{2}{\sqrt{P}}.$$

Since the coefficients α_k and β_k are less than unity, roundoff difficulties do not arise and hence this method is preferred.

The asymptotic rate of convergence of the second order nonstationary Richardson iterative method is $\frac{2}{\sqrt{P}}$ (P is the P-condition number given by $\mid \frac{b}{a} \mid$) (Hageman and Young [34], Westlake [80]). A matrix with large P-condition number is an ill conditioned matrix and causes great difficulties in solving the linear system. The greater the P-condition number of the matrix the smaller is the rate of convergence. An average value for the P-condition number is n. The maximum and minimum eigenvalues can be calculated using the Gersgorin and Collatz theorems which are given by

$$|\lambda|_{max} \leq \max \sum_{j=1}^{n} |a_{ij}| \text{ and} \tag{5.1.33}$$

$$|\lambda|_{min} \geq a_{ii} - \sum_{i \neq j}^{n} |a_{ij}| \text{ respectively.} \tag{5.1.34}$$

5.1.5 The Accelerated Overrelaxation (A.O.R.) iterative method

Now it can be shown from (Hadjidimos, [33]) that, the equation,

$$D\underline{x}^{(k+1)} = (1-\omega)D\underline{x}^{(k)} + \omega(\underline{b} - U\underline{x}^{(k)}) - (\omega - r)L\underline{x}^{(k)} - rL\underline{x}^{(k+1)}, \tag{5.1.35}$$

represents the Accelerated Overrelaxation iterative method (A.O.R.), where "ω" and "r" are the overrelaxation and acceleration factors, applied to the $U\underline{x}^{(k)}$ and $L\underline{x}^{(k+1)}$ respectively, (Martins, [53]). Here, equation (5.1.35) represents the forward ordering of the equations (5.1.1) and (5.1.2). Equation (5.1.35) can be written as,

$$\underline{x}^{(k+1)} = (1-\omega)\underline{x}^{(k)} + D^{-1}\left[\omega(\underline{b} - U\underline{x}^{(k)}) - (\omega - r)L\underline{x}^{(k)} - rL\underline{x}^{(k+1)}\right]$$

$$\begin{aligned} \underline{x}^{(k+1)} + rD^{-1}L\underline{x}^{(k+1)} &= (1-\omega)\underline{x}^{(k)} + \omega D^{-1}(\underline{b} - U\underline{x}^{(k)} - \omega L\underline{x}^{(k)}) + \\ & \quad rD^{-1}L\underline{x}^{(k)} \end{aligned}$$

$$\begin{aligned}(I + rD^{-1}L)\underline{x}^{(k+1)} &= \left[(1-\omega)I - D^{-1}(\omega U + \omega L - rL)\right]\underline{x}^{(k)} + \\ &\quad (\omega D^{-1}\underline{b})\end{aligned}$$

$$\begin{aligned}\underline{x}^{(k+1)} &= \left[I + rD^{-1}L\right]^{-1}\left[(1-\omega)I - D^{-1}(\omega U + \omega L - rL)\right]\underline{x}^{(k)} + \\ &\quad \left[I + rD^{-1}L\right]^{-1}(\omega D^{-1}\underline{b}) \qquad (5.1.36)\end{aligned}$$

The matrix $[I + rD^{-1}L]^{-1}[(1-\omega)I - D^{-1}(\omega U + \omega L - rL)]$ is called the A.O.R. iteration matrix.

The 2^{nd} and 3^{rd} iterations of the A.O.R. iterative method are as follows,

$$D\underline{x}^{(k+2)} = (1-\omega)D\underline{x}^{(k+1)} + \omega(\underline{b} - U\underline{x}^{(k+1)}) - (\omega - r)L\underline{x}^{(k+1)} - rL\underline{x}^{(k+2)}, \quad (5.1.37)$$

$$D\underline{x}^{(k+3)} = (1-\omega)D\underline{x}^{(k+2)} + \omega(\underline{b} - U\underline{x}^{(k+2)}) - (\omega - r)L\underline{x}^{(k+2)} - rL\underline{x}^{(k+3)}. \quad (5.1.38)$$

Observing equations (5.1.35), (5.1.37) and (5.1.38) it can be noticed that the factors $L\underline{x}^{(k+1)}$ and $L\underline{x}^{(k+2)}$ are common in equations (5.1.35), (5.1.37) and (5.1.37), (5.1.38) respectively. Hence these need be computed only once. This will reduce the computational work in the iterative solver as the factor $L\underline{x}^{(k)}$ is the only one which needs to be computed. A similar technique has been previously used for the S.S.O.R. iterative method by Conrad and Wallach [11] and, Evans and Haider [18]. Thus, extra hardware is only required to compute the $L\underline{x}^{(k)}$ factor for the 1^{st} iteration, because in the successive iterations the computation is already performed.

The optimum value of "r" and "ω" can be found by using the relations (see Hadjidimos [33])

$$r_1 = \frac{2\left(1+\sqrt{1-\mu^2}\right)}{\mu^2}, \quad r_2 = \frac{2}{1+\sqrt{1-\mu^2}};$$
$$\omega_1 = \frac{-1}{\sqrt{1-\mu^2}}, \quad \omega_2 = \frac{1}{\sqrt{1-\mu^2}},$$

where μ is the maximum eigenvalue obtained from the Jacobi iteration matrix.

5.1.6 The Symmetric A.O.R. iterative method

Equation (5.1.35) represents the forward ordering of the A.O.R. iterative method, similarly, using the backward ordering of the vector $x^{(k+1)}$, the A.O.R. iterative method can also be written as follows,

$$D\underline{x}^{(k+2)} = (1-\omega)D\underline{x}^{(k+1)}+\omega(\underline{b}-L\underline{x}^{(k+1)})-(\omega-r)U\underline{x}^{(k+1)}-rU\underline{x}^{(k+2)}. \tag{5.1.39}$$

Finally, if the iterations are performed alternately using the forward and backward orderings respectively then the method is known as the Symmetric Accelerated Overrelaxation (S.A.O.R.) iterative method.

The next iteration of the S.A.O.R. iterative method after equation (5.1.39) will be,

$$D\underline{x}^{(k+3)} = (1-\omega)D\underline{x}^{(k+2)}+\omega(\underline{b}-U\underline{x}^{(k+2)})-(\omega-r)L\underline{x}^{(k+2)}-rL\underline{x}^{(k+3)}. \tag{5.1.40}$$

Equation (5.1.39) can be written in the form,

$$\begin{aligned}\underline{x}^{(k+2)} &= \left[I+rD^{-1}U\right]^{-1}\left[(I-\omega)I-D^{-1}(\omega L+\omega U-rU)\right]\underline{x}^{(k+1)}+\\ &\quad \left[I+rD^{-1}U\right]^{-1}(\omega D^{-1}\underline{b})\end{aligned} \tag{5.1.41}$$

The iteration matrix for the 1^{st} half is as that of the A.O.R. iterative method and the iteration matrix for the 2^{nd} half is given by, $[I+rD^{-1}U\ ]^{-1}[(1-\omega)I-D^{-1}(\omega L+\omega U-rU)]$. A single sweep of the S.A.O.R. iterative method comprises of a forward and a backward iteration. This means that $\underline{x}^{(k+1)}$ and $\underline{x}^{(k)}$ are the 1^{st} and 2^{nd} half of the S.A.O.R. iterative method, and can be written as,

$$\begin{aligned}\underline{x}^{(k+\frac{1}{2})} &= \left[I+rD^{-1}L\right]^{-1}\left[(1-\omega)I-D^{-1}(\omega U+\omega L-rL)\right]\underline{x}^{(k)}+\\ &\quad \left[I+rD^{-1}L\right]^{-1}(\omega D^{-1}\underline{b})\end{aligned} \tag{5.1.42}$$

$$\begin{aligned}\underline{x}^{(k+1)} &= \left[I+rD^{-1}U\right]^{-1}\left[(1-\omega)I-D^{-1}(\omega L+\omega U-rU)\right]\underline{x}^{(k+\frac{1}{2})}+\\ &\quad \left[I+rD^{-1}U\right]^{-1}(\omega D^{-1}\underline{b})\end{aligned} \tag{5.1.43}$$

Eliminating $\underline{x}^{(k+\frac{1}{2})}$ in equation (5.1.43) we get,

$$\begin{aligned}\underline{x}^{(k+1)} &= \left[I+rD^{-1}U\right]^{-1}\left[(1-\omega)I-D^{-1}(\omega L+\omega U-rU)\right]\\ &\quad \left[I+rD^{-1}L\right]^{-1}\left[(1-\omega)I-D^{-1}(\omega U+\omega L-rL)\right]\underline{x}^{(k)}+\\ &\quad \left[I+rD^{-1}U\right]^{-1}\left[(1-\omega)I-D^{-1}(\omega L+\omega U-rU)\right]\\ &\quad \left\{\left[I+rD^{-1}L\right]^{-1}(\omega D^{-1}\underline{b})\right\}+\left[I+rD^{-1}U\right]^{-1}(\omega D^{-1}\underline{b})\end{aligned}$$

or,

$$\underline{x}^{(k+1)} = K\underline{x}^{(k)}+\underline{g}$$

and $\underline{g}$ and K are evaluated as follows,

$$\begin{aligned}\underline{g} &= \left[I+rD^{-1}U\right]^{-1}\left[I+rD^{-1}L\right]^{-1}\\ &\quad \left[(1-\omega)I-D^{-1}(\omega L+\omega U-rU-rL)+I\right](\omega D^{-1}\underline{b})\\ &= \left[I+rD^{-1}U\right]^{-1}\left[I+rD^{-1}L\right]^{-1}\\ &\quad \left[(2-\omega)I-D^{-1}\left\{\omega(L+U)-r(L+U)\right\}\right](\omega D^{-1}\underline{b})\\ &= \omega\left[I+rD^{-1}U\right]^{-1}\left[I+rD^{-1}L\right]^{-1}\\ &\quad \left[(2-\omega)I+D^{-1}(r-\omega)\left[L+U\right]\right](D^{-1}\underline{b}).\end{aligned}$$

$$\begin{aligned}K &= \left[I+rD^{-1}U\right]^{-1}\left[I+rD^{-1}L\right]^{-1}\\ &\quad \left[(1-\omega)I-D^{-1}(\omega L+\omega U-rU)\right]\\ &\quad \left[(1-\omega)I-D^{-1}(\omega U+\omega L-rL)\right].\end{aligned}$$

The matrix K represents the S.A.O.R. iteration matrix. When $r=\omega$ the S.S.O.R. iterative method is obtained.

5.1.7 The Unsymmetric A.O.R. iterative method

If two overrelaxation and acceleration parameters ω, r and $\hat{\omega}$ and $\hat{r}$ are cyclically used alternatively in the successive iterations of the S.A.O.R. method, the new method obtained is known as Unsymmetric Accelerated Overrelaxation iterative method (U.S.A.O.R). If equation (5.1.39) is changed to,

$$D\underline{x}^{(k+2)} = (1-\hat{\omega})D\underline{x}^{(k+1)}+\hat{\omega}(\underline{b}-L\underline{x}^{(k+1)})-(\hat{\omega}-\hat{r})U\underline{x}^{(k+1)}-\hat{r}U\underline{x}^{(k+2)}. \tag{5.1.44}$$

Then the equations (5.1.35), (5.1.44) and (5.1.40) represent the 1^{st} three iterations of the U.S.A.O.R. iterative method.

5.2 Systolic designs

5.2.1 Systolic design for the second order Richardson iterative method with Chebyshev acceleration

The design can simulate a dense ($n \times n$) system of linear equations. The user needs to define the order of the system at compile time. The "a", "b", "α_0" and "β_0" are precomputed in the host computer.

Equation (5.1.21) can be decomposed into the following sub-computations

$$\underline{y}^{(k)} = A\underline{x}^{(k)} \tag{5.2.1}$$

$$\underline{z}^{(k)} = \alpha_k(\underline{b} - \underline{y}^{(k)}) \tag{5.2.2}$$

$$\underline{r}^{(k)} = \beta_k(\underline{x}^{(k)} - \underline{x}^{(k-1)}) \tag{5.2.3}$$

$$\underline{x}^{(k+1)} = \underline{x}^{(k)} + \underline{r}^{(k)} + \underline{z}^{(k)} \tag{5.2.4}$$

which comprises of matrix-vector and constant vector multiplications together with vector addition and subtraction operations. This decomposition suggests that for a system of order n, $(n + 2)$ processing cells are required to form the array to solve the system. To compute the product $A\underline{x}$ represented by equation (5.2.1) n cells are required. One cell is required to compute equation (5.2.2). The remaining cell is the boundary cell which computes the equations

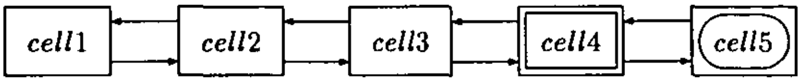


Figure 5.1: Linear systolic array for the second order Richardson iterative method.

(5.2.3) and (5.2.4). The equation (5.2.3) is computed in the idle cycles of this cell, because the computation is not required "on the fly". The new "α" and "β" are also computed in the idle cycle by the boundary cell, expressed by equation (5.1.28) and (5.1.29) respectively. The old value of α is kept in the boundary cell until the new α is computed for the next iteration. The old α is sent to cell4 (in our example) and simultaneously the new α is computed in the boundary cell. This is done because the α cannot be updated in the cell4 until the last component of $\underline{z}_1^0$, i.e. equation (5.2.2) has been calculated. So an extra register or memory location is required in the boundary cell (cell5). This process speeds up the iteration by saving the extra cycles required for calculating the α and β for the next iterations.

The system configured for a (3 × 3) system of linear equations is described as follows. The system consists of 5 cells as shown in figure (5.1). cell1, cell2, and cell3 are the basic inner product step (IPS) cells with cell1 having a slightly different I/O architecture. The cell1 does not need to send and receive any data on the channels x_{out} and y_{in}. This eliminates the need of a source and sink provided by the array hardware or the host computer and saves both area and time requirements. Cell4 performs the operation defined by equation (5.2.2) and cell5 does the job expressed by the equations (5.2.3), (5.2.4), (5.1.28) and (5.1.29). Figure (5.2) shows the structure of the cells and the operations performed. Figure (5.3) shows the snap shots for each iteration of the nonstationary second order Richardson iterative method. The array takes 12 time steps to complete the each iteration.

The cells are initialised with the required data. The cells (cell3, cell2, and

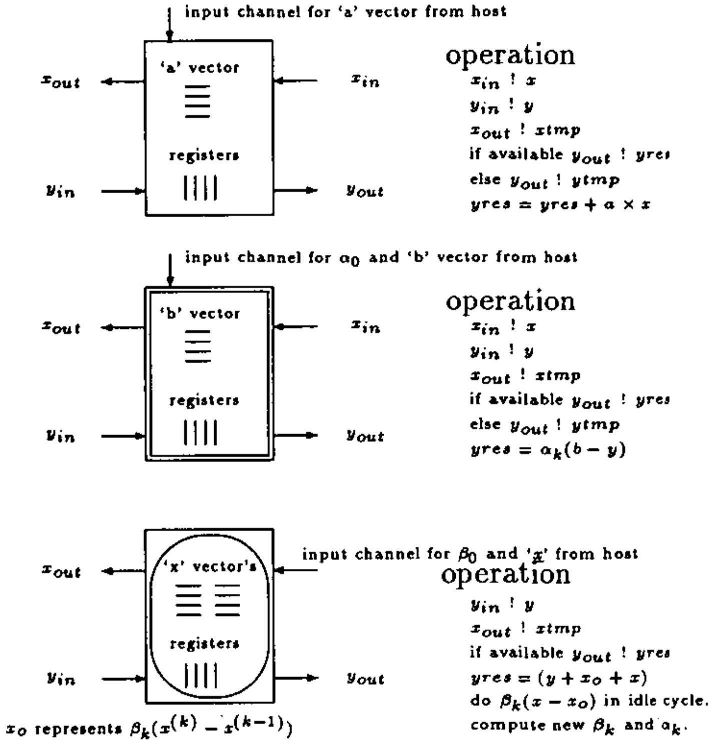


Figure 5.2: Cell operations for the nonstationary second order Richardson iterative method.

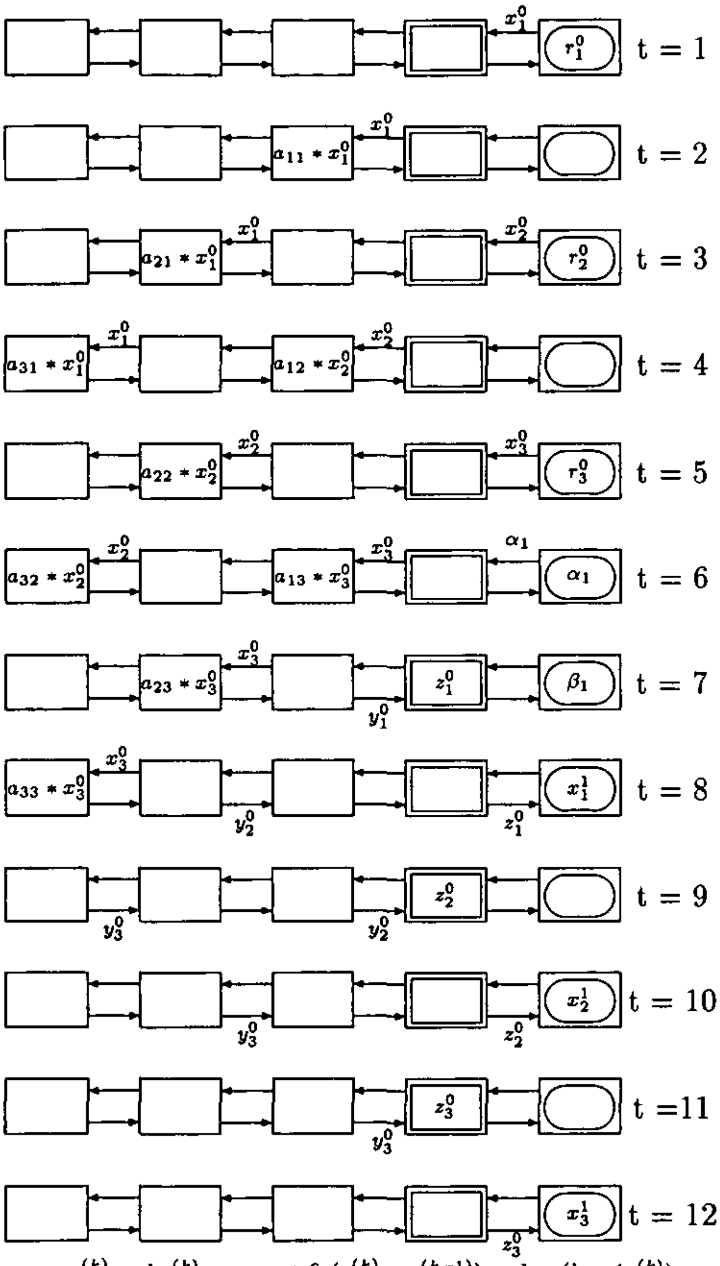
As in the array there are different operations going on and the array needs to be synchronised, so that the global timing of the system is preserved. If a slow memory is used then due to fetch/store operations the systems global timing needs to be adjusted, which can degrade the performance of the array considerably. If a high speed memory is available then this problem might not be of great importance.

The design could have been designed to process the data as soon as the data enters from the host computer to the array but it needs rather complex cell structures and synchronisation which slows down the first iteration. Already cell5 is quite complex due to its control and operational requirements. The array starts iterating and terminates the process when one of the following conditions occur.

- The solution is found, i.e. the system has converged.
- The number of iteration performed equals the maximum allowed.

The array interrupts the host computer to receive the solution or the error message, which is then displayed on the host screen. An upper limit on the number of iterations to be performed is implemented so that the system escapes from the infinite loop, in the case, the system does not converge. When an element of $\underline{x}^{(k+1)}$ is computed then the convergence test is performed for the element computed. The convergence test is performed by comparing $\underline{x}^{(k+1)}$ and $\underline{x}^{(k-1)}$ elements as they are produced. This was done because while comparing the elements of $\underline{x}^{(k+1)}$ and $\underline{x}^{(k)}$ the results were almost the same and the process was terminated "False termination". The solution obtained was close but not correct. The array processing can be terminated in one of the following ways,

- After convergence has been achieved a reset signal can be broadcast to reset the array. However in the systolic designs broadcasting of signals is avoided so this strategy is not implemented.



$\underline{r}^{(k)}$ and $\underline{z}^{(k)}$ represent $\beta_k(\underline{x}^{(k)} - \underline{x}^{(k-1)})$ and $\alpha_k(\underline{b} - A\underline{x}^{(k)})$ respectively.

Figure 5.3: Snap shots for the nonstationary second order Richardson iterative method.

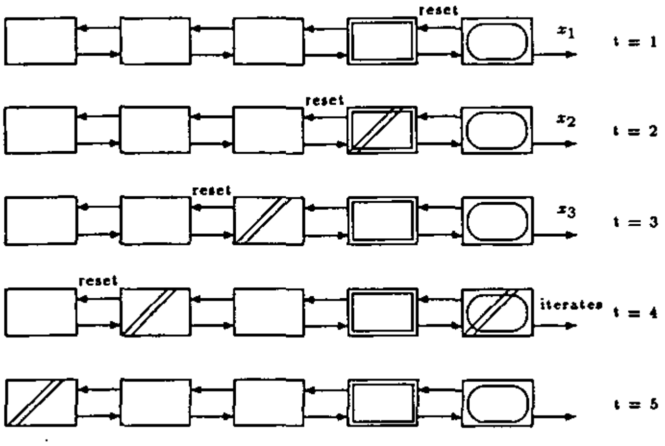


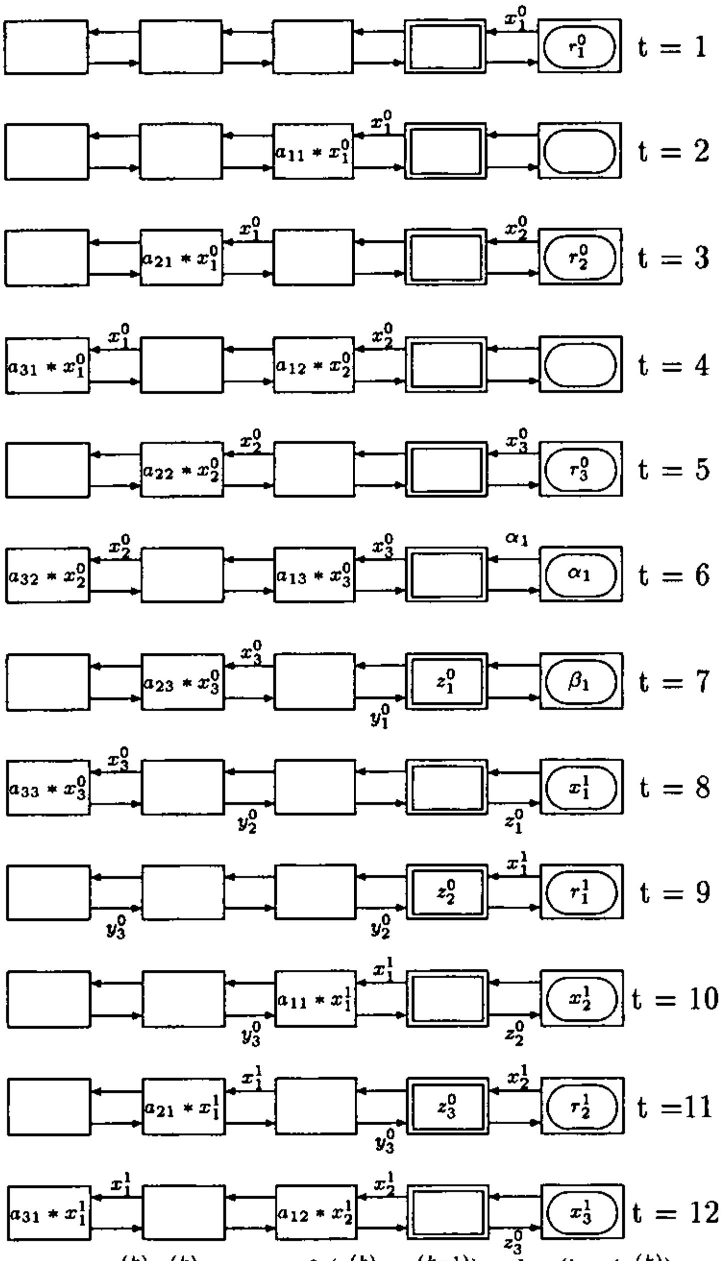
Figure 5.4: Termination process for a (3 × 3) system.

- Reset the array automatically before the convergence test is made (i.e. the iteration is complete.) But in our design two iterations are being performed simultaneously in the array, so implementing this strategy will be a disaster.
- If the system has converged the pipeline can be flushed by a reset signal generated by the boundary cell. This has been implemented to reset the array gracefully.

It takes $(n + 2)$ cycles to reset the array. The termination process is shown in figure (5.4). The cells marked with double lines terminate after sending the reset signal to the neighbouring cell. The boundary cell, i.e. cell5, terminates after sending all the data to the host computer including the iteration count.

5.2.2 Design improvements

The systolic design presented by figure (5.3) processes one iteration at a time. If we are ready to pay for more complex control logic and utilise the idle cycles of the cell5 we can modify the design in such a way that two iterations can be partially processed. As soon as boundary cell has evaluated the first element of new solution vector 2^{nd} iteration can be started in the 1^{st} iteration at time step 9. At time step 12, the second iteration has progressed one third (see figure (5.5)). Now we start the second iteration cycle. At time step 2, the new α is calculated and the previous α is sent to the cell4. The new β is computed in the time step 3, (see figure (5.6)). The second and successive iterations take $(2n + 2)$ time steps, this is quite an improvement over the normal regular systolic design which takes $(4n)$ time steps to complete an iteration. The computation of α and β in the idle cycles, saves two time steps per iteration. The pipeline design saves 4 time steps per iteration after the first iteration, in case of our example. Thus the hardware is utilised more efficiently. The other improvement can be achieved by using some other technique to terminate the iterating process. The convergence test we have implemented takes a substantial amount of computing. If the approximate number of iteration required could be predicted and the system is allowed to perform the upper bound of the approximation then the overhead of convergence can be eliminated. This means that the approximation should be very close to the exact number, other wise the overhead due to extra iterations performed would over come the time required to perform the convergence test.



$\underline{r}^{(k)}, \underline{z}^{(k)}$ represent $\beta_k(\underline{x}^{(k)} - \underline{x}^{(k-1)})$ and $\alpha_k(\underline{b} - A\underline{x}^{(k)})$ respectively.

Figure 5.5: Snap shots for the pipelined design of the 1^{st} iteration of the nonstationary second order Richardson iterative method.

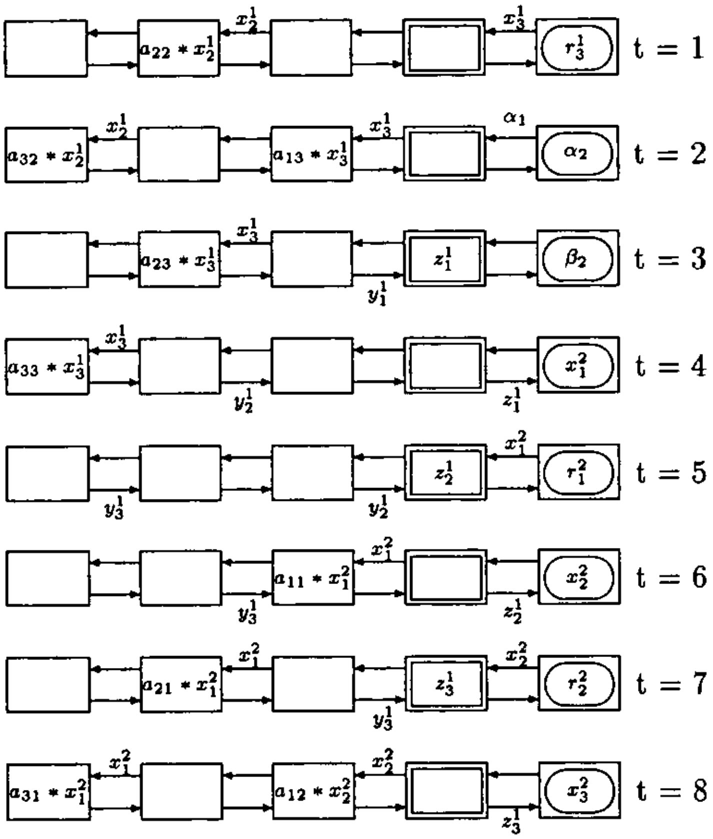


Figure 5.6: Snap shots for the pipelined design of the 2^{nd} iteration of the nonstationary second order Richardson iterative method.

5.2.3 Systolic design for the stationary second order Richardson iterative method

The systolic design for the second order Richardson iterative method (see Ha[illegible] and Evans [35]) is exactly the same as the systolic design for the nonstationary second order Richardson iterative method. The parameters "α" and "β" remain constant. This does not affect the design as these parameters are calculated in the idle cycles for the nonstationary second order Richardson iterative method. The area and time requirements for the stationary and nonstationary second order Richardson iterative method with and without improved design are given in table (5.1) for comparison, where "k" is the number of iterations performed and "n" is the size of the system.

5.2.4 Systolic design for the A.O.R. iterative method

The systolic design for the A.O.R. iterative method for a $(n*n)$ banded linear system, with bandwidth 5, is simulated soft systolically in OCCAM as follows. The example discussed is for a $(4*4)$ linear system with $p = q = 3$.

The systolic design for the 1^{st} iteration of the A.O.R. iterative method is shown in figure (5.7). Figure (5.8) shows the systolic design for the successive iterations of the A.O.R. iterative method. Figures (5.9) and (5.10) show the snap

Method	Architecture	Time	Area
Richardson	Normal	$k(4n)$	$n+2$
Richardson	Pipelined	$(2n-2)+(k-1)\times(2n+2)$	$n+2$
Chebyshev	Normal	$k(4n)$	$n+2$
Chebyshev	Pipelined	$(2n-2)+(k-1)\times(2n+2)$	$n+2$

Table 5.1: Area and time requirements for the stationary and nonstationary second order Richardson iterative method.

Iterative method	Overrelaxation factor	Acceleration factor
Jacobi	1	0
Gauss-Seidel	1	1
S.O.R.	ω	ω
A.O.R.	ω	r

Table 5.2: Combination of ω and r to yield different iterative methods.

shots for the 1^{st} and successive iterations respectively. It takes "$w + 2n - 2$" time units to compute the 1^{st} and the successive iterations. However, speedup is not obtained as the $U\underline{x}$ factor is to be computed in all the iterations.

Consider figures (5.7) and (5.9). The $L\underline{x}^{(k+1)}$ factor need not be computed in the 2^{nd} iteration because it is precomputed in the 1^{st} iteration. This reduces the computational work but not the time, as this factor is computed in parallel with the $L\underline{x}^{(k+2)}$. However if the $U\underline{x}$ factor was precomputed in the most previous iteration then both the computational work as well as the time to complete the iteration would be reduced. Thus it can be proved that the amount of work of the S.A.O.R. iterative method is equivalent to that of the A.O.R. iterative method. The $U\underline{x}$ factor dominates the computational work, as the lower factor $L\underline{x}$ cannot be started until the 1^{st} element of the $U\underline{x}$ factor has been computed. This can be seen from the figures (5.9) and (5.10) and also by expanding equations (5.1.35) and (5.1.37).

The different combination of "r" and "ω" yield different iterative methods. Table (5.2) represents this fact.

To realise the linear systolic array implementing the A.O.R. algorithm consider equation (5.1.35). Equation (5.1.35) can be decomposed into the following sub-computations.

$$\underline{y}_i^2 = L\underline{x}^{(k)} \tag{5.2.5}$$

$$\underline{y}_i^1 = L\underline{x}^{(k+1)} \tag{5.2.6}$$

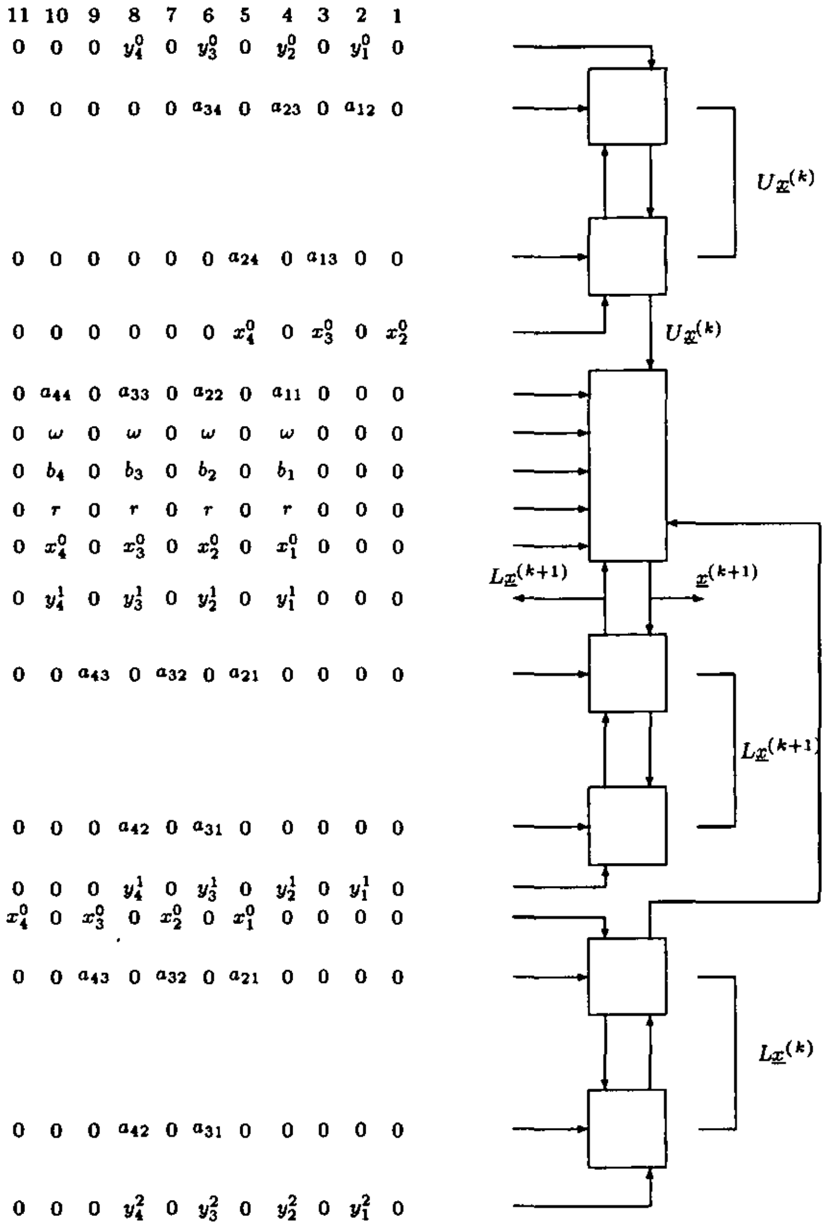
$$\underline{y}_i^0 = U\underline{x}^{(k)} \tag{5.2.7}$$

$$\underline{x}_i^{(k+1)} = (1-\omega)\ \ \underline{x}_i^{(k)} + (\omega(\underline{b}_i - y_i^0) - (\omega - r)y_i^2 - ry_i^1)/a_{ii} \tag{5.2.8}$$

This decomposition suggests that the linear systolic array consists of four hardware parts each performing the sub-computations (5.2.5), (5.2.6), (5.2.7) and (5.2.8).

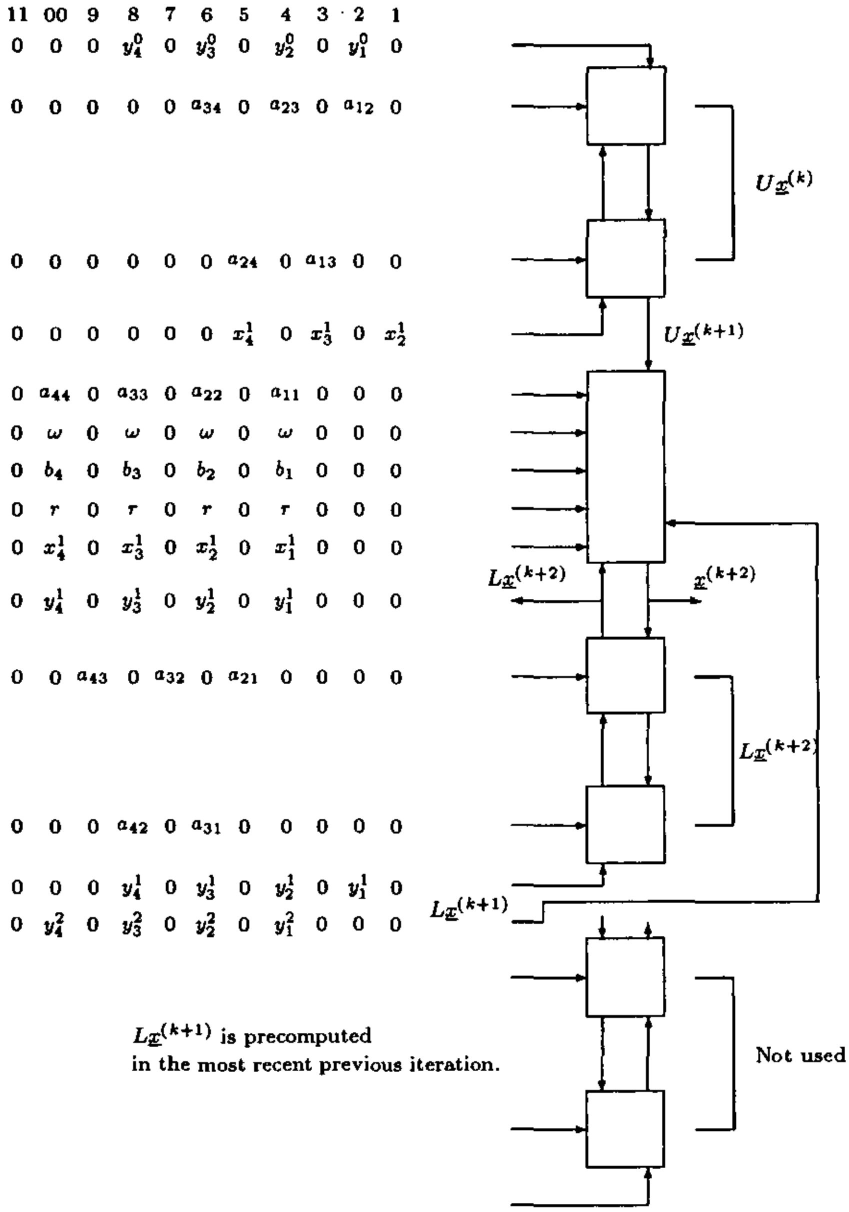
To compute x_2^1, the following computations are performed by the respective parts. $y_2^2 = a_{21}x_1^0$, $y_2^0 = a_{23}x_3^0 + a_{24}x_4^0$, $y_2^1 = a_{21}x_1^1$ and $x_2^1 = (1-\omega)\ \ \underline{x}_2^0 + (\omega(\underline{b}_2 - y_2^0) - (\omega - r)y_2^2 - ry_2^1)/a_{22}$. In figure (5.9), assume that the cells are numbered from 1 to 7, starting from left to right. Cells 1 and 2 compute y_2^1, cells 4 and 5 compute y_2^0, cells 6 and 7 compute y_2^2 and cell 3 computes x_2^1.

At time step 4, y_2^1 enters cell1and y_2^2 enters cell 6 and are passed unaltered, y_2^0 enters cell 5 and gets the partial result $y_2^0 = a_{23}x_3^0$. Cell 3 computes x_1^1. At time step 5, y_2^1 enters cell 2 and $y_2^1 = a_{21}x_1^1$ is computed. y_2^2 enters cell 7 and $y_2^2 = a_{21}x_1^0$ is computed. y_2^0 enters the cell 4 and $y_2^0 = a_{23}x_3^0 + a_{24}x_4^0$. Now the sub-computations y_2^0, y_2^1 and y_2^2 have been computed. At time step 6, cell 3 gets y_2^0, y_2^1, y_2^2, x_2^0, b_2, a_{22}, ω and r and x_2^1 is computed. The y_2^1 sub-computation is saved and reused as y_2^2 sub-computation in the successive iteration. This means that the sub computations y_i^2 are not performed in the successive iterations.



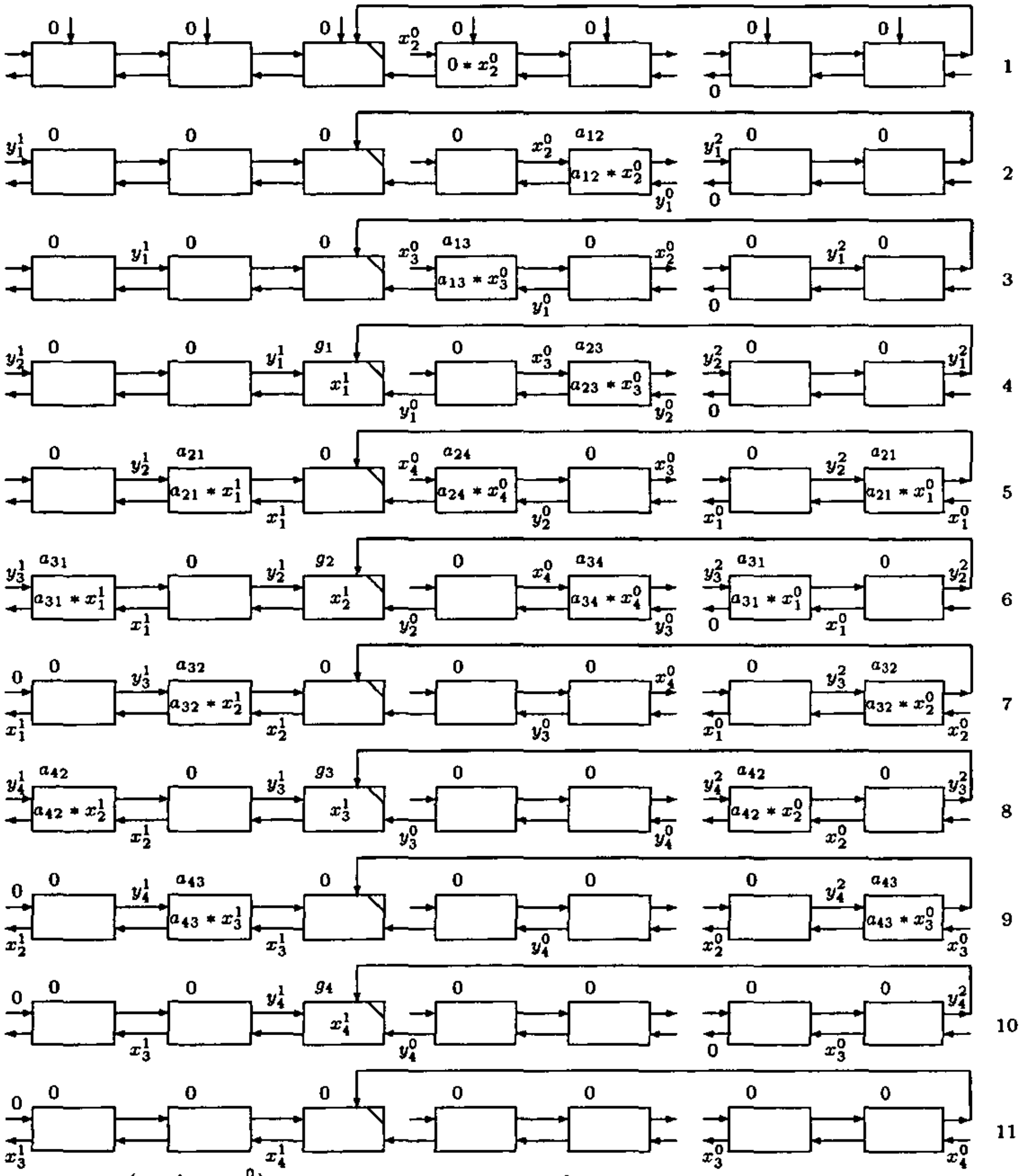
11 10 9 8 7 6 5 4 3 2 1
0 0 0 y_4^0 0 y_3^0 0 y_2^0 0 y_1^0 0
0 0 0 0 0 a_{34} 0 a_{23} 0 a_{12} 0
0 0 0 0 0 0 a_{24} 0 a_{13} 0 0
0 0 0 0 0 0 x_4^0 0 x_3^0 0 x_2^0
0 a_{44} 0 a_{33} 0 a_{22} 0 a_{11} 0 0 0
0 ω 0 ω 0 ω 0 ω 0 0 0
0 b_4 0 b_3 0 b_2 0 b_1 0 0 0
0 r 0 r 0 r 0 r 0 0 0
0 x_4^0 0 x_3^0 0 x_2^0 0 x_1^0 0 0 0
0 y_4^1 0 y_3^1 0 y_2^1 0 y_1^1 0 0 0
0 0 a_{43} 0 a_{32} 0 a_{21} 0 0 0 0
0 0 0 a_{42} 0 a_{31} 0 0 0 0 0
0 0 0 y_4^1 0 y_3^1 0 y_2^1 0 y_1^1 0
x_4^0 0 x_3^0 0 x_2^0 0 x_1^0 0 0 0 0
0 0 a_{43} 0 a_{32} 0 a_{21} 0 0 0 0
0 0 0 a_{42} 0 a_{31} 0 0 0 0 0
0 0 0 y_4^2 0 y_3^2 0 y_2^2 0 y_1^2 0
$U\underline{x}^{(k)}$
$U\underline{x}^{(k)}$
$L\underline{x}^{(k+1)}$
$\underline{x}^{(k+1)}$
$L\underline{x}^{(k+1)}$
$L\underline{x}^{(k)}$

Figure 5.7: Systolic design for the 1st iteration of the A.O.R. iterative method.


11 00 9 8 7 6 5 4 3 2 1
0 0 0 y_4^0 0 y_3^0 0 y_2^0 0 y_1^0 0
0 0 0 0 0 a_{34} 0 a_{23} 0 a_{12} 0
0 0 0 0 0 0 a_{24} 0 a_{13} 0 0
0 0 0 0 0 0 x_4^1 0 x_3^1 0 x_2^1
0 a_{44} 0 a_{33} 0 a_{22} 0 a_{11} 0 0 0
0 ω 0 ω 0 ω 0 ω 0 0 0
0 b_4 0 b_3 0 b_2 0 b_1 0 0 0
0 r 0 r 0 r 0 r 0 0 0
0 x_4^1 0 x_3^1 0 x_2^1 0 x_1^1 0 0 0
0 y_4^1 0 y_3^1 0 y_2^1 0 y_1^1 0 0 0
0 0 a_{43} 0 a_{32} 0 a_{21} 0 0 0 0
0 0 0 a_{42} 0 a_{31} 0 0 0 0 0
0 0 0 y_4^1 0 y_3^1 0 y_2^1 0 y_1^1 0
0 y_4^2 0 y_3^2 0 y_2^2 0 y_1^2 0 0 0
$U\underline{x}^{(k)}$
$U\underline{x}^{(k+1)}$
$L\underline{x}^{(k+2)}$
$\underline{x}^{(k+2)}$
$L\underline{x}^{(k+2)}$
$L\underline{x}^{(k+1)}$
Not used

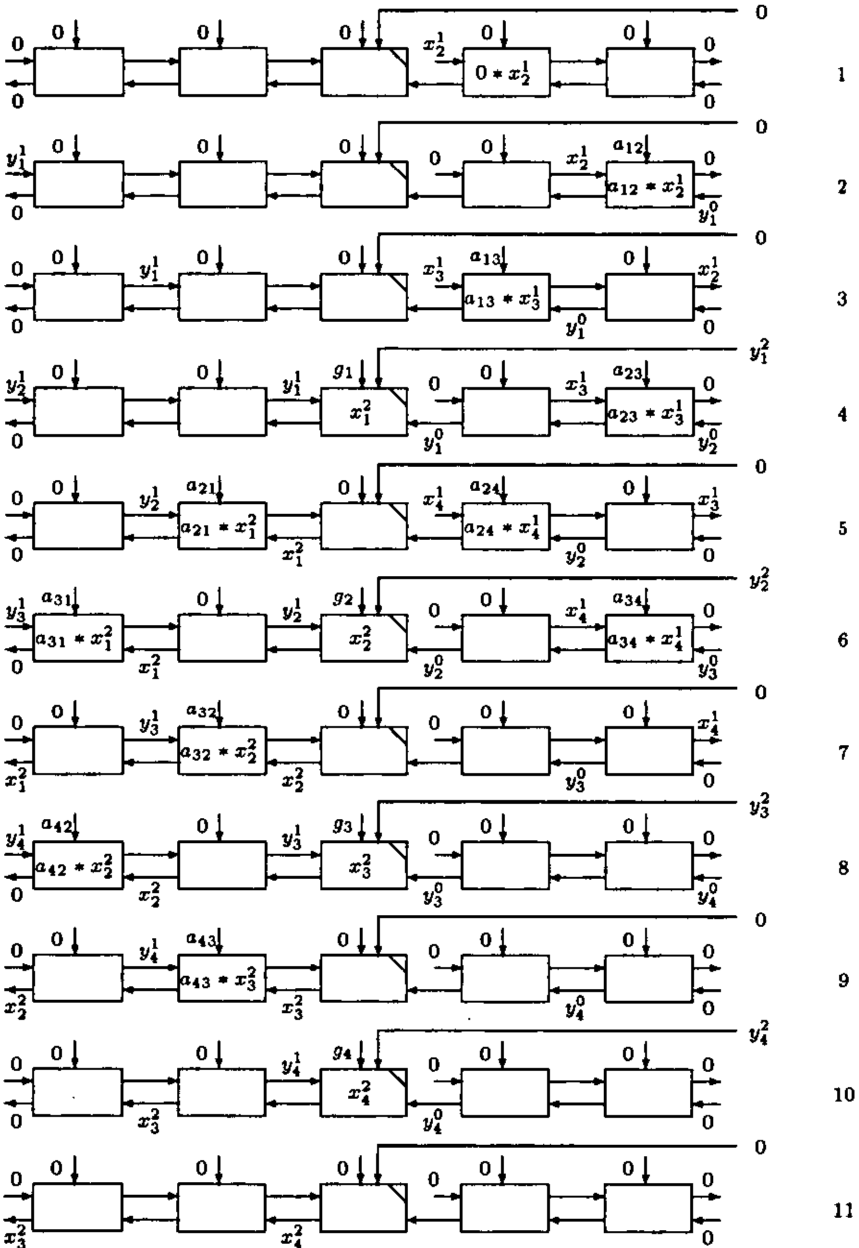
$L\underline{x}^{(k+1)}$ is precomputed
in the most recent previous iteration.

Figure 5.8: Systolic design for the 2^{nd} iteration of the A.O.R. iterative method.



$$\left.\begin{array}{l} g_i = (\omega, r, b_i, a_{ii}, x_i^0) \\ x_i^1 = (((\omega(b_i - y_i^0) - (\omega - r)y_i^2 - r y_i^1)/a_{ii}) + (1-\omega)x_i^0) \end{array}\right\} i = 1, 2, \ldots, n.$$

Figure 5.9: Snap shots for the 1^{st} iteration of the A.O.R. iterative method.



$$\left.\begin{array}{l} g_i = (\omega, r, b_i, a_{ii}, x_i^1) \\ x_i^1 = (((\omega(b_i - y_i^0) - (\omega - r)y_i^2 - ry_i^1)/a_{ii}) + (1-\omega)x_i^1) \end{array}\right\} \quad i = 1, 2, \ldots, n.$$

Figure 5.10: Snap shots for the 2^{nd} iteration of the A.O.R. iterative method.

5.2.5 Systolic design for the S.A.O.R. iterative method

Equations (5.1.35) and (5.1.39) represent the forward and backward ordering for the A.O.R. iterative method respectively. If the iterations are performed alternately using forward and backward ordering then as has been shown earlier the method Symmetric Accelerated Overrelaxation (S.A.O.R.) follows. In the S.A.O.R. systolic design it can be noticed that,

1. The terms $L\underline{x}^{(k+1)}$, $L\underline{x}^{(k+2)}$ as computed in the previous iterations for the A.O.R. iterative method are no longer common in the S.A.O.R. iterative method. As explained for the A.O.R. iterative method these factors do not decrease the iteration time but the computational work. In the S.A.O.R. iterative method these factors are not computed in the most recent previous iterations. Thus the computational work is increased without increasing the time to compute the iteration.

2. Observing equations (5.1.35), (5.1.39) and (5.1.40), the factors $L\underline{x}^{(k+1)}$, $U\underline{x}^{(k+2)}$ are common in equations (5.1.35) and (5.1.39), (5.1.39) and (5.1.40) and hence need be computed only once. This is known as the Conrad-Wallach technique. One fact to note here is that $L\underline{x}^{(k)}$, $U\underline{x}^{(k+1)}$,... are always new and hence are computed for every iteration. This requires extra hardware but as these factors are computed along with the $L\underline{x}^{(k+1)}$, $U\underline{x}^{(k+2)}$,... no extra delay or time is required and so the computation time is the same as the S.S.O.R. (see Evans and Haider[18]) iterative method.

Figures (5.7) and (5.12) represent the systolic design and figures (5.9) and (5.13) represent the snap shots for 1^{st} and successive iterations of the S.A.O.R. iterative method respectively. Table (5.3) gives the information of hardware area requirements and computation times.

Note that the area requirements for the S.A.O.R. iterative method is $(p-1)$ IPS cells more than the S.S.O.R. (see Evans and Haider [18]) iterative method, but

Method	Area (IPS+ DIV-ADD-IPS)	Time
S.A.O.R.	$\frac{3w-3}{2}+1$	$(w+2n-2)k$
S.A.O.R. (Conrad Wallach)	same as S.A.O.R.	$(w+2n-2)+(w+2n-5)\times(k-1)$

Table 5.3: Area and time requirements for the S.A.O.R. iterative method.

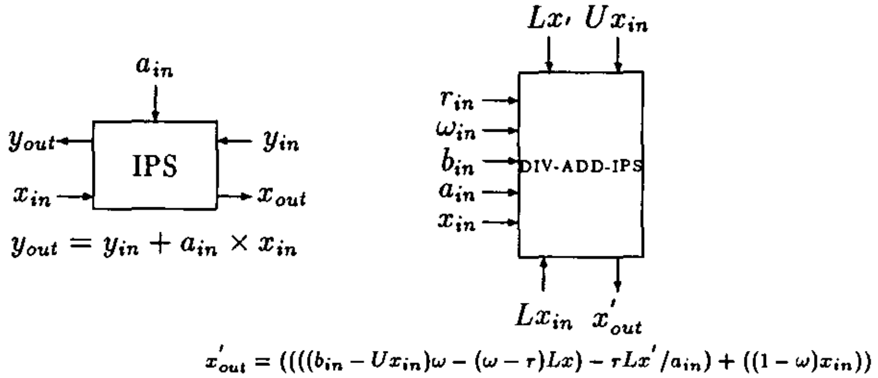

a_{in}
y_{out}
IPS
y_{in}
x_{in}
x_{out}
$y_{out} = y_{in} + a_{in} \times x_{in}$
Lx' Ux_{in}
r_{in}
ω_{in}
b_{in}
DIV-ADD-IPS
a_{in}
x_{in}
Lx_{in} x'_{out}
$x'_{out} = ((((b_{in} - Ux_{in})\omega - (\omega - r)Lx) - rLx'/a_{in}) + ((1-\omega)x_{in}))$

Figure 5.11: Cells for the A.O.R. and S.A.O.R. iterative methods.

the computation time is the same for the 1^{st} and successive iterations respectively, as explained earlier. The working of the 2^{nd} and successive iterations is similar to that of the S.S.O.R. iterative method. The DIV ADD IPS and IPS cell design is shown in figure (5.11).

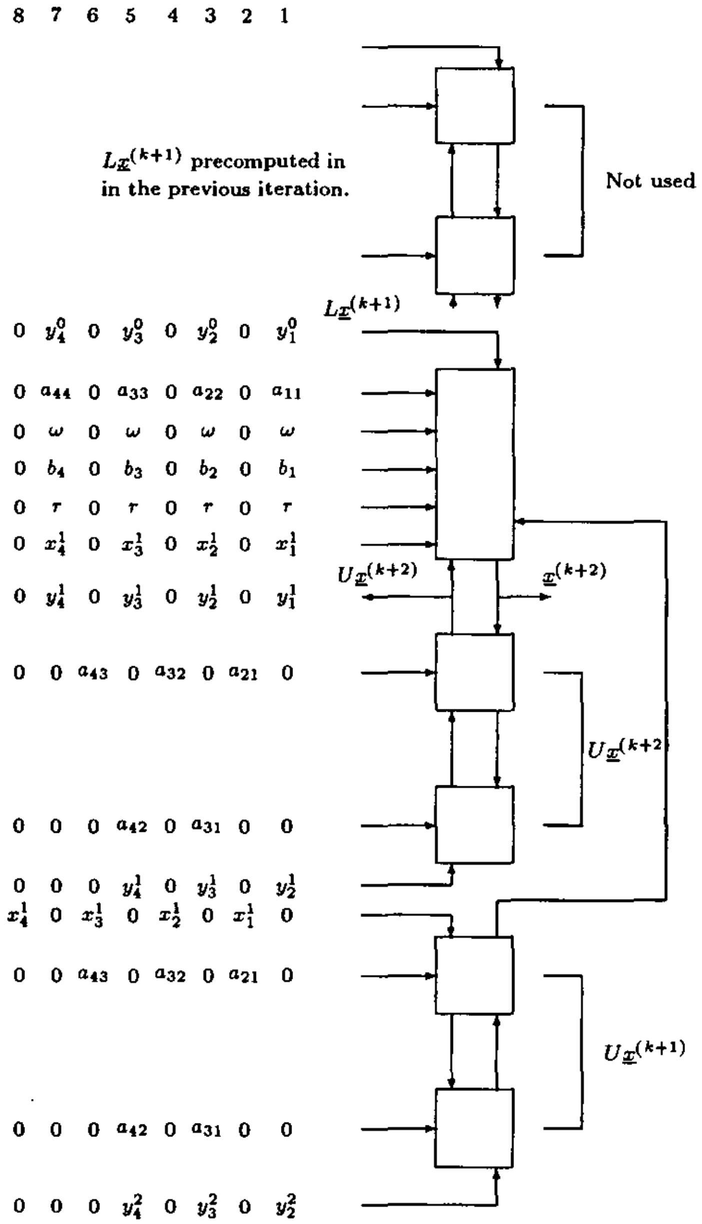
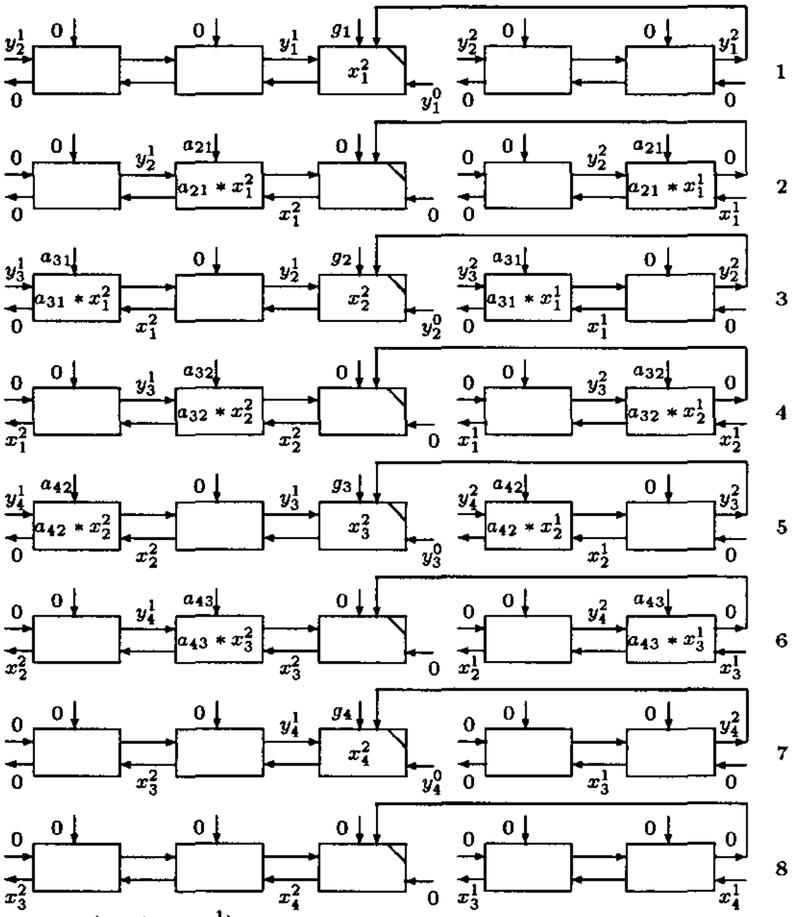

8 7 6 5 4 3 2 1
$L\underline{x}^{(k+1)}$ precomputed in in the previous iteration.
Not used
$L\underline{x}^{(k+1)}$
0 y_4^0 0 y_3^0 0 y_2^0 0 y_1^0
0 a_{44} 0 a_{33} 0 a_{22} 0 a_{11}
0 ω 0 ω 0 ω 0 ω
0 b_4 0 b_3 0 b_2 0 b_1
0 r 0 r 0 r 0 r
0 x_4^1 0 x_3^1 0 x_2^1 0 x_1^1
$U\underline{x}^{(k+2)}$
$\underline{x}^{(k+2)}$
0 y_4^1 0 y_3^1 0 y_2^1 0 y_1^1
0 0 a_{43} 0 a_{32} 0 a_{21} 0
$U\underline{x}^{(k+2}$
0 0 0 a_{42} 0 a_{31} 0 0
0 0 0 y_4^1 0 y_3^1 0 y_2^1
x_4^1 0 x_3^1 0 x_2^1 0 x_1^1 0
0 0 a_{43} 0 a_{32} 0 a_{21} 0
$U\underline{x}^{(k+1)}$
0 0 0 a_{42} 0 a_{31} 0 0
0 0 0 y_4^2 0 y_3^2 0 y_2^2

Figure 5.12: Systolic design for the 2^{nd} iteration of the S.A.O.R. iterative method.



$$g_i = (\omega, r, b_i, a_{ii}, x_i^1)$$
$$x_i^2 = (((\omega(b_i - y_i^0) - (\omega - r)y_i^2 - ry_i^1)/a_{ii}) + (1-\omega)x_i^1)$$
$\Big\}$ $i = 1, 2, \ldots, n.$

Note: y_i^0 contains y_i^1 computed in the most recent previous iteration.

Figure 5.13: Snap shots for the 2^{nd} iteration of the S.A.O.R. iterative method.

5.2.6 Systolic design for the U.S.A.O.R. iterative method

The systolic design for the U.S.A.O.R iterative method is the same for the 1^{st} iteration as shown in figures (5.7) and (5.9) but for figures (5.12) and (5.13) ω and r are replaced by $\hat{\omega}$ and $\hat{r}$. The area and time requirements remain the same. However the host computer does a little extra work to take care of the two overrelaxation and acceleration factors.

5.3 Conclusions

The stationary and nonstationary Richardson methods take asymptotically the same number of iterations for a specified degree of accuracy. The systolic designs for stationary and nonstationary second order Richardson iterative methods take same area and time, for both normal and pipelined design. The pipelined systolic design has complex control logic due to the switching process which takes place after the 1^{st} iteration. The data is permanently stored in the cells which increases the area requirements and hence the cost of the cells. The local memory of each cell increases the throughput of the overall system as the communication time is reduced. The array is independent of the host computer after the data has been transferred to the array and reuses its local data and resources as many times until the solution is achieved. The draw back of such a design is that as the problem size increases the length of the array is increased.

For large linear systems of linear equations the A.O.R. iterative method converges faster. The S.A.O.R. iterative method using the Conrad-Wallach technique for $n = 4$ takes 72.72 % of the computational time required by the A.O.R. iterative method. The A.O.R. and S.A.O.R. linear systolic arrays require extra hardware compared to the S.S.O.R. linear systolic array, but are more flexible as different combinations of ω and r yield different iterative methods, hence the A.O.R. and S.A.O.R. linear arrays are more general purpose.

5.4 Example

The following example of a (5×5) system of linear equations is generated by the discretisation of the system $(1 - x^3) sin\, h\pi x$.

$$A = \begin{bmatrix} 2.274156 & -1 & 0 & 0 & 0 \\ -1 & 2.274156 & -1 & 0 & 0 \\ 0 & -1 & 2.274156 & -1 & 0 \\ 0 & 0 & -1 & 2.274156 & -1 \\ 0 & 0 & 0 & -1 & 2.274156 \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} 1.247647 \\ 0.228754 \\ 0.225092 \\ 0.244276 \\ 13.842661 \end{bmatrix}$$

where x_i is arbitary and for this example is taken to be $(12121)^T$.

$a = 0.274156$, $b = 4.274156$ and $Error = 0.05$

Number of iterations performed = 11

```
  exact      result      error
1.552483, 1.556690, -0.004207
2.286404, 2.307172, -0.020768
3.426299, 3.436317, -0.010018
5.294988, 5.369592, -0.074604
8.396328, 8.458521, -0.062193
```

Chapter 6

2D-Design of Systolic Arrays

This chapter presents the systolic designs for the Jacobi Overrelaxation (J.O.R.), Successive Overrelaxation (S.O.R.), Accelerated Overrelaxation (A.O.R.), stationary and non-stationary second order Richardson, Conjugate Gradient and Preconditioned Conjugate Gradient methods for the iterative solution of large linear systems. The linear systems are obtained from the discretisation of a two and three dimensional Laplace equation by the finite difference method and the coefficient matrices are sparse symmetric and positive definite. We investigate the hardware implementation of this system to achieve a low cost and optimal area efficient VLSI solution.

6.1 Introduction

Several iterative solutions to the large sparse linear systems obtained from the finite difference discretisation of boundary value problems have been developed like the Jacobi, Gauss-Seidel and S.O.R. methods (Young [82], Hageman and Young [34]). Further several acceleration and overrelaxation strategies have been incorporated in order to decrease the computation time.

The S.O.R., A.O.R., S.S.O.R. and non-stationary iterative methods depend upon parameters which sometimes are difficult to choose properly. For the Chebyshev acceleration (second order Richardson iterative method) to be successful a good estimate of the largest and smallest eigenvalues of the iteration matrix is required.

This chapter deals with the well known Conjugate Gradient method developed by Hestenes and Stiefel [39], without having the difficulty of the above mentioned methods. Later in the chapter we describe the Preconditioned Conjugate Gradient method in which a preconditioning strategy is introduced, to improve the rate of convergence, which is quite dramatic for large linear systems.

The systolic array solution for the Jacobi, Gauss-Seidel and S.O.R. iterative methods for dense and one dimensional problems have been previously presented in Margaritis [51]. The solutions consist of pipeline designs. Several other designs have been presented to enhance the utilisation of the array either by reducing the area or the execution time, and by using different techniques. See chapter 3 and Kung [45], Bekakos [5], Suros and Montagne [75], Gusev [32] and Berzins, Buckley and Dew [7].

For the systems under consideration in this chapter these designs do not offer greater efficiency due to the sparseness of the system. Most of the time the processors will be doing unwanted computations and need extra I/O which increases the bandwidth for pumping the data.

A design is required that can reduce the area as well as the I/O and hence the bandwidth of the data path. The unwanted computations can be eliminated (skipped) by using the concept of Virtual IPS cells. These cells do not require the expensive multiplier and adder, therefore area is saved and so the cost. Also these cells do not require the matrix elements to be pumped from the host hence the I/O bandwidth is saved. These cells consist of delays with an amount equal to the clock period of the synchronising clock. The execution time of the iteration remains the same. The sparseness of the matrix increases as the size of the problem increases. The efficient solution to this problem will be to manufacture a Programmable Virtual IPS cell which will enable the array to be used for different problem sizes. Using these concepts improved sparse systolic designs for the matrix iterative methods are presented.

6.2 The overrelaxation iterative methods

Consider equation,

$$u_{i,j} \approx \frac{u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1}}{4} \tag{6.2.1}$$

which represents the evaluation of the five point molecule and can be rewritten in the form,

$$-u_{i,j-1} - u_{i-1,j} + 4u_{i,j} - u_{i+1,j} - u_{i,j+1} = 0. \tag{6.2.2}$$

We want to develop a system of linear equations, for the Laplace equation in the unit square with 9 internal mesh points (see figure 6.1). The left, right, lower and upper boundaries i.e. $u_{0,j}$, $u_{n+1,j}$, $u_{i,0}$ and $u_{i,n+1}$ for $i, j = 0, 1, \ldots, n+1$ respectively are known ($n = 3$ for the example discussed). Expanding equation (6.2.2) for $i, j = 1, 2, \ldots, n$ ($n = 3$) we get, for $i = 1, j = 1$ (by scanning the points row wise from left to right, and from lower to upper)

$$-u_{1,0} - u_{0,1} + 4u_{1,1} - u_{2,1} - u_{1,2} = 0.$$

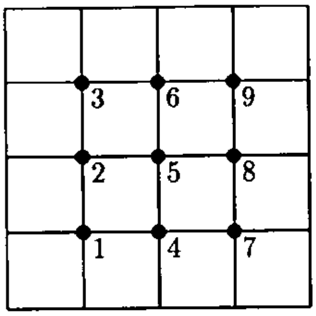


Figure 6.1: Unit square with 9 internal mesh points.

$u_{1,0}$ and $u_{0,1}$ are the known boundary values, we shift them to the right hand side and get,

$$4u_{1,1} - u_{2,1} - u_{1,2} = u_{1,0} + u_{0,1} \tag{6.2.3}$$

Similarly for $i = 2$, and $j = 1$,

$$-u_{1,1} + 4u_{2,1} - u_{3,1} - u_{2,2} = u_{2,0} \tag{6.2.4}$$

for $i = 3$, and $j = 1$,

$$-u_{2,1} + 4u_{3,1} - u_{3,2} = u_{3,0} + u_{4,1} \tag{6.2.5}$$

for $i = 1$, and $j = 2$,

$$-u_{1,1} + 4u_{1,2} - u_{2,2} - u_{1,3} = u_{0,2} \tag{6.2.6}$$

for $i = 2$, and $j = 2$,

$$-u_{2,1} - u_{1,2} + 4u_{2,2} - u_{3,2} - u_{2,3} = 0 \tag{6.2.7}$$

for $i = 3$, and $j = 2$,

$$-u_{2,1} - u_{2,2} + 4u_{3,2} - u_{3,3} = u_{4,2} \tag{6.2.8}$$

for $i = 1$, and $j = 3$,

$$-u_{1,2} + 4u_{1,3} - u_{2,3} = u_{1,4} + u_{0,3} \tag{6.2.9}$$

for $i = 2$, and $j = 3$,

$$-u_{2,2} - u_{1,3} + 4u_{2,3} - u_{3,3} = u_{2,4} \tag{6.2.10}$$

and for $i = 3$, and $j = 3$,

$$-u_{3,2} - u_{2,3} + 4u_{3,3} = u_{3,4} + u_{4,3} \tag{6.2.11}$$

The equations (6.2.3), (6.2.4), (6.2.5), (6.2.6), (6.2.7), (6.2.8), (6.2.9), (6.2.10) and (6.2.11) can be written in the matrix form as follows,

$$\begin{bmatrix} 4 & -1 & 0 & -1 & & & & & \\ -1 & 4 & -1 & 0 & -1 & & & & \mathbf{0} \\ 0 & -1 & 4 & 0 & 0 & -1 & & & \\ -1 & 0 & 0 & 4 & -1 & 0 & -1 & & \\ & -1 & 0 & -1 & 4 & -1 & 0 & -1 & \\ & & -1 & 0 & -1 & 4 & 0 & 0 & -1 \\ & & & -1 & 0 & 0 & 4 & -1 & 0 \\ \mathbf{0} & & & & -1 & 0 & -1 & 4 & -1 \\ & & & & & -1 & 0 & -1 & 4 \end{bmatrix} * \begin{bmatrix} u_{1,1} \\ u_{2,1} \\ u_{3,1} \\ u_{1,2} \\ u_{2,2} \\ u_{3,2} \\ u_{1,3} \\ u_{2,3} \\ u_{3,3} \end{bmatrix} = \begin{bmatrix} u_{1,0} & + & u_{0,1} \\ u_{2,0} & & \\ u_{3,0} & + & u_{4,1} \\ & & u_{0,2} \\ & & 0 \\ & & u_{4,2} \\ u_{1,4} & + & u_{0,3} \\ u_{2,4} & & \\ u_{3,4} & + & u_{4,3} \end{bmatrix} \tag{6.2.12}$$

As the right hand side of equation (6.2.12) is constant we can replace them by a single subscript letter b_i for $i = 1, 2, \ldots n$ and similarly the $u_{i,j}$ can be replaced by the single subscript as x_i for $i = 1, 2, \ldots, n$ and we get the equation,

$$\begin{bmatrix} 4 & -1 & 0 & -1 & & & & & \\ -1 & 4 & -1 & 0 & -1 & & & & \mathbf{0} \\ 0 & -1 & 4 & 0 & 0 & -1 & & & \\ -1 & 0 & 0 & 4 & -1 & 0 & -1 & & \\ & -1 & 0 & -1 & 4 & -1 & 0 & -1 & \\ & & -1 & 0 & -1 & 4 & 0 & 0 & -1 \\ & & & -1 & 0 & 0 & 4 & -1 & 0 \\ \mathbf{0} & & & & -1 & 0 & -1 & 4 & -1 \\ & & & & & -1 & 0 & -1 & 4 \end{bmatrix} * \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \\ b_8 \\ b_9 \end{bmatrix} \tag{6.2.13}$$

which represents the linear system

$$A\underline{x} = \underline{b}.$$

6.2.1 The J.O.R. iterative method

Now equation (6.2.1) can be written as an iterative procedure using the subscripts,

$$u_{i,j}^{(k+1)} = \frac{u_{i+1,j}^{(k)} + u_{i-1,j}^{(k)} + u_{i,j+1}^{(k)} + u_{i,j-1}^{(k)}}{4} \tag{6.2.14}$$

Adding and subtracting $u_{i,j}^{(k)}$ on the right hand side we get,

$$u_{i,j}^{(k+1)} = u_{i,j}^{(k)} + [\frac{u_{i+1,j}^{(k)} + u_{i-1,j}^{(k)} + u_{i,j+1}^{(k)} + u_{i,j-1}^{(k)} - 4u_{i,j}^{(k)}}{4}] \tag{6.2.15}$$

The term in brackets is known as the "residual" and will be zero when the solution is achieved. The bracketed term is an adjustment to the old approximation, which gives the improved approximation. If a larger value is added, faster convergence will result. This is known as overrelaxation. If "ω" is the overrelaxation factor then equation (6.2.15) can be written as,

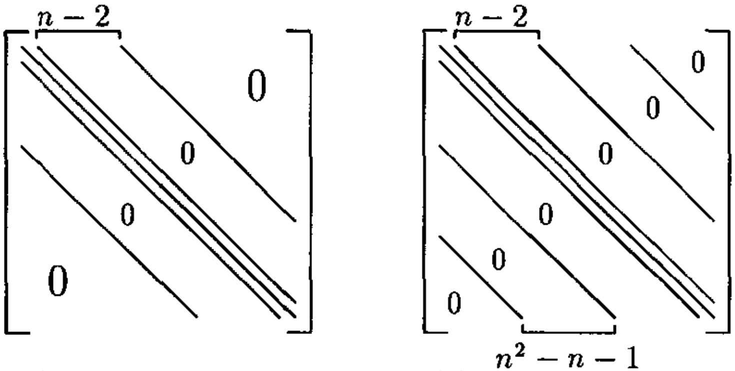
$$u_{i,j}^{(k+1)} = u_{i,j}^{(k)} + \omega[\frac{u_{i+1,j}^{(k)} + u_{i-1,j}^{(k)} + u_{i,j+1}^{(k)} + u_{i,j-1}^{(k)} - 4u_{i,j}^{(k)}}{4}] \tag{6.2.16}$$

Equation (6.2.16) represents the J.O.R. iterative method. Maximum acceleration is obtained for some optimum value of "ω", which lies in the range 0 and $\frac{2}{b}$, where b is the largest eigenvalue of the matrix A.

6.2.2 The S.O.R. iterative method

Similarly we can apply the S.O.R. iterative method to the equation (6.2.1) which can be written as an iterative procedure using the subscripts,

$$u_{i,j}^{(k+1)} = \frac{u_{i+1,j}^{(k)} + u_{i-1,j}^{(k+1)} + u_{i,j+1}^{(k)} + u_{i,j-1}^{(k+1)}}{4} \tag{6.2.17}$$



(a) Two dimensional (b) Three dimensional

Figure 6.2: Two and three dimensional forms of sparse matrices.

Adding and subtracting $u_{i,j}^{(k)}$ on the right hand side we get,

$$u_{i,j}^{(k+1)} = u_{i,j}^{(k)} + [\frac{u_{i+1,j}^{(k)} + u_{i-1,j}^{(k+1)} + u_{i,j+1}^{(k)} + u_{i,j-1}^{(k+1)} - 4u_{i,j}^{(k)}}{4}] \tag{6.2.18}$$

The term in brackets is known as the "residual" and will be zero when the solution is achieved. The bracketed term is an adjustment to the old approximation, which gives the improved approximation. If a larger value is added, faster convergence will result. This is known as overrelaxation. If "ω" is the overrelaxation factor then equation can be written as,

$$u_{i,j}^{(k+1)} = u_{i,j}^{(k)} + \omega[\frac{u_{i+1,j}^{(k)} + u_{i-1,j}^{(k+1)} + u_{i,j+1}^{(k)} + u_{i,j-1}^{(k+1)} - 4u_{i,j}^{(k)}}{4}] \tag{6.2.19}$$

Equation (6.2.19) represents the S.O.R. iterative method. Maximum acceleration is obtained for some optimum value of "ω", which lies between 1 and 2.

6.3 Virtual IPS cell

The two and three dimensional problems obtained from the discretisation of the partial differential equations encompasses matrices of the form, shown in figure (6.2). The two dimensional matrices have $2(n-2)$ null sub diagonals, where n is the size of the grid. No useful computations are performed by these null

Grid points	9	16	81	196
Group				
Group 1	2	4	14	24
Group 2	10	22	142	362
Total	12	26	156	386

Table 6.1: Number of Virtual IPS cells vs Grid points.

sub diagonals, and they cannot be eliminated due to synchronisations of the data movement in the systolic array. The data movement can be synchronised in the systolic array by delaying the data movement for the null sub diagonals, i.e. introducing a Virtual IPS cell in place of a IPS cell. The working of a Virtual IPS cell is the same as the ordinary IPS cell except that it does not have the multiplier and adder circuitry. The Virtual IPS cell consists of a simple D flip flop, the data on the input channel of the D flip flop is available on the output channel after one clock cycle. A Virtual IPS cell (VC) is shown in figure (6.3).

As the grid size is increased or the number of points in the system are increased the number of null sub diagonals is increased. Table (6.1) shows the number of null sub diagonals and the size of the grid. A two dimessional problem with $(n * n)$ grid generates a $(n^2 * n^2)$ matrix with the band width, $w = 2n + 1$. The number of null subdiagonals in this band width is $2(n - 2)$. A three dimensional problem with $(n * n)$ grid generates an $(n^3 * n^3)$ matrix with band width, $w = 2n^2 + 1$. In this case there are two groups of the null sub diagonals. One contains $2(n - 2)$ null sub diagonals and the other contains $2(n^2 - n - 1)$ null sub diagonals as indicated in figure (6.2), with a total of $2(n^2 - 3)$ null sub diagonals. In table (6.1) group 1 gives the number of Virtual IPS cells for a two dimensional problem, whereas in the case of a three dimensional problem both groups are present. The working of the Programmable Virtual cell is shown in figure (6.3).

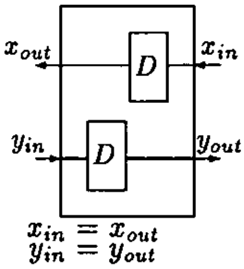


Figure 6.3: Virtual IPS cell.

A cascade of Virtual IPS cells can be used to implement the systolic array. This solution is not very graceful, as the Virtual IPS cell is very small in size and it will be quite time consuming to set the array for different grid sizes. Also the wire connections will be long and so the signals and data will have delays. To solve this problem the idea of a Programmable Virtual IPS cell (PVC) is introduced. Such a cell may contain a number of cascaded Virtual cells depending on the implementations. The PVC can be a module, which can be easily plugged into the systolic array. A partial realisation of such a design with four delays is shown in figure (6.4) (i.e. a single input channel). The PVC can be programmed either by hardware switches to provide the required delay or can be done by software switches. Flexibility in the PVC is obtained at the cost of extra control logic.

6.4 Systolic designs for the overrelaxation iterative methods

6.4.1 Systolic Sparse Array Design (SSAD) for the J.O.R. iterative method

To understand the operational aspects of the design a simple model problem of the Laplace equation in the unit square with 9 internal mesh points (see

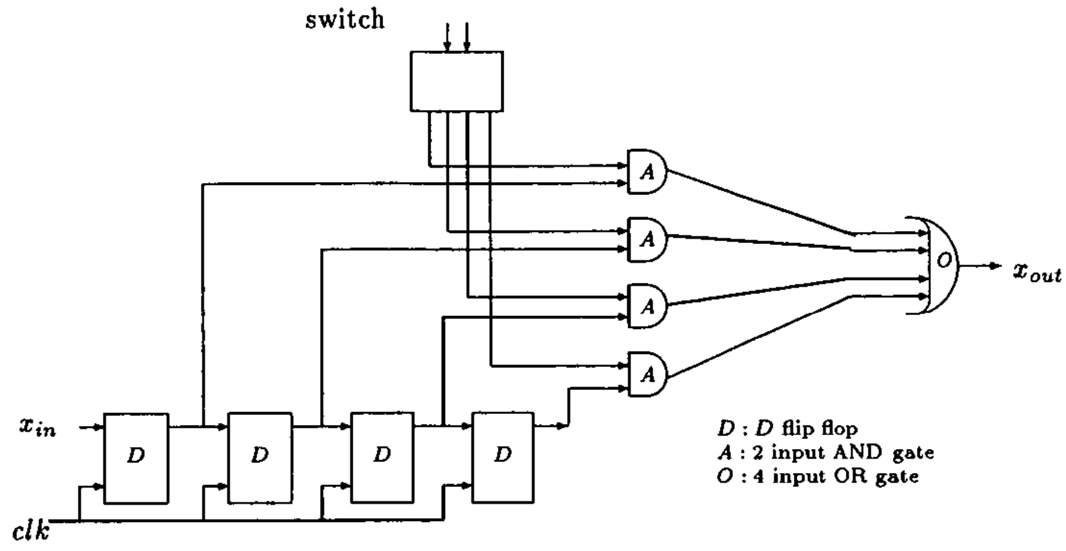


Figure 6.4: Section of a Programmable Virtual IPS cell.

Method	Time	IPS	Virtual IPS	DIV-ADD-IPS
J.O.R.	$(2n^2 + 2n - 1)$	4	$2(n - 2)$	1

Table 6.2: Area and time requirements for the 2D-J.O.R. systolic design.

figure (6.1)) is selected leading to the sparse matrix A shown in figure (6.5).

The design for the model example consists of 4 IPS cells, a boundary cell (DIV-ADD-IPS cell) and two Programmable Virtual IPS cells (which can simulate 4 IPS cells each, in other words they are capable of producing 4 delays each). The black boxes in figure (6.6) represent the Programmable Virtual IPS cells and the box with a cut is the boundary cell. The systolic design is simulated in OCCAM. The design can solve any size 2D-discretised problem. The systolic simulator for the J.O.R. iterative method solving 2D-discretised problems was tested for correctness of the design for different sizes of the problems and varying overrelaxation factor "ω". The area and time requirements for the J.O.R. design are shown in table (6.2), where "n" is the size of the grid. Figure (6.6) represents the first 10 snap shots of the J.O.R. iterative method.

$$\frac{1}{4}\begin{array}{|ccc|ccc|ccc|}\hline 4 & -1 & & -1 & & & & & \\ -1 & 4 & -1 & & -1 & & & & \\ & -1 & 4 & & & -1 & & & \\ \hline -1 & & & 4 & -1 & & -1 & & \\ & -1 & & -1 & 4 & -1 & & -1 & \\ & & -1 & & -1 & 4 & & & -1 \\ \hline & & & -1 & & & 4 & -1 & \\ & & & & -1 & & -1 & 4 & -1 \\ & & & & & -1 & & -1 & 4 \\ \hline \end{array}$$

Figure 6.5: Sparse matrix A.

The system of equations solved is represented by equation (6.2.13).

To compute $x_2^{(1)}$ the following computation is done,

$$x_2^{(1)} = (1-\omega)x_2^{(0)} + (b_2 - a_{23}x_3^{(0)} - a_{25}x_5^{(0)} - a_{21}x_1^{(1)})$$

In the snap shots shown in figure (6.6) at time step 5, $y_2^{(0)}$ enters the right most cell i.e. cell 7 (cells are numbered 1 through 7 from left to right irrespective of the type) and $a_{23}x_3^{(0)}$ is computed i.e. $y_2^{(0)} = a_{23}x_3^{(0)}$. $y_2^{(1)}$ enters the leftmost cell i.e. cell 1 and is passed unaltered. At time step 6, $y_2^{(0)}$ and $y_2^{(1)}$ enter the cells 6 and 2 (Programmable Virtual IPS cells) respectively. They remain in these Programmable Virtual IPS cells for one clock cycle and no operation is performed, simulating the dummy computation $a_{24}x_4^{(0)}$ for cell 6. At time step 7, $y_2^{(0)}$ and $y_2^{(1)}$ enter the cells 5 and 3 respectively to form $y_2^{(0)} = a_{23}x_3^{(0)} + a_{25}x_5^{(0)}$ and $y_2^{(1)} = a_{21}x_1^{(0)}$. At time step 8, $y_2^{(0)}$, $y_2^{(1)}$, b_2, a_{22}, $x_2^{(0)}$ and ω enter cell 4 (boundary cell) and $x_2^{(1)}$ is computed.

Table (6.3) represents the results of the design, with ω varied from 0.2 to 1.0. For this particular problem the fastest convergence is obtained when $\omega = 1$. If ω is increased beyond 1 then the method diverges.

ω	.2	.3	.4	.5	.6	.7	.8	.9	1.0
size									
16	291	200	152	124	109	89	78	69	63
81	885	616	476	390	330	287	255	228	220
196	1620	1142	889	730	622	542	482	435	395
361	2417	1721	1348	1117	954	837	744	673	668

Table 6.3: Iteration count for the J.O.R. iterative method for different sizes and ω.

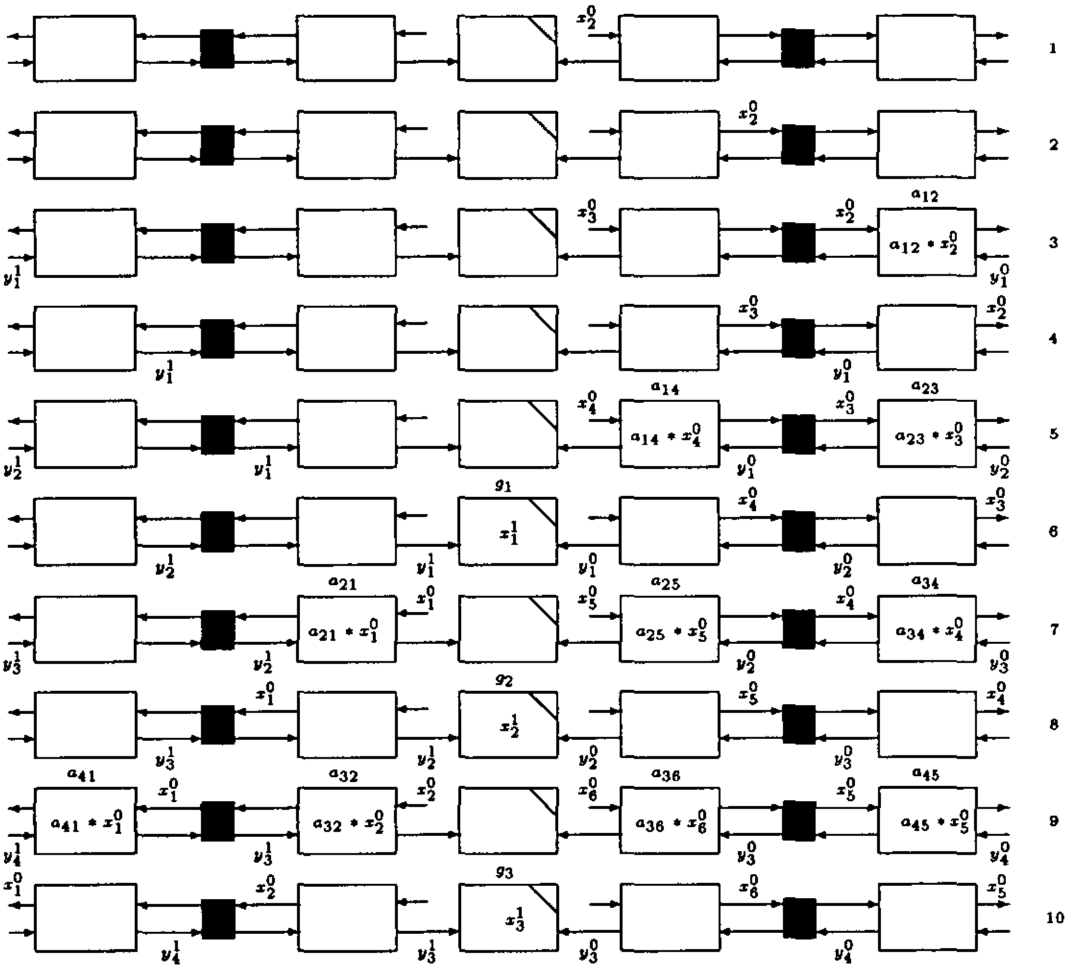
ω	1.1	1.2	1.3	1.4	1.5	1.6	1.7	1.8	1.9
size									
16	28	21	17	20	24	32	45	72	151
81	94	77	63	49	36	39	48	71	146
196	182	151	125	100	80	60	52	74	152
361	286	236	196	161	132	99	69	81	161

Table 6.4: Iteration counts for the S.O.R. iterative method for different sizes and ω.

6.4.2 Systolic design for the S.O.R. iterative method

The systolic design is similar to the J.O.R. iterative method, except that the J.O.R. iterative method uses the old values of the initial approximation, whereas the S.O.R. iterative method uses the newly computed approximations. The time and area requirements of the design are same as the J.O.R. systolic design (see the table (6.2)). Figure (6.7) represents the 1^{st} 10 snap shots of the S.O.R. iterative method for solving the linear system of equations represented by equation (6.2.13).

The systolic simulator was tested for correctness of the design for different sizes of the problem and varying acceleration factor ω. The results are shown in table (6.4).



g_i represents $a_{ii}, \omega, b_i, x_i^0$ and
x_i^1 represents $((((b_i - y_i^0 - y_i^1)\omega)/a_{ii}) + (1 - \omega)x_i^0)$

Figure 6.6: 1^{st} 10 snap shots for the J.O.R. iterative method.

ω	1.1	1.2	1.3	1.4	1.5	1.6	1.7	1.8	1.9
size									
64	29	22	17	20	28	34	47	74	153
729	137	112	89	69	48	44	63	93	186
2744	327	269	218	173	132	92	68	105	203

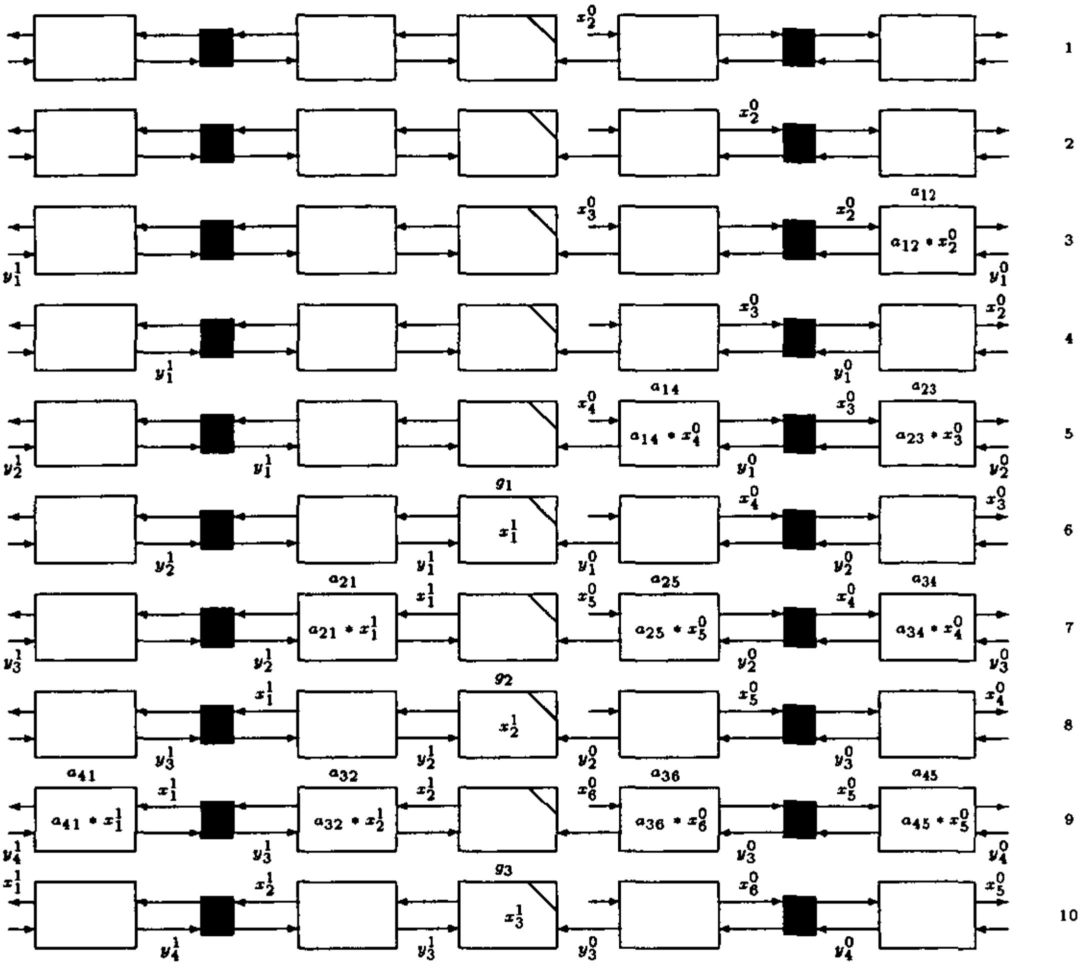
Table 6.5: Iteration counts for the S.O.R. iterative method for different sizes and ω.

6.4.3 Systolic designs for the S.S.O.R., A.O.R. and S.A.O.R. iterative methods

The systolic designs for the S.S.O.R., A.O.R. and S.A.O.R. iterative methods for 2 dimensional problems are straight forward to implement. The S.O.R. design can be extended to perform the S.S.O.R. iterative method if the order of the system is reversed on each iteration.

6.5 3D Problems

The 3D matrix form obtained from the discretisation of the partial differential equation is shown in figure (6.2). The linear systolic array design for the J.O.R., S.O.R., stationary and non-stationary second order Richardson method, C.G. and P.C.G. methods have been developed, simulated and tested for correctness. The structure of the linear systolic array is shown in the figure (6.8). Table (6.5) shows the results for the S.O.R. iterative method for different problem sizes and parameter "ω". The working of the designs are similar to that of the 2D problems, but the area and time requirements are different, due to the larger bandwidth of the 3D problems.



g_i represents $a_{ii}, \omega, b_i, x_i^0$ and
x_i^1 represents $((((b_i - y_i^0 - y_i^1)\omega)/a_{ii}) + (1 - \omega)x_i^0)$

Figure 6.7: 1^{st} 10 snap shots for the S.O.R. iterative method.

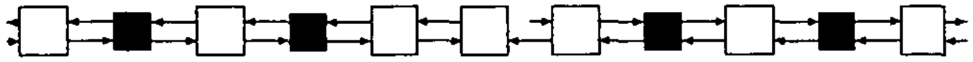


Figure 6.8: 3D design for the S.O.R. systolic array.

6.6 The Gradient method

The linear system of equation represented by

$$A\underline{x} = \underline{b}, \tag{6.6.1}$$

can be written as the gradient of the quadratic function,

$$F(\underline{x}) = \frac{1}{2}(\underline{x}, A\underline{x}) - (\underline{x}, \underline{b}), \tag{6.6.2}$$

where A is a positive definite $(n*n)$ matrix, i.e. all the eigenvalues are positive, $\underline{x}$ is the unknown and $\underline{b}$ is the known vector. We want to find vector $\underline{x}$ which minimizes $F(\underline{x})$. The Gradient of a function $F(\underline{x})$ is defined as,

$$\nabla F(\underline{x}) = \frac{\partial F(\underline{x})}{\partial \underline{x}}. \tag{6.6.3}$$

The function $F(\underline{x})$ defines an ellipsoid in n-dimensional space of the $\underline{x}$, with a common center at $A^{-1}\underline{b}$. If $\underline{x}^{(k)}$ is an arbitrary solution of equation (6.6.1), then the residual vector is given by,

$$\underline{r}^{(k)} = \underline{b} - A\underline{x}^{(k)}, \tag{6.6.4}$$

$$\underline{r}^{(k)} = -\nabla F(\underline{x}^{(k)}). \tag{6.6.5}$$

The Gradient method can be represented by the non-stationary iterative method,

$$\underline{x}^{(k+1)} = \underline{x}^{(k)} + \alpha_k \underline{p}^{(k)}, \tag{6.6.6}$$

where α_k is a scalar and $p^{(k)}$ is a vector to be defined.

We want to choose α_k such that the quadratic function $F(\underline{x}^{(k+1)})$ will be minimum in the direction $\underline{p}^{(k)}$. Using equation (6.6.2) we can write equation (6.6.6) as,

$$F(\underline{x}^{(k+1)}) = \frac{1}{2}((\underline{x}^{(k)} + \alpha_k \underline{p}^{(k)}), A(\underline{x}^{(k)} + \alpha_k \underline{p}^{(k)})) - ((\underline{x}^{(k)} + \alpha_k \underline{p}^{(k)}), \underline{b}). \tag{6.6.7}$$

The gradient of equation (6.6.7) can be written as,

$$\nabla F(\underline{x}^{(k+1)}) = (\underline{p}^{(k)}, (A(\underline{x}^{(k)} + \alpha_k \underline{p}^{(k)}))) - (\underline{p}^{(k)}, \underline{b}), \tag{6.6.8}$$

or

$$\nabla F(\underline{x}^{(k+1)}) = (\underline{p}^{(k)}, A\underline{x}^{(k)}) + (\underline{P}^{(k)}, \alpha_k A\underline{P}^{(k)}) - (\underline{p}^{(k)}, \underline{b}), \tag{6.6.9}$$

$$\nabla F(\underline{x}^{(k+1)}) = (\underline{p}^{(k)}, [(A\underline{x}^{(k)} - \underline{b}) + \alpha_k A\underline{p}^{(k)}]), \tag{6.6.10}$$

$$\nabla F(\underline{x}^{(k+1)}) = (\underline{p}^{(k)}, [-\underline{r}^{(k)} + \alpha_k A\underline{p}^{(k)}]). \tag{6.6.11}$$

The optimum α_k will be given when $\nabla F(\underline{x}^{(k+1)}) = 0$, i.e.

$$(\underline{p}^{(k)}, [-\underline{r}^{(k)} + \alpha_k A\underline{p}^{(k)}]) = 0, \tag{6.6.12}$$

or

$$\alpha_k = \frac{(\underline{p}^{(k)}, \underline{r}^{(k)})}{(\underline{p}^{(k)}, A\underline{p}^{(k)})}. \tag{6.6.13}$$

Thus α_k minimizes $F(\underline{x}^{(k+1)})$ for any given $\underline{p}^{(k)}$.

Hence the Gradient method can be represented as,

$$\underline{x}^{(k+1)} = \underline{x}^{(k)} - \frac{(\underline{p}^{(k)}, \underline{r}^{(k)})}{(\underline{p}^{(k)}, A\underline{p}^{(k)})}\underline{p}^{(k)}. \tag{6.6.14}$$

Now we wish to determine $\underline{p}^{(k)}$ so that the correction term $\alpha_k \underline{p}^{(k)}$ in equation (6.6.6) is minimized.

Let $\underline{p}^{(k)} = \underline{r}^{(k)}$ then we get,

$$\underline{x}^{(k+1)} = \underline{x}^{(k)} - \frac{(\underline{r}^{(k)}, \underline{r}^{(k)})}{(\underline{r}^{(k)}, A\underline{r}^{(k)})}\underline{r}^{(k)} \tag{6.6.15}$$

This method is also known as the Steepest Descent method.

6.7 The Conjugate Direction method

Conjugate Direction methods were initially designed and analysed for minimising the purely quadratic problem represented by (6.6.2) where A is an $(n * n)$ symmetric positive definite matrix.

The Conjugate Direction algorithms have been one of major creativity in the field of linear programming. Indeed, Conjugate Direction methods, especially the method of Conjugate Gradients have proved to be extremely effective in dealing with general objective functions. In addition, Conjugate Direction methods have recently gained acceptance as methods for solving the sparse linear systems of finite difference/element equations.

If A is a symmetric matrix, then two vectors p_1 and p_2 are said to be A-orthogonal, or Conjugate with respect to A if $(p_1, Ap_2) = 0$. Now we describe the standard Conjugate Direction method whose important properties are established in the following theorems.

Theorem 6.1 *If the vectors $\underline{p}^{(j)}$ are mutually conjugate, then they are linearly independent.*

Proof

Suppose that, $\sum_j \alpha_j \underline{p}^{(j)} = 0$ for some scalar j. If k is any one of the values of j, then

$$\left(\underline{p}^{(k)}, A \sum_j \alpha_j \underline{p}^{(j)}\right) = 0,$$

i.e.,

$$\alpha_k \left(\underline{p}^{(k)}, A\underline{p}^{(k)}\right) = 0.$$

Since A is positive definite and $\underline{p}^{(k)} \neq 0$, therefore $\alpha_k = 0$

Corresponding to the $(n*n)$ positive definite matrix A let $\underline{p}^{(0)}, \underline{p}^{(1)}, \ldots, \underline{p}^{(n-1)}$, be n nonzero A-orthogonal vectors. By the above theorem they are linearly independent which implies that the solution vector x can be expressed as,

$$\underline{x} = \alpha_0 \underline{p}^{(0)} + \alpha_1 \underline{p}^{(1)} \cdots + \alpha_{n-1} \underline{p}^{(n-1)}$$

$$\underline{x} = \sum_j \alpha_j \underline{p}^{(j)}$$

By multiplying by A on both sides we get,

$$A\underline{x} = \sum_j \alpha_j A\underline{p}^{(j)}$$

and taking the scalar product with $\underline{p}^{(k)}$ yields,

$$\left(\underline{p}^{(j)}, A\underline{x}\right) = \sum_j \alpha_j \left(\underline{p}^{(j)}, A\underline{p}^{(j)}\right)$$

Solving for α_k gives,

$$\alpha_k = \frac{\left(\underline{p}^{(k)}, A\underline{x}\right)}{\left(\underline{p}^{(k)}, A\underline{p}^{(k)}\right)} = \frac{\left(\underline{p}^{(k)}, \underline{b}\right)}{\left(\underline{p}^{(k)}, A\underline{p}^{(k)}\right)}.$$

This shows that the α_k's and consequently the solution $\underline{x}$ can be found by the evaluation of some simple scalar products. The final result is,

$$\underline{x} = \sum_{j=0}^{n-1} \frac{\left(\underline{p}^{(k)}, \underline{b}\right)}{\left(\underline{p}^{(k)}, A\underline{p}^{(k)}\right)} \underline{p}^{(j)}.$$

The expansion for $\underline{x}$ can be considered to be the result of an iterative process of n steps where at the k^{th} step $\alpha_k \underline{p}_k$ is added. By viewing the procedure this way and allowing for an arbitrary initial point for the iteration, the basic Conjugate Direction method is obtained.

Theorem 6.2 Conjugate Direction Theorem

Let $\left[\underline{p}^{(j)}\right], j = 0, 1, \ldots, n-1$ *be a set of non-zero A-orthogonal vectors. For any* $\underline{x} \in R^n$ *the sequence* $\underline{x}^{(k)}$ *is generated according to,*

$$\underline{x}^{(k+1)} = \underline{x}^{(k)} + \alpha_k \underline{p}^{(k)}, k \geq 0$$

with

$$\alpha_k = \frac{(\underline{r}^{(k)}, \underline{p}^{(k)})}{(\underline{p}^{(k)}, A\underline{p}^{(k)})}$$

and

$$\underline{r}^{(k)} = \underline{b} - A\underline{x}^{(k)}$$

converges to the unique solution $\underline{x}$ *of* $A\underline{x} = \underline{b}$ *after* n *steps, that is* $\underline{x}^{(n)} = \underline{x}$.

Proof

(For proof of this theorem see Luenberger [50]).

6.8 The Conjugate Gradient method

The Conjugate Gradient (C.G.) method was developed by Hestenes and Stiefel [39] for solving linear systems (see Beckman [4], Reid [66]). The solution is usually achieved in less than n steps, where n is the size of the system.

Given a linear system of equations,

$$A\underline{x} = \underline{b}. \tag{6.8.1}$$

The C.G. method for solving equation (6.8.1) can be described as follows:

$$\underline{x}^{(k+1)} = \underline{x}^{(k)} + \lambda_k \underline{p}^{(k)}, \quad \text{k} = 0,\ 1,\ 2,\ \ldots \tag{6.8.2}$$

The initial approximation, $\underline{x}^{(0)}$, to the solution is arbitrary and $p^{(k)}$ is a direction vector given by the relation,

$$\underline{p}^{(k)} = \begin{cases} \underline{r}^{(k)} & \text{if k} = 0 \\ \\ \underline{r}^{(k)} + \alpha_k \underline{p}^{(k-1)} & \text{k} = 1,\ 2,\ \ldots \end{cases} \tag{6.8.3}$$

$$\lambda_k = \frac{(\underline{p}^{(k)}, \underline{r}^{(k)})}{(\underline{p}^{(k)}, A\underline{p}^{(k)})} \quad k = 0, 1, 2, \ldots \tag{6.8.4}$$

$r^{(k)}$ is the residual vector and is given by

$$\underline{r}^{(k)} = \underline{b} - A\underline{x}^{(k)} \quad k = 0, 1, 2, \ldots \tag{6.8.5}$$

$$\alpha_k = \frac{(\underline{r}^{(k)}, A\underline{p}^{(k-1)})}{(\underline{p}^{(k-1)}, A\underline{p}^{(k-1)})} \quad k = 1, 2, \ldots \tag{6.8.6}$$

From equation (6.8.3) we observe that the new direction vector is dependent on the previous direction vector. The Conjugate Gradient method is classified as a direct method for solving small dense system of linear equations, and hence is not a popular method. But if the linear system of equations is sparse and well conditioned then this fact is no longer valid and the Conjugate Gradient method is very powerful as it requires no estimation of the parameters λ and α.

6.8.1 Systolic design for the C.G. iterative method

The algorithm suggests that the 1^{st} step towards computing the new approximation is to calculate the residual vector $\underline{r}^{(k)}$. The next step is to set $\underline{p}^{(0)}$ vector equal to the $\underline{r}^{(0)}$ vector for the first iteratation. The next step is to calculate $A\underline{p}^{(k)}$ and simultaneously compute the dot product $(\underline{r}^{(k)}, \underline{p}^{(k)})$. As soon as the first element of $A\underline{p}^{(k)}$ is available the dot product $(\underline{p}^{(k)}, A\underline{p}^{(k)})$ is evaluated. Next λ_k is computed and then the new approximation is computed.

For the 2nd iteration as soon as the first residual vector element is computed the product $(\underline{r}^{(k+1)}, A\underline{p}^{(k)})$ is evaluated and when this is computed α_k is evaluated and the new direction vector is computed. The remaining procedure is the same as that of the 1^{st} iteration. From the different operations involved i.e. dot product, inner step product, subtraction etc. the cell design has a complex control circuitry.

Figures (6.9, 6.10, 6.11, 6.12, 6.13, and 6.14) represent the data flow for the second iteration of the C.G. method for solving linear system of equation represented by equation (6.2.13). Table (6.6) gives the area and time requirements

Method	Time	IPS	Virtual IPS	Boundary cell
2D-C.G.	$(6n^2+2n+5)+k(8n^2+4n+4)$	5	$2(n-2)$	2

Table 6.6: Area and time requirements for the 2D-C.G. systolic design.

of the C.G. method for solving linear system obtained by discretisation of 2D Laplace equation (k represents the number of iterations).

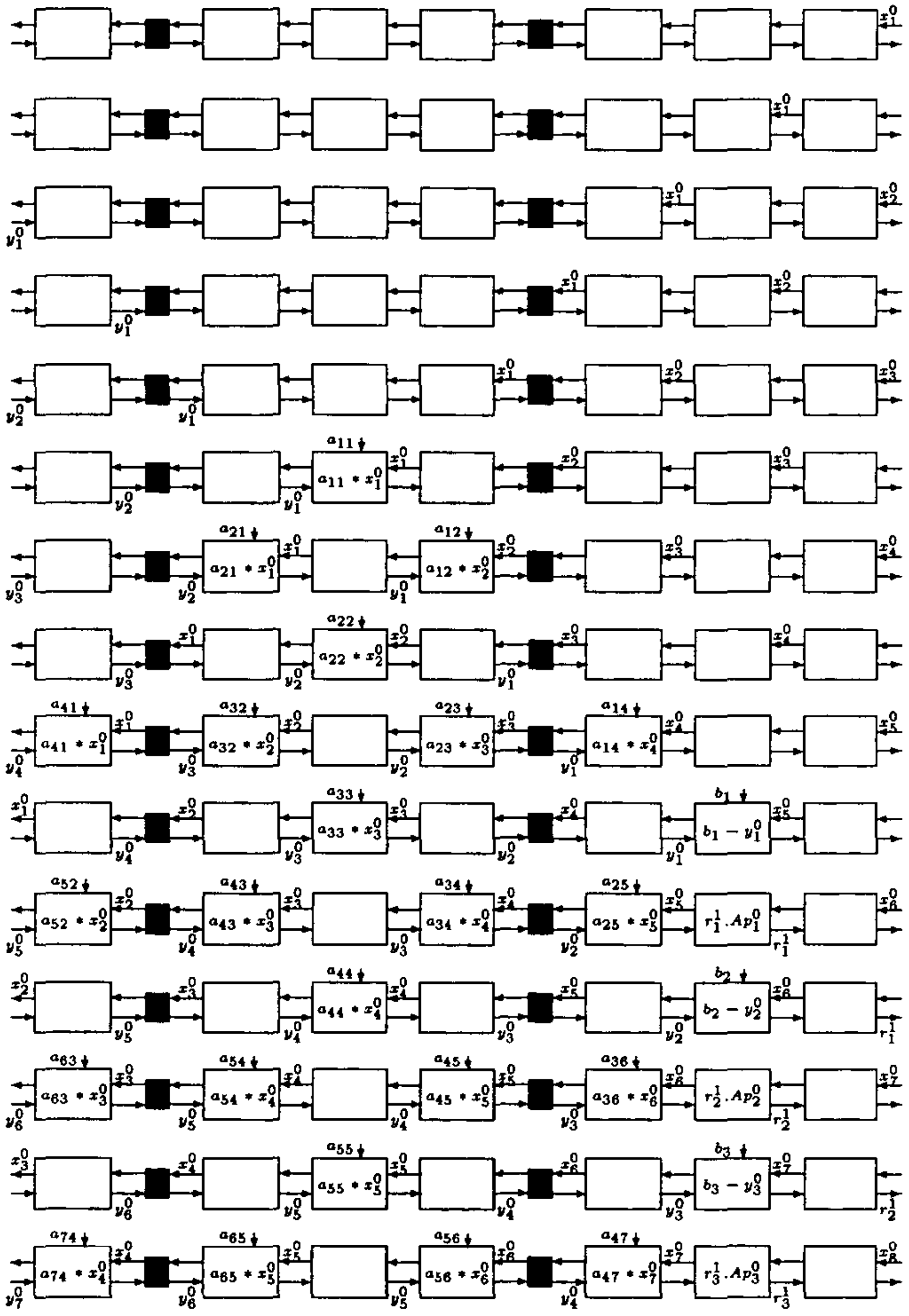


Figure 6.9: Snap shots for the 2^{nd} iteration of the C.G. iterative method.

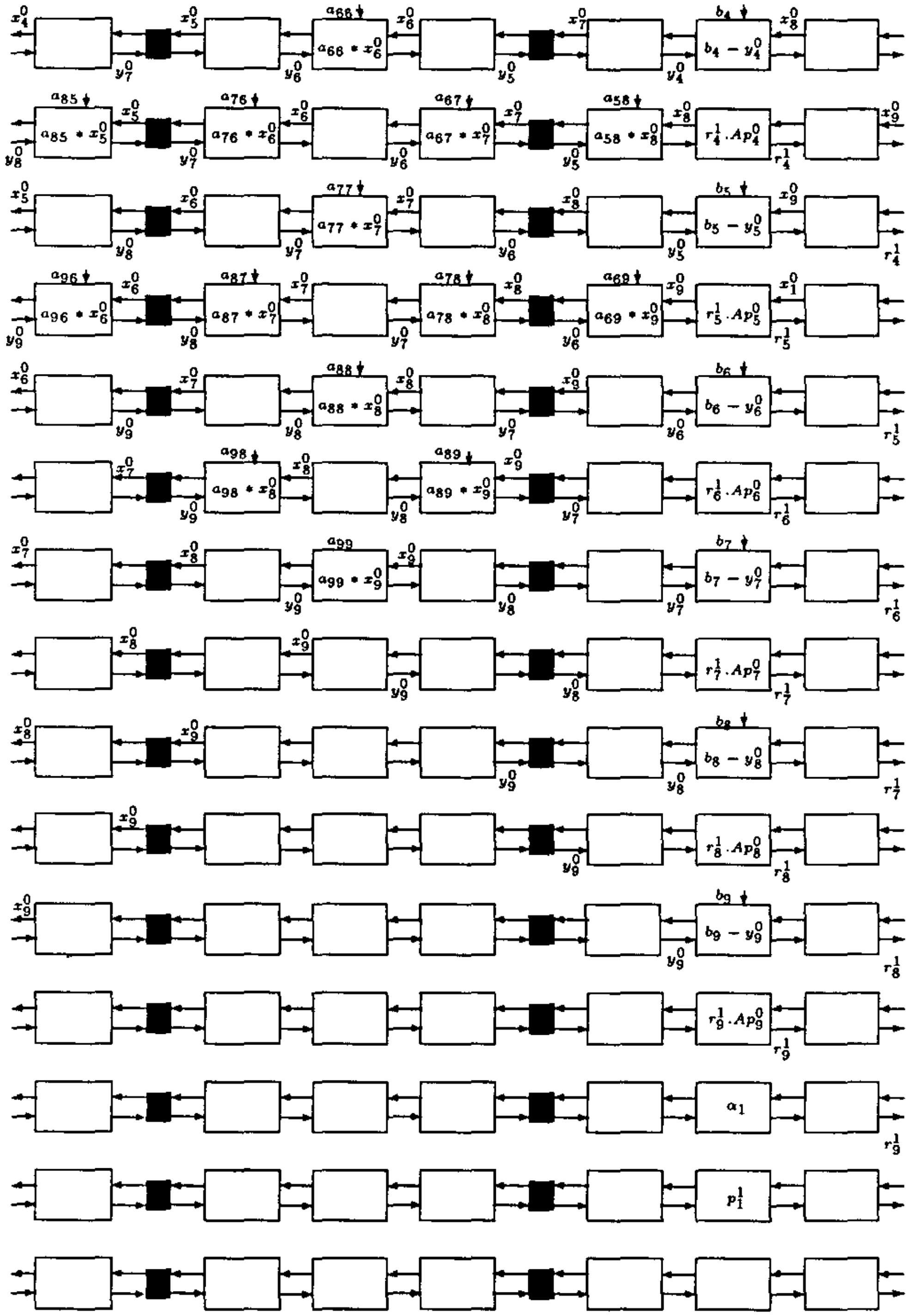


Figure 6.10: Snap shots for the 2^{nd} iteration of the C.G. iterative method.

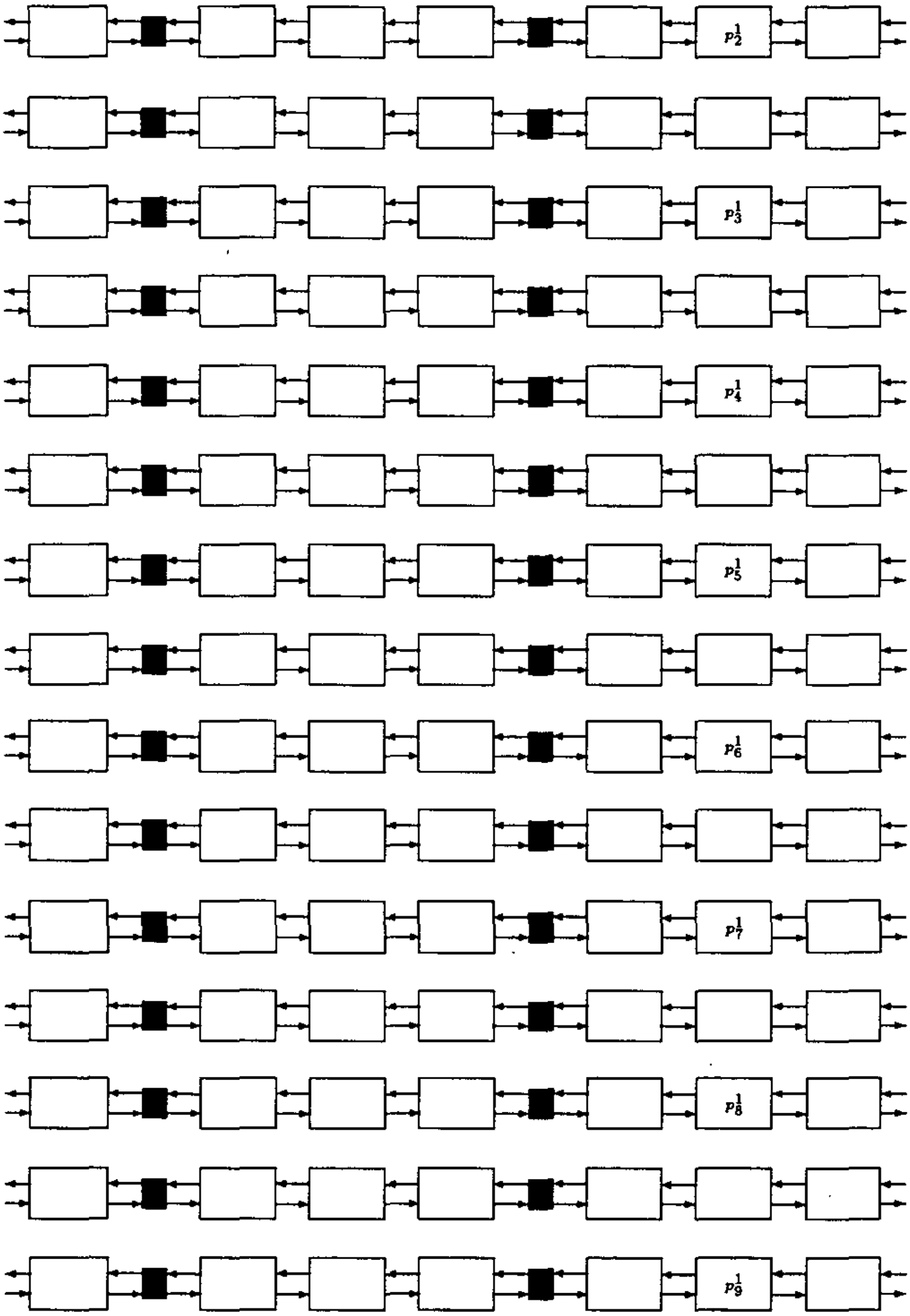


Figure 6.11: Snap shots for the 2^{nd} iteration of the C.G. iterative method.

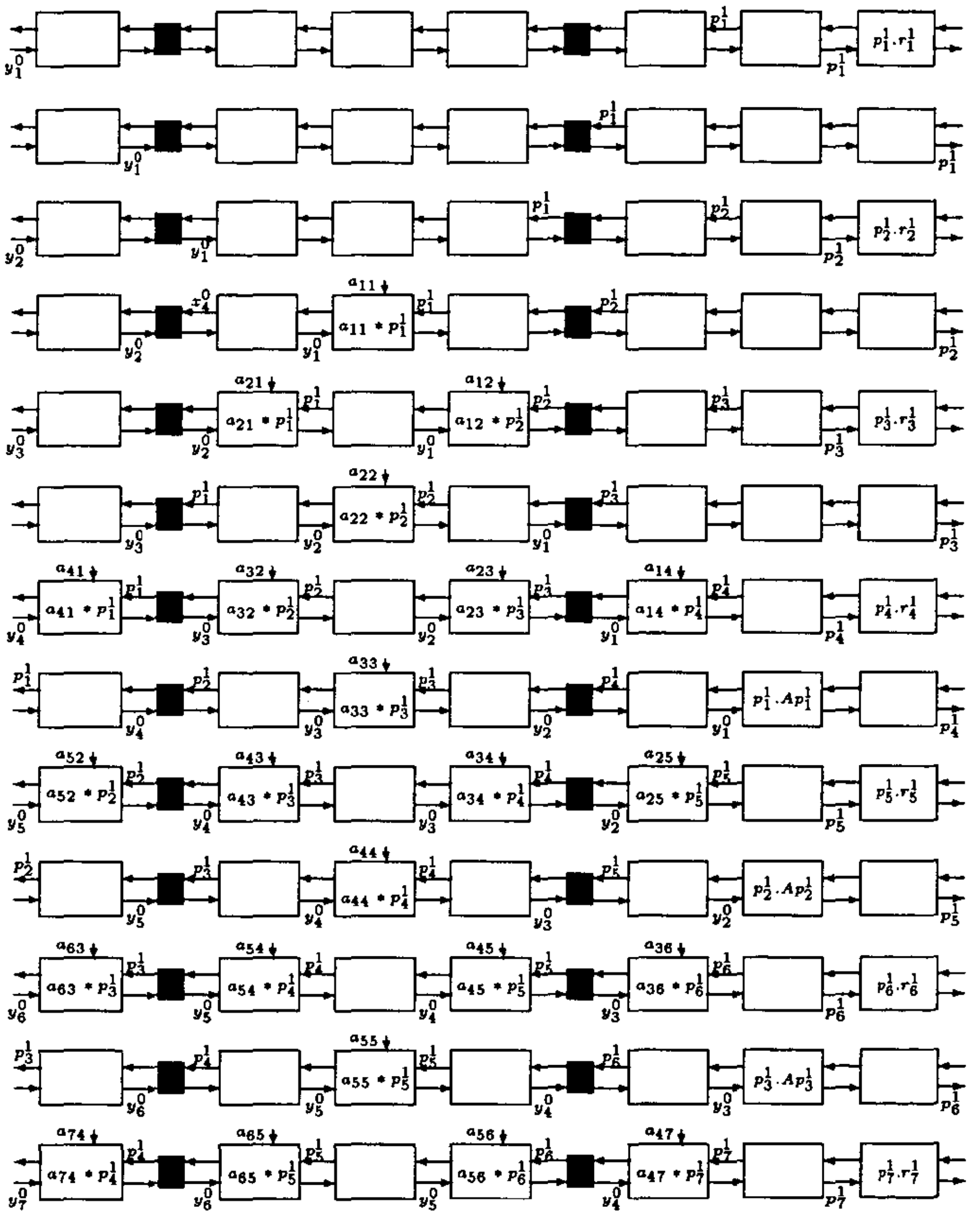


Figure 6.12: Snap shots for the 2^{nd} iteration of the C.G. iterative method.

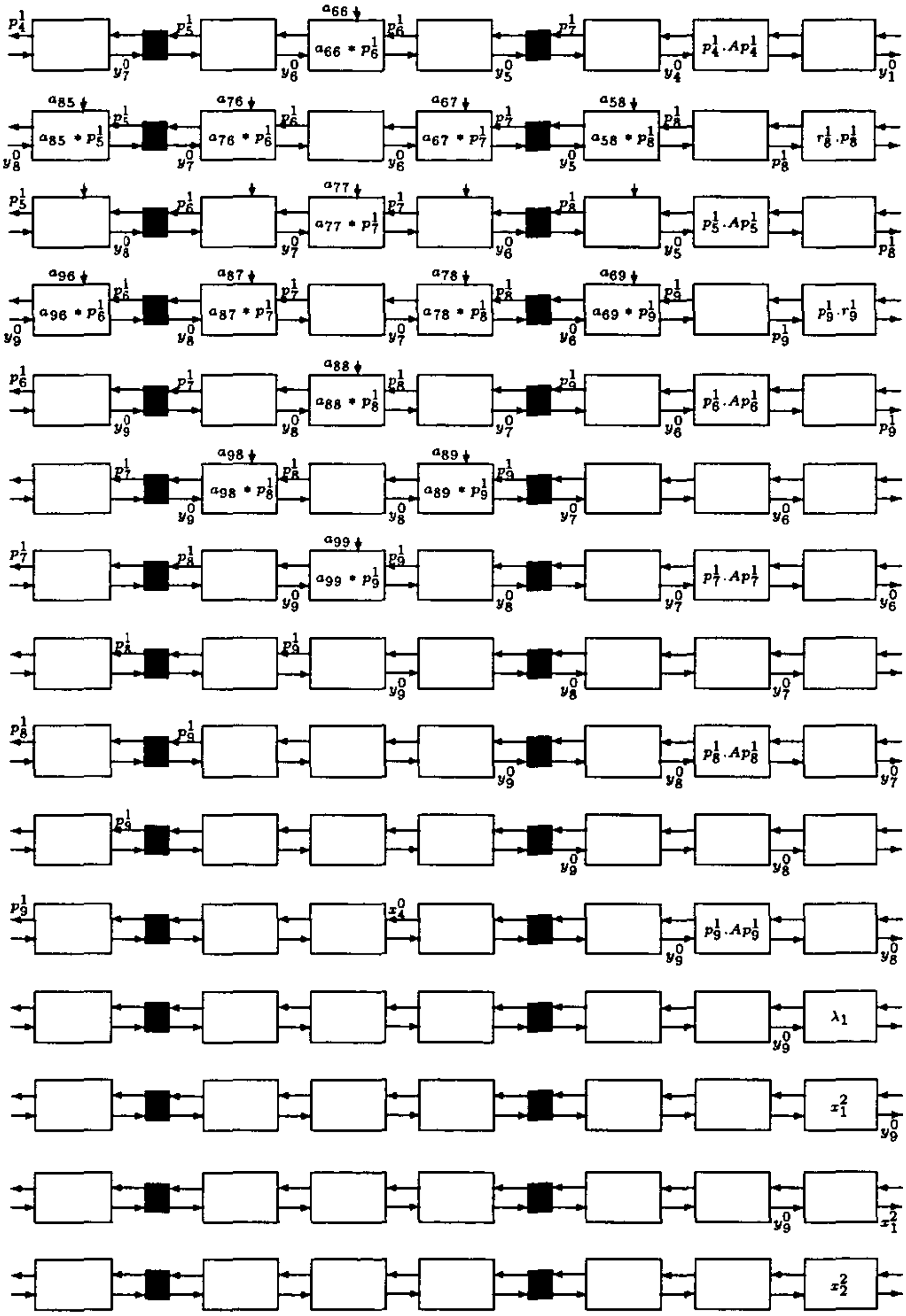


Figure 6.13: Snap shots for the 2^{nd} iteration of the C.G. iterative method.

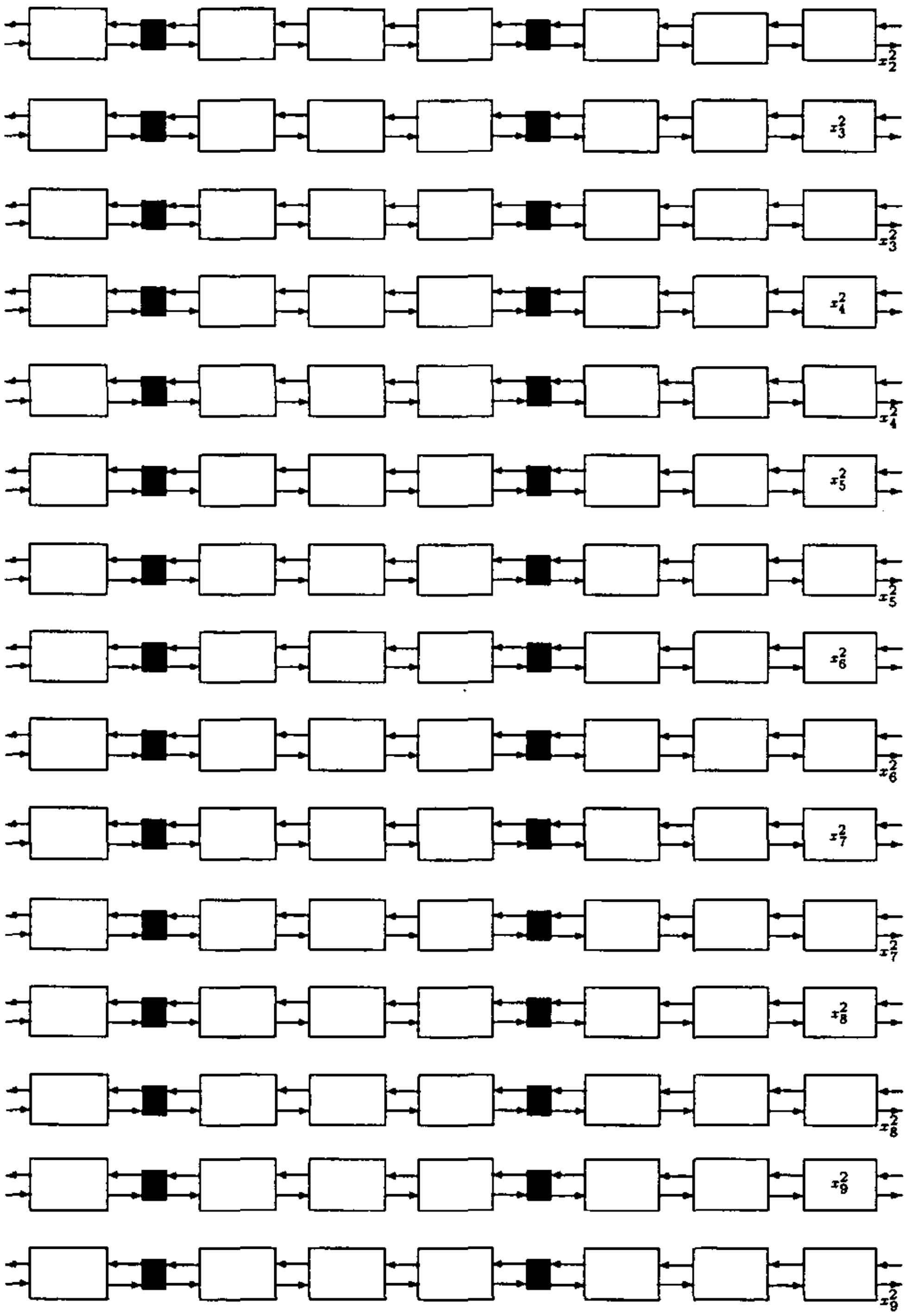


Figure 6.14: Snap shots for the 2^{nd} iteration of the C.G. iterative method.

6.9 The Preconditioned Conjugate Gradient method

Consider the $(n * n)$ symmetric positive definite linear system $A\underline{x} = \underline{b}$. The idea behind the Preconditioned Conjugate Gradient (P.C.G.) is to apply the regular Conjugate Gradient method to the transformed system,

$$\tilde{A}\underline{\tilde{x}} = \underline{\tilde{b}} \tag{6.9.1}$$

where $\tilde{A} = M^{-1}AM^{-1}$, $\underline{\tilde{x}} = M^{-1}\underline{x}$ and $\underline{\tilde{b}} = M^{-1}\underline{b}$ and M is symmetric positive definite. M should be chosen such that $\tilde{A}$ is well conditioned.

The technique of preconditioning was introduced by Evans [14]. The idea was to increase the rate of convergence of an iterative method by minimising the P-condition number of the coefficient matrix A. Decreasing the P-condition number speeds up the rate of convergence, which explains why the preconditioning of the coefficient matrix prior to solving the system is an interesting idea. The preconditioned matrix $M^{-1}A$ has a smaller condition number than the coefficient matrix A, i.e.

$$\frac{\lambda_{max}(M^{-1}A)}{\lambda_{min}(M^{-1}A)} < \frac{\lambda_{max}(A)}{\lambda_{min}(A)} \tag{6.9.2}$$

The other important qualities of the good preconditioner matrix M are that M should be sparse and easy to construct. For the detailed survey of preconditioning strategies see Evans [16]. Several preconditioning techniques have been developed such as,

Diagonal Preconditioner

The simplest preconditioner is the diagonal of the matrix A, i.e.,

$$M = \text{diag}(A) \tag{6.9.3}$$

Unfortunately this preconditioner even though it is perfectly parallel is not efficient regarding the rate of convergence. However simplicity is often a main factor when compared with a more complex form of preconditioner.

Tridiagonal Preconditioner

This preconditioner is simply the tridiagonal part of matrix A,

$$M = \text{tridiag}(A) \tag{6.9.4}$$

Polynomial Preconditioners

The most efficient parallel preconditioners, i.e. explicit polynomial preconditioning is obtained from truncating the finite Neumann expansion of A^{-1}.

Consider the form $A = I - J$

Then,

$$A^{-1} = (I - J)^{-1} = I + J + J^2 + \cdots. \tag{6.9.5}$$

Then we can consider M to have the forms:

1^{st} order $M = I + J$

2^{nd} order $M = I + J(I + J)$

Thus our P.C.G. method can be applied to the systems as,

$$(I + J)A\underline{x} = (I + J)\underline{b}. \tag{6.9.6}$$

In this thesis we present the polynomial preconditioning of the Conjugate Gradient method. So we will be solving the linear system

$$P(A)A\underline{x} = P(A)\underline{b} = \hat{b} \tag{6.9.7}$$

where $P(A)$ represents a polynomial in matrix A. The polynomial P is chosen so that the matrix $P(A)A$ when used by the Conjugate Gradient method converges faster than that of matrix A.

The preconditioned Conjugate Gradient algorithm to solve the system of linear equations is as follows,

Choose $\underline{x}^{(0)}$

$$\underline{r}^{(0)} = \underline{\hat{b}} - A\underline{x}^{(0)} \tag{6.9.8}$$

solve the auxiliary system $M\underline{z}^{(0)} = \underline{r}^{(0)}$ i.e. $\underline{z}^{(0)} = M^{-1}\underline{r}^{(0)}$ and where $M^{-1} = [I + J(I + J)]$. Let

$$\underline{p}^{(0)} = \underline{z}^{(0)} \tag{6.9.9}$$

compute,

$$\alpha_0 = \left(\underline{z}^{(0)}, \underline{r}^{(0)}\right) \tag{6.9.10}$$

$k = 1, 2, \cdots$

compute $A\underline{p}^{k-1}$

$$a_{k-1} = \frac{\alpha_{k-1}}{\left(\underline{p}^{(k-1)}, A\underline{p}^{(k-1)}\right)} \tag{6.9.11}$$

$$\underline{x}^{(k)} = \underline{x}^{(k-1)} - a_{k-1}\underline{p}^{(k-1)} \tag{6.9.12}$$

$$\underline{r}^{(k)} = \underline{r}^{(k-1)} - a_{k-1}A\underline{p}^{(k-1)} \tag{6.9.13}$$

$$\underline{z}^{(k)} = M^{-1}\underline{r}^{(k)} \tag{6.9.14}$$

IF convergence test = TRUE

then END

solve the auxiliary system $M\underline{z}^{(k+1)} = \underline{r}^{(k+1)}$

$$\alpha_k = \left(\underline{z}^{(k)}, \underline{r}^{(k)}\right) \tag{6.9.15}$$

$$b_k = \frac{a_k}{a_{k-1}} \tag{6.9.16}$$

$$\underline{p}^{(k)} = \underline{z}^{(k)} + b_k\underline{p}^{(k-1)} \tag{6.9.17}$$

Method	Time	IPS	Virtual IPS	Boundary Cell
2D-C.G.	$(10n^2 + 2n + 5) + k(10n^2 + 4n + 4)$	5	$2(n-2)$	2

Table 6.7: Area and time requirements for the 2D-P.C.G. systolic design.

6.9.1 Systolic design for the P.C.G. iterative method

The systolic design for the P.C.G. iterative method for solving a 2D discretised problem is similar to the C.G. method, except that in the P.C.G. iterative method extra matrix vector multiplication operations are performed due to the application of preconditioning. Table (6.7) give the time and area requirements of the 2D P.C.G. systolic design.

6.10 The Model problem

The large sparse system of linear equations of the form

$$A\underline{x} = \underline{b} \tag{6.10.1}$$

commonly arise when one uses the method of finite differences to approximate the solutions of partial differential equations. The self-adjoint elliptic boundary-value problem in two dimensions will be considered here. This problem can be written as,

$$F(u) = \frac{\partial}{\partial x}(a(x,y)\frac{\partial u}{\partial x}) + \frac{\partial}{\partial y}(b(x,y)\frac{\partial u}{\partial x}) + c(x,y)u = f(x,y) \tag{6.10.2}$$

for $u(x,y)$ and $(x,y) \in D$ with Dirichlet boundary conditions,

$$u(x,y) = 0 \text{ for } x \in \delta D$$

where D is the interior of the unit square and δD is the boundary of D.

where $N = n^2$ and $\underline{u}$ is the vector of unknowns $U(i,j)$ in the order given when a "natural ordering" is used. ("Natural ordering" is given by labelling the mesh points in order row by row from left to right, beginning with the bottom row). The matrix A is symmetric and block tridiagonal, and each row of A contains no more than five non-zero entries.

$$A = \begin{bmatrix} T_1 & D_1 & & 0 \\ D_1 & \ddots & \ddots & \\ & \ddots & \ddots & D_{n-1} \\ 0 & & D_{n-1} & T_n \end{bmatrix} \tag{6.10.6}$$

where the D_i's are the $n*n$ diagonal sub matrices and T_i's are $(n*n)$ tridiagonal sub matrices.

6.10.1 Model problem 1

The first model problem that will be considered is for the specified coefficients,

$$a(x,y) = b(x,y) = -1.0 \tag{6.10.7}$$

$$c(x,y) = 0.0 \tag{6.10.8}$$

that is

$$-\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} = f(x,y) \quad \text{[Poisson's Equation]} \tag{6.10.9}$$

Table (6.8) shows the results of this model problem for various methods.

6.10.2 Model problem 2

The second model problem that will be considered is defined as the specified coefficients,

$$a(x,y) = b(x,y) = \begin{cases} -10.0 & \text{in the subsquare } (\frac{2}{5}, \frac{3}{5}) * (\frac{2}{5}, \frac{3}{5}) \\ -1.0 & \text{elsewhere} \end{cases} \tag{6.10.10}$$

Method	2DRICH	C.G.	P.C.G.
size			
16	26	4	4
81	48	14	11
196	69	24	14
361	91	33	20
576	112	40	23
841	133	48	28

Table 6.8: Iteration counts for various iterative methods for model problem 1.

$$c(x,y) = 0.0 \tag{6.10.11}$$

that is

$$-\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} = f(x,y) \text{ [Poisson's Equation]} \tag{6.10.12}$$

The table (6.9) gives the results of model problem 2 for various iterative method.

6.11 Conclusions

The designs presented in this chapter are best suited for sparse matrices occurring in 2D and 3D problems. In case of overrelaxation methods, the array design requires only 4 IPS, 1 DIV-ADD-IPS and $2(n-2)$ Virtual IPS cells. So the design is capable of solving any order problem and only requires the cascading of Virtual IPS cells, programming effort and computational time for the algorithm. It is therefore of practical value to consider such a design for which the number of working processors is fixed for any size problem. Using the Programmable Virtual IPS cell, the design becomes simple, regular and free of hardware modification. The computational time for the algorithm re-

Method	2DRICH	C.G.	P.C.G.
size			
16	111	4	4
81	202	15	12
196	210	25	15
361	322	34	21
576	276	42	24
841	427	51	30

Table 6.9: Iteration counts for various iterative methods for model problem 2.

mains the same but chip area is saved by using the Programmable Virtual IPS cell.

The cell designs for the J.O.R., S.O.R., A.O.R., stationary and non-stationary second order Richardson methods are simple, but the C.G. and P.C.G. methods require special cells (two cells at the extreme right hand side see figure (6.9)) which have complex control logic due to the various mathematical operations to be performed. Some of these operations degrade the utilisation of the array due to systolic design constraint.

Chapter 7

Conclusions and suggestions for further work

The sequential architecture devised by Von Neumann was brilliantly successful in matching the technology of the 1940s and several decades there after, when the processor and memory were both expensive and precious resources. By now, however, there have been a million-fold improvements in the speed, cost, and power dissipation of processors as well as the size and speed of memories, and these no longer stand in the way of parallel computer architectures.

The advent of VLSI technology and falling cost of hardware was the original motivation behind the systolic array architectures. The design complexity in systolic arrays (having one or few different types of simple processing elements) is reduced due to their regularity, repetitiveness, expandability and modularity compared to a complex processor design. The same is true for layout design and verification. The simple control logic of the systolic arrays makes them area efficient as well, because complex control logic requires substantial VLSI area.

Data once input into the systolic array is reused and intermediate results remain in the systolic array until the final results are produced. Thus the extensive use of pipelining, multiprocessing and reusability of data in systolic arrays increases the on-chip speed.

Systolic arrays are special purpose devices. The production count is low compared to a general purpose VLSI device, hence the high cost per chip. This is one of the reasons why their use is not widespread. But in case of real time applications the product count and cost could be immaterial. The inner product step function ($y = y + ab$) when realised with a 64 bit floating point multiplier and adder may take the area equivalent to an advanced microprocessor on a chip, hence it will be difficult to build a whole systolic array on a chip. In addition, for a particular application once the array is implemented as an integrated circuit, its size and application is fixed.

As the size of systolic array is increased (i.e. the number of cells) the pin count becomes high, which is limited by the length of the chip perimeter and the current packaging technology.

This thesis has introduced some new systolic algorithms and architectures for the solution of matrix iterative methods, on linear VLSI systolic arrays.

The OCCAM simulator, developed at Loughborough University of Technology, is extensively used for the soft-systolic simulation, and the verification of the matrix iterative methods presented. By soft-systolic simulation we mean the simulation of systolic designs on a conventional uniprocessor. By verification we mean the production of expected results for given inputs. Thus, systolic architectures are considered as a network of processes, rather than processors, and the computation is data driven. All the simulators developed may possibly be implemented directly on VLSI processor arrays with minor modifications. The appendix includes the listing of the program that simulates the systolic array implementing the S.A.O.R. iterative method.

In this thesis, the aim was to develop linear systolic arrays for the matrix iterative methods.

In chapter 3 various systolic designs for the matrix vector multiplication (mvm) algorithm were explained. We then applied the matrix iterative methods to these designs for their systolic solution. It was found that some of these designs were not suitable for such application, even though they are area and time efficient.

Chapter 4 described the S.S.O.R and related iterative methods with their systolic designs. It was shown that the computational work done by the S.S.O.R. iterative method is comparable to that of the S.O.R. iterative method.

Chapter 5 presented two different design philosophies. The systolic designs for the stationary and non-stationary second order Richardson iterative method use the row wise distribution of the elements of the coefficient matrix. The IPS cell accumulates the partial results, i.e. the elements x and a enter the cell and after the multiplication their product is accumulated in y, which resides in the cell, until the final result is obtained. The rows of the matrix are delayed by one clock cycle for synchronisation with the x component. This design is well suited for the dense matrices. This accumulation of the partial results in the design allows us to schedule two iterations (one full and the other partial) of the same problem to be pipelined and hence improves the utilisation of the array. The design requires n IPS cells where n is the number of rows of the coefficient matrix. The execution time of the array for the mvm portion can be reduced if only the mvm part is required. In this particular design the cells are allowed to have enough local memory to store the rows of the coefficient matrix and the newly computed solution vector (in the boundary cells). This eliminates the host requirement after the data is downloaded into the array. The design is well suited for the iterative algorithms as there is no host communication delay and the array can run at its maximum speed. As the data is down loaded from the host the pin count can be decreased by having control logic for distributing the data to the respective cells, this might increase the chip area but is still worth consideration.

The second design philosophy is the column wise distribution of the coefficient matrix. In this case, the elements of the coefficient matrix are multiplied by the x element and the partial result is passed to the neighbouring cell until the final result is obtained. This type of design is well suited for sparse banded matrices and algorithms where the diagonal element is eliminated from the coefficient matrix for the mvm operation like Gauss-Seidel iterative method. This design requires $(2n - 1)$ IPS cells for the square coefficient matrix, or in general $(p + q - 1)$ IPS cells. The execution time for the mvm part of the array remains the same. Systolic designs for the A.O.R. and S.A.O.R. iterative methods are similarly designed on this philosophy. The A.O.R. and S.A.O.R. designs are more general compared to the S.O.R. and S.S.O.R. designs as different combinations of the parameters "ω" and "r" yield different iterative methods. As for certain problems one method might out perform the other.

In chapter 6 we presented systolic designs for the sparse systems obtained from the discretisation of 2D and 3D partial differential equations by finite difference methods and the concept of Virtual IPS cell was introduced. This design allows the expansion of the array by adding modules of programmable Virtual IPS cells to cope with different program sizes. This concept can be extended to the general systolic arrays, i.e. any size problem can be dealt with by adding these modules especially in case of iterative methods and the boundary cell can be programmed for data and control synchronisation. In this way, the restriction on the size of the systolic array can be eliminated to a certain degree and a module can have the maximum processing elements on one chip permissible by the VLSI technology. The chapter presented the systolic designs for the C.G. and P.C.G. (for the first and second order polynomial) algorithms. The main objective was to choose a suitable preconditioning technique for the C.G. method so its rate of convergence is accelerated by an order of magnitude especially when it is used to solve ill conditioned systems. The polynomial preconditioning strategy was implemented due to its high degree of inherent parallelism. The efficiency of the algorithm increases until it almost reaches its optimum. The order of the preconditioning polynomial can be increased for a better convergence rate, however, the results will not be very impressive compared to the amount of extra computational work involved.

In conclusion, the lessons learnt from the application of systolic arrays for the matrix iterative methods are that the application requires making the processing elements programmable and reconfigurable to a certain degree, at the cost of complex control logic and hence larger silicon area. The increased availability of Transputer systems and general purpose parallel computers having structures appropriate for implementing systolic array algorithms has broadened the horizons for the systolic concepts. In addition, the knowledge obtained so far in the field of systolic arrays can contribute to the organisation of the computations efficiently on a multiprocessing system

To sum up, of the systolic architectures proposed for the VLSI implementation, only few bit level systolic arrays have been developed. The systolic architecture proposed in the literature and this thesis are simulators, which provide the computational and communications requirements of the respective algorithms.

References

[1] G. S. ALMASI AND A. GOTTLIEB, *Highly Parallel Computing*, The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, 1989.

[2] M. ANNARATONE, E. ARNOULD, T. GROSS, H. T. KUNG, M. LAM, O. MENZILCIOGLU, AND J. A. WEBB, *The Warp computer: Architecture, implementation, and performance*, IEEE, Transactions on computers, 36(12) (1987), pp. 1523–1538.

[3] J. L. BAER, *Computer Systems Architecture*, Computer Science Press, 1980.

[4] F. S. BECKMAN, *The solution of linear equations by Conjugate Gradient method*, in Mathematical Methods for Digital Computers, A. Ralston and H. S. Wlif, eds., John Wiley & Sons, Inc., 1960, pp. 62–72.

[5] M. P. BEKAKOS, *A Study of Systolic Algorithms for Parallel Computers and VLSI Processor Arrays*, PhD thesis, Loughborough University of Technology, 1986.

[6] M. P. BEKAKOS AND D. J. EVANS, *The exposure and exploitation of parallelism on fifth generation computer systems*, Parallel Computing, (1985), pp. 425–442.

[7] M. BERZINS, T. F. BUCKLEY, AND P. M. DEW, *Systolic matrix iterative algorithms*, in Parallel Computing 83, M. Feilmeier, G. Joubert, and U. Schendel, eds., Elsevier Science Publishers, 1984, pp. 483–488.

[8] J. C. BROWNE, *Parallel architectures for computer systems*, Physics today, 37(5) (1984), pp. 28–35.

[9] R. L. BURDEN AND J. D. FAIRES, *Numerical Analysis*, Prindle, Weber and Schmidt Publishers, 1989.

[10] D. CASASENT, *Acoustooptic linear algebra processors: architecture, algorithms and applications*, Proceedings of the IEEE, 72(7) (1984), pp. 831–849.

[11] V. CONRAD AND Y. WALLACH, *A faster S.S.O.R. algorithm*, Num. Math., 27 (1977), pp. 371–372.

[12] P. M. DEW, L. J. MANNING, AND K. MCEVOY, *A tutorial on systolic array architectures for high performance processors*, Tech. Rep. 205, University of Leeds, 1986.

[13] P. H. ENSLOW, *Multiprocessing organization-A survey*, ACM computing Surveys, 9(1) (1977), pp. 103–129.

[14] D. J. EVANS, *The use of preconditioning in iterative methods for solving linear equations with symmetric positive definite matrices*, Journal of the institute of Mathematics and its applications, 4 (1968), pp. 295–314.

[15] ——, ed., *Systolic Algorithms*, vol. 3 of Topics in Computer Mathematics, Gordon and Breach Science Publishers, 1991.

[16] ——, *A survey of preconditioning strategies*, Tech. Rep. 754, University of Technology Loughborough, Parallel Algorithms Research Centre, January 1993.

[17] D. J. EVANS AND M. GUSEV, *Implementation of folding transformations on linear systolic and VLSI processor arrays*, Tech. Rep. 650, University of Technology Loughborough, Department of Computer Studies, Nov 1991.

[18] D. J. EVANS AND S. A. HAIDER, *Systolic arrays for the matrix iterative methods*, in Parallel Computing and Transputer Applications, M. Valero, E. Onate, M. Jane, J. L. Larriba, and B. Suarer, eds., IOS Press, 1992, pp. 216–225. Technical report 708, Parallel Algorithm Research Centre, Loughborough University of Technology.

[19] D. J. EVANS, S. A. HAIDER, AND M. M. MARTINS, *A systolic design for the M.S.O.R. iterative method for the solution of linear equations*, Tech. Rep. 623, University of Technology Loughborough, Department of Computer Studies, July 1991.

[20] D. J. EVANS AND K. G. MARGARITIS, *A re-usable systolic array for matrix-vector multiplication*, International Journal of Computer Mathematics, 41 (1991), pp. 19–30.

[21] T. Y. FENG, *Parallel processors and processing*, ACM computing Surveys, 9(1) (1977), pp. 1–2.

[22] A. L. FISHER AND H. T. KUNG, *Special-purpose VLSI architectures: General discussions and a case study*, in VLSI and Modern Signal Processing, S. Y. Kung, H. J. Whitehouse, and T. Kailath, eds., Prentice-Hall, 1985, pp. 153–169.

[23] M. J. FLYNN, *Some computer organisations and their effectiveness*, IEEE Transactions on Computers, 21(9) (1972), pp. 948–960.

[24] M. J. FLYNN, A. PODVIN, AND K. SHIMIZU, *A multiple instruction stream with shared resources*, in Parallel Processor Systems, Technologies and Applications, L. C. Hobbs and et al., eds., Spartan Books, Washington, 1969, pp. 251–286.

[25] G. E. FORSYTHE AND W. R. WASOW, *Finite-Difference Methods for Partial Differential Equations*, John Wiley & Sons, INC., New York, 1960.

[26] J. A. B. FORTES AND B. W. WAH, *Systolic arrays - from concept to implementation*, IEEE Computer, 20(7) (1987), pp. 12–17.

[27] M. J. FOSTER AND H. T. KUNG, *The design of special purpose vlsi chips*, IEEE, Computer, 13(1) (1980), pp. 26–40.

[28] G. C. FOX AND S. W. OTTO, *Algorithms for concurrent processors*, Physics today, 37(5) (1984), pp. 53–59.

[29] S. T. FRANKEL, *Convergence rules of iterative treatments of partial differential equations*, (M.T.A.C.) Mathematics of Computation, 4 (1950), pp. 65–75.

[30] G. H. GOLUB AND C. F. V. LOAN, *Matrix Computations*, The John Hopkins University Press, Maryland, 1989.

[31] G. H. GOLUB AND R. S. VARGA, *Chebyshev semi-iterative methods, successive over-relaxation iterative methods, and second-order Richardson iterative methods*, Numer. Math., Part I and II 3 (1961), pp. 147–168.

[32] M. GUSEV, *Processor Array Implementation of Systems of Affine Recurrence Equations for Digital Signal Processing*, PhD thesis, University of Ljubljani, 1992.

[33] A. HADJIDIMOS, *Accelerated overrelaxation method*, Mathematics of Computation, 32 (1978), pp. 149–157.

[34] L. A. HAGEMAN AND D. M. YOUNG, *Applied Iterative Analysis*, vol. Computer Science and Applied Mathematics, Academic Press Inc., New York, 1981.

[35] S. A. HAIDER AND D. J. EVANS, *Systolic simulation of second order Richardson method for the iterative solution of linear systems*, Tech. Rep.

672, University of Technology Loughborough, Department of Computer Studies, Feb 1992.

[36] W. HANDLER, *The impact of classification schemes on computer architecture*, in Proc. 1977 Int Conf. on Parallel Processing, 1977, pp. 7–15.

[37] L. S. HAYNES, R. L. LAU, D. P. SIEWIOREK, AND D. W. MIZELL, *A survey of highly parallel computing*, IEEE, Computer, 15(1) (1982), pp. 9–24.

[38] J. L. HENNESSY AND D. A. PATTERSON, *Computer Architecture A Quantitative Approach*, Morgan Kaufmann Publishers, San Mateo, California, 1990.

[39] M. R. HESTENES AND E. L. STIEFEL, *Methods of conjugate gradients for solving linear systems*, Nat. Bur. Std. J Res., (1952), pp. 409–436.

[40] C. A. R. HOARE, *Communicating sequential processes*, Communications of the ACM, 21(8) (1978), pp. 666–677.

[41] K. HWANG AND F. A. BRIGGS, *Computer Architecture and Parallel Processing*, McGraw-Hill, Inc., 1984.

[42] B. KOLMAN, *Elementary Linear Algebra*, Macmillan Publishing Company, New York, 1986.

[43] H. T. KUNG, *The structure of parallel algorithms*, Advances in COMPUTERS, 19 (1980), pp. 65–112.

[44] ——, *Notes on VLSI computation*, in Parallel processing systems, D. J. Evans, ed., Cambridge University Press, 1982, pp. 339–356.

[45] ——, *Why systolic architectures*, IEEE, Computer, 15(1) (1982), pp. 37–46.

[46] H. T. KUNG AND C. E. LEISERSON, *Systolic arrays (for VLSI)*, in Sparse Matrix Symposium, vol. 32, Academic Press Inc., 1978, pp. 256–282.

[47] S. Y. KUNG, *On supercomputing with systolic/wavefront array processors*, Proceedings of the IEEE, 72 (1984), pp. 867–884.

[48] C. E. LEISERSON, *Systolic and semisystolic design*, in IEEE international conference on computer design: VLSI in computers, 1983, pp. 627–632.

[49] S. LIPSCHUTZ, *Theory and Problems of Linear Algebra*, McGraw-Hill, Inc., New York, 1991.

[50] D. G. LUENBERGER, *Linear and Nonlinear Programming*, Addison Wesley Publishing Company, Inc., Reading, Massachusetts, 1984.

[51] K. G. MARGARITIS, *A Study of Systolic Algorithms for VLSI Processor Arrays and Optical Computing*, PhD thesis, Loughborough University of Technology, 1987.

[52] K. G. MARGARITIS AND D. J. EVANS, *Systolic networks for iterative methods for linear equations using cyclic reduction.*, Computers and Artificial Intelligence, 9(5) (1990), pp. 503–520.

[53] M. MARTINS, *On the accelerated overrelaxation iterative method for linear systems with strictly diagonally dominant matrices*, Mathematics of Computation, 35 (1980), pp. 1269–1273.

[54] M. M. MARTINS, *A note on the convergence of the M.S.O.R. method*, Linear Algebra and Applications., 141 (1990), pp. 223–226.

[55] C. MEAD AND L. CONWAY, *Introduction to VLSI systems*, Addison Wesley Publishing Company, Inc., Reading, Massachusetts, 1980.

[56] G. M. MEGSON, *An introduction to systolic algorithm design*, Oxford University Press, Oxford, 1992.

[57] G. M. MEGSON AND D. J. EVANS, *Triangular systolic arrays for matrix product and factorisation*, International Journal of Computer Mathematics, 25 (1988), pp. 321–343.

[58] R. E. MILLER AND J. COCKE, *Configurable computers: a new class of general purpose machines*, in International symposium on theoretical programing, A. P. Ershov and V. A. Nepomniaschy, eds., Springer-Verlag, 1972, pp. 285–298. In Lecture notes in computer science (5), edited by G. Goss et al. in 1974.

[59] A. R. MITCHELL AND D. F. GRIFFITHS, *The finite difference method in partial differential equations*, John Wiley and Sons, New York, 1980.

[60] D. I. MOLDOVAN, *On the design of algorithms for VLSI systolic arrays*, Proceedings of the IEEE, 71(1) (1983), pp. 113–120.

[61] R. V. NORTON, *The solution of linear equations by Gauss-Seidel method*, in Mathematical Methods for Digital Computers, A. Ralston and H. S. Wlif, eds., John Wiley & Sons, Inc., 1960, pp. 56–61.

[62] P. QUINTON, *The systematic design of systolic arrays*, Tech. Rep. 193, IRISA, March 1983.

[63] P. QUINTON, P. JANNAULT, AND P. GACHET, *A new matrix multiplication systolic array*, in Parallel Algorithms and Architectures, M. Cosnard, P. Quinton, Y. Robert, and M. Tchuente, eds., vol. 32, North Holland, 1986, pp. 259–268.

[64] P. QUINTON AND Y. ROBERT, *Systolic Algorithms and Architectures*, Prentice Hall Inc., 1989.

[65] S. S. REDDI AND E. A. FEUSTEL, *A conceptual framework for computer architecture*, ACM computing Surveys, 8(2) (1976), pp. 277–300.

[66] J. K. REID, *On the method of Conjugate Gradients for solution of large sparse systems of linear equations*, Large sparse sets of linear equations, Academic Press Inc., 1971.

[67] J. SHELDON, *On the numerical solution of elliptic difference equations*, Math. Tables Aids Comput, 9 (1955), pp. 101–112.

[68] J. E. SHORE, *Second thoughts of parallel processing*, Comput. Elect. Eng., 1 (1973).

[69] D. B. SKILLICORN, *A taxonomy for computer architectures*, IEEE Computer, 20(11) (1988), pp. 46–57.

[70] G. D. SMITH, *Numerical Solution of Partial Differential equations: finite difference methods*, Oxford University Press, Walton Street, Oxford, 1985.

[71] L. SNYDER, *Introduction to the configurable, highly parallel computer*, IEEE, Computer, 15(1) (1982), pp. 47–56.

[72] V. P. SRINI, *Architectural comparison of data flow systems*, IEEE Computer, 9(3) (1986), pp. 68–88.

[73] E. L. STIEFEL, *Kernel polynomials in linear algebra and their numerical applications*, Nat. Bur. Standards Appl. Math. Series, 49 (1958), pp. 1–22.

[74] H. S. STONE, *Introduction to Computer Architecture*, Academic Press Inc., New York, 1975.

[75] R. SUROS AND E. MONTAGNE, *Optimizing systolic networks by fitting diagonals*, Parallel Computing, 4 (1987), pp. 167–174.

[76] A. TREW AND G. WILSON, *Past, Present, Parallel: A Survey of Available Parallel Computer systems*, Springer-Verlag, London, 1991.

[77] P. S. TSENG, *Iterative sparse linear system solvers on WARP*, Int. Conf. Parallel Processing, 1 (1988), pp. 32–38.

[78] R. S. VARGA, *Matrix iterative analysis*, Prentice Hall, Englewood Cliffs, N. J., 1962.

[79] E. L. WACHSPRESS, *Iterative Solution of Elliptic Systems And Applications to the Neutron Diffusion Equations of Reactor Physics*, Prentice-Hall, Inc., Englewood Cliffs, N. J., 1966.

[80] J. R. WESTLAKE, *A Handbook of Numerical Matrix Inversion and Solution of Linear Equations*, John Wiley & Sons, Inc., New York, 1968.

[81] J. H. WILKINSON, *The algebraic Eigenvalue Problem*, Oxford University Press, Oxford, 1965.

[82] D. M. YOUNG, *Iterative Solution of Large Linear Systems*, vol. Computer Science and Applied Mathematics, Academic Press Inc., New York, 1971.

Appendix A

Systolic simulator for the S.A.O.R. iterative method

A.1 Main program

```
-- This program simulates the systolic design for the S.A.O.R.
-- iterative method for p=q=3 and n is even.
--              ********************
--              *     S.A.O.R.     *
--              ********************

EXTERNAL PROC num.to.screen(VALUE n):
EXTERNAL PROC str.to.screen(VALUE s[]):
EXTERNAL PROC fp.num.to.screen(VALUE FLOAT n):
EXTERNAL PROC fp.num.to.screen.f(VALUE FLOAT n, VALUE w,d):
EXTERNAL PROC fp.num.from.keyboard(VAR FLOAT n):
EXTERNAL PROC getdata(VAR FLOAT xv[],av[],bv[],r,w,VALUE n):
EXTERNAL PROC errortest(VAR FLOAT val[],val1[],VAR con,VALUE FLOAT err,VALUE n):
EXTERNAL PROC invertdata(VAR FLOAT aa[], xx[], bb[], xxx[], VALUE n):
EXTERNAL PROC putdata(VAR itera, VALUE FLOAT x[], VALUE flag, n):
EXTERNAL PROC cell(CHAN xin, yin, xout, yout, a):
EXTERNAL PROC celldia(CHAN ci1, co1, ci2, ci3, ci4, bn, an, rn, wn, ngs):

DEF n = 8:
DEF maxiterations = 100:

LIBRARY VAR  total.time:
LIBRARY VAR flag, conv, k, j, iterations:
```

```
LIBRARY VAR FLOAT aval[n*n], xval[n], bval[n], yin[n]:
LIBRARY VAR FLOAT temp, error, rmega, Lsx[(n*n)+1]:
LIBRARY VAR FLOAT omega, dummy, yval[n+1], xval.1[n], Lx[(n*n)+1]:
LIBRARY CHAN x[n+1], y[n+1], a[n+1]:
LIBRARY CHAN x.1[n+1], y.1[n+1], a.1[n+1]:
LIBRARY CHAN x.2[n+1], y.2[n+1], a.2[n+1]:
LIBRARY CHAN ain, win, bin, xin, rin, newgss:

-- *********************************************************
-- Provides a,w,r,b,i and x vectors to the DIV ADD IPS cell *
-- *********************************************************

PROC source(CHAN d,w,r,h,m,VAR FLOAT a[],b[],x[],VALUE FLOAT om,rm,VALUE del)=
  VAR k, j, switch:
  SEQ
    k := 1
    j := 1
    switch := true
    SEQ i = [1 FOR total.time]
      SEQ
        IF
          i < (del+1)
            PAR
              d ! 1.0
              m ! 0.0
              h ! 0.0
              w ! 0.0
              r ! 0.0
        IF
          (( k < (n+1) ) OR ( j < (n+1)))
            IF
              switch
                SEQ
                  PAR
                    d ! a[((k-1)*n)+(k-1)]
                    m ! x[k-1]
                    h ! b[k-1]
                    w ! om
                    r ! rm
                    k := k + 1
                    switch := false
              true
                SEQ
                  PAR
                    d ! 1.0
                    m ! 0.0
                    h ! 0.0
                    w ! 0.0
                    r ! 0.0
                  j := j + 1
                  switch := true:

-- ************************************************************
-- Provides the lower and upper matrices to the desired cells *
-- ************************************************************

PROC sourcea(CHAN a0, a1, VAR FLOAT a[], VALUE delay, set) =
  VAR k, j, switch, delay1,dummy,lim1,lim2,kinit,jinit:
  SEQ
    switch := true
    IF
      set
        SEQ
          delay1 := delay
          dummy := delay + 4
          k := 1
          j := 1
          kinit := k-1
          jinit := j
      true
        SEQ
          delay1 := delay + 3
          dummy := 2
          k := 2
          j := 3
          kinit := (-1)*k
          jinit := (-1)*j
    SEQ i = [1 FOR delay1]
      SEQ
        PAR
          a0 ! 0.0
          a1 ! 0.0
    SEQ i = [1 FOR (total.time-(delay1+dummy))]
      SEQ
        IF
          switch
            SEQ
              IF
                set
                  PAR
                    a0 ! ((-1.0)*(a[(((k-1)*n)+(k+kinit))]))
                    a1 ! 0.0
                true
                  PAR
                    a1 ! ((-1.0)*(a[(((k-1)*n)+(k+kinit))]))
                    a0 ! 0.0
              k := k + 1
              switch := false
          true
            SEQ
              IF
                set
                  PAR
                    a0 ! 0.0
                    a1 ! ((-1.0)*(a[(((j-1)*n)+(j+jinit))]))
                true
                  PAR
                    a1 ! 0.0
                    a0 ! ((-1.0)*(a[(((j-1)*n)+(j+jinit))]))
              j := j + 1
              switch := true
    SEQ i = [1 FOR dummy]
      SEQ
        PAR
          a0 ! 0.0
          a1 ! 0.0:

-- ***********************
--  Collects the garbage *
-- ***********************

PROC sinkl.1(CHAN xout1) =
  VAR FLOAT dummy:
  SEQ i = [1 FOR total.time]
    xout1 ? dummy:

-- ********************************************************
-- used to provide the pre computed upper or lower matrix *
-- ********************************************************

PROC source.precomputed.upper(CHAN output) =
  SEQ
    SEQ i = [1 FOR total.time]
      SEQ
        output ! (Lsx[i]):

-- **************************************
-- Collects the newly computed solution *
-- **************************************

PROC sink1(CHAN xout, VALUE delay) =
  VAR k, set:
  VAR FLOAT xin:
  SEQ
    k:= 0
    set := true
    SEQ i = [1 FOR total.time]
      SEQ
        xout ? xin
        IF
          i > (delay +3)
            SEQ
              IF
                set
                  SEQ
                    xval.1[k] := xin
                    k:= k+1
                    set := false
                true
                  set := true:

-- **************************************************************
-- This arranges the new solution to be repumped in right order *
-- **************************************************************

PROC nux(VALUE ran1, ran2) =
  SEQ
    j:= ran2
    k:= ran1
    SEQ i= [1 FOR (n/2)]
      SEQ
        temp := Lx[k]
        Lx[k] := Lx[j]
        Lx[j] := temp
        j:=j-2
        k:=k+2
    SEQ i= [1 FOR total.time]
      SEQ
        Lsx[i] := Lx[i]
        Lx[i] := 0.0:

-- ************************************************
-- Depending on the arguments provides the x or y *
-- ************************************************
```

```
PROC source.x.or.y(CHAN x1, VAR FLOAT x[], VALUE delay, flag) =
  VAR k, switch, dummy:
  SEQ
    IF
      flag
        SEQ
          k := 2
          dummy := (total.time-((delay+(n*2))-3))
      true
        SEQ
          k := 1
          dummy := (total.time-((delay+(n*2))-1))
    switch := true
    SEQ i = [1 FOR delay]
      x1 ! 0.0
    SEQ i = [1 FOR (total.time-(delay+dummy))]
      SEQ
        IF
          switch
            SEQ
              x1 ! x[k-1]
              k := k + 1
              switch := false
          true
            SEQ
              x1 ! 0.0
              switch := true
    SEQ i = [1 FOR dummy]
      x1 ! 0.0:

-- ********************************************************
-- The routines lower, upper, lower2 and central represent *
-- ********************************************************

PROC lower(VALUE delay1, flag1, delay2, flag2) =
  SEQ
    PAR
      source.x.or.y(y.1[0], yval, delay1, flag1)
      sourcea(a.1[0], a.1[1], aval, delay2,flag2)
      PAR j = [1 FOR 2]
        cell(x.1[j], y.1[j-1], x.1[j-1], y.1[j], a.1[j-1])
      sinkl.1(x.1[0]):

PROC lower2(VALUE delay1, flag1, delay2, flag2, delay3, flag3) =
  SEQ
```

```
    PAR
      source.x.or.y(x.2[2], xval, delay1, flag1)
      source.x.or.y(y.2[0], yval, delay2, flag2)
      sourcea(a.2[0], a.2[1], aval, delay3,flag3)
      PAR j = [1 FOR 2]
        cell(x.2[j], y.2[j-1], x.2[j-1], y.2[j], a.2[j-1])
      sinkl.1(x.2[0]):

PROC upper(VALUE delay1, flag1, delay2, flag2, delay3, flag3) =
  SEQ
    PAR
      source.x.or.y(x[2], xval, delay1, flag1)
      source.x.or.y(y[0], yval, delay2, flag2)
      sourcea(a[0], a[1], aval, delay3, flag3)
      PAR j = [1 FOR 2]
        cell(x[j], y[j-1], x[j-1], y[j], a[j-1])
      sinkl.1(x[0]):

PROC central(VALUE delay1, delay2) =
  SEQ
    PAR
      source(ain,win,rin,bin,xin,aval,bval,xval,omega,rmega,delay1)
      celldia(y[2],x.1[2],y.1[2],xin,y.2[2],bin,ain,rin,win,newgss)
      sink1(newgss,delay2):

-- *************************
-- Initialize the variables *
-- *************************

PROC setup =
  SEQ
    total.time := (2*n)+3
    iterations := 1
    error := 0.00001
    conv := true
    SEQ i= [0 FOR n]
      SEQ
        xval.1[i] := 0.0
        yval[i] := 0.0:

-- ***************************************************************
-- Sets up the new solution in correct order as the delays change *
-- for 2nd and successive iterations                              *
-- ***************************************************************

PROC newsetup  =
  SEQ
    total.time := (2*n)
    SEQ i = [1 FOR total.time]
      SEQ
        Lsx[i] := Lsx[i+3]:

-- ******************************************************
-- Check if the iterations have exceded the max allowed *
-- ******************************************************

PROC max.iter.check(VAR ite, flg, con, VALUE maxite) =
  SEQ
    IF
      ite > maxite
        SEQ
          flg := true
          con := false:

-- *******************************************************
-- This routine performs the 1st iteration of the method *
-- *******************************************************

PROC iter1 =
  PAR
    lower(1, false, 1, false)
    lower2(4, false, 1, false, 1, false)
    upper(0, true, 1, false, 1, true)
    central(3,1):

-- *********************************************************
-- This routine performs the 2nd and successive iterations *
-- *********************************************************

PROC iter2 =
  PAR
    lower(0, true, -2, false)
    lower2(1, false, 0, true, -2, false)
    central(0,-2)
    source.precomputed.upper(y[2]):

-- *******************************************************
-- This routine perfoms various tests in the host system *
-- *******************************************************

PROC hostcomp(VALUE arg1, arg2, flag2) =
  SEQ
```

```
    errortest(xval,xval.1,conv,error,n) -- for convergence
    SEQ
      IF
        conv = true  -- more iterations to be performed
          SEQ
            iterations := iterations+1
            invertdata(aval, xval, bval, xval.1, n)
            nux(arg1,arg2)
            max.iter.check(iterations, flag, conv, maxiterations)
    IF
      flag2  -- if 1st iteration
        newsetup : -- setup of data and delays for 2nd iteration

--  ********
--  * main *
--  ********

SEQ
  getdata(xval,aval,bval,rmega,omega,n)
  setup
  iter1
  hostcomp(4, (2*n)+2, true)
  WHILE conv
    SEQ
      iter2
      hostcomp(1, (2*(n-1))+1, false)
  putdata(iterations, xval, flag, n)
```

A.2 Cell architecture

```
EXTERNAL VAR total.time:
EXTERNAL VAR FLOAT Lx[]:

-- *********************************
-- The DIV ADD IPS Cell definition *
-- *********************************

LIBRARY PROC celldia(CHAN ci1, co1, ci2, ci3, ci4, bn, an, rn, wn, ngs) =
  VAR FLOAT x1, x2, x3, x4, b, a, w, r, tmpx1:
  SEQ
    tmpx1 := 0.0
    SEQ i = [1 FOR total.time]
      SEQ
        PAR
          ci1 ? x1
          ci2 ? x2
          ci3 ? x3
          ci4 ? x4
          bn ? b
          an ? a
          wn ? w
          rn ? r
          co1 ! tmpx1
          ngs ! tmpx1
    SEQ
      SEQ
        tmpx1 := ((((((x1+b)*w)+((w-r)*x4))+(r*x2))/a)+((1.0-w)*x3))
        Lx[i] := x2:

-- *********************
-- The Basic IPS Cell  *
-- *********************

LIBRARY PROC cell(CHAN xin, yin, xout, yout, a) =
  VAR FLOAT x, y, xtmp, ytmp, p:
  SEQ
    xtmp := 0.0
    ytmp := 0.0
    SEQ i = [1 FOR total.time]
      SEQ
        PAR
          xin ? x
          yin ? y
          a ? p
  xout ! xtmp
  yout ! ytmp
ytmp := y + (p*x)
xtmp := x:
```

A.3 Get data routine

```
EXTERNAL PROC num.to.screen(VALUE n):
EXTERNAL PROC str.to.screen(VALUE s[]):
EXTERNAL PROC fp.num.to.screen(VALUE FLOAT n):
EXTERNAL PROC fp.num.from.chan(CHAN c, VAR FLOAT n):
EXTERNAL PROC open.file(VALUE path.name[], access[], CHAN io.chan):

-- ********************************************************************
-- This routine reads data from the key board  or a file and stores   *
-- it in the desired format.                                          *
-- ********************************************************************

LIBRARY PROC getdata(VAR FLOAT xv[],av[],bv[], r,w, VALUE n) =
  CHAN chfin:
  SEQ
    str.to.screen(" The order of the matrix A is ")
    num.to.screen(n)
    screen!'*n'
    str.to.screen(" Input A matrix ")
    screen!'*n'
    open.file("data", "r", chfin)
    SEQ i=[1 FOR n]
      SEQ
        screen!'*n';'['
        SEQ j=[1 FOR n]
          SEQ
            fp.num.from.chan(chfin, av[(((i-1)*n)+j)-1])
            fp.num.to.screen(av[(((i-1)*n)+j)-1])
            screen!'*s'
        screen!']'
      screen!'*n'
    screen!'*n'
    str.to.screen(" Input B  vector ")
    screen!'*n'
    screen!'*n';'['
    SEQ i=[0 FOR n]
      SEQ
        fp.num.from.chan(chfin, bv[i])
        fp.num.to.screen(bv[i])
        screen!'*s'
    screen!']';'*n'
    screen!'*n'
    str.to.screen(" Input initial guess vector ")
    screen!'*n'
    screen!'*n';'['
SEQ i=[0 FOR n]
  SEQ
    fp.num.from.chan(chfin, xv[i])
    fp.num.to.screen(xv[i])
    screen!'*s'
screen!']';'*n'
screen!'*n'
str.to.screen(" Input omega =  ")
SEQ
  fp.num.from.chan(chfin, w)
  fp.num.to.screen(w)
  screen!'*n'
screen!'*n'
str.to.screen(" Input acceleration factor r = ")
SEQ
  fp.num.from.chan(chfin, r)
  fp.num.to.screen(r)
  screen!'*n':
```

A.4 Convergence test

```
-- **********************************************************
-- This routine tests for convergence for a given tolerance *
-- **********************************************************


LIBRARY PROC errortest(VAR FLOAT val[],val1[],VAR con,VALUE FLOAT err,VALUE n)=
  VAR FLOAT absx, absy, dummy, test:
  SEQ
    dummy := 0.0
    con := false
    SEQ i = [0 FOR n]
      SEQ
        absx := val[i]
        IF  -- if the number is -ve take its absolute value
          absx < dummy
        SEQ
          absx := ((-1.0)*(absx))
        absy := val1[i]
        IF
          absy < dummy
            absy := ((-1.0)*(absy))
        IF
          (absx - absy) < dummy
            test := ((-1.0)*(absx - absy))
          true
            test := (absx - absy)
        IF
          (test > err)
            SEQ
              con := true:
```

A.5 Reverse the system

```
-- ************************************************************
-- Inverts the data in the A matrix and the vectors b, x etc. *
-- ************************************************************


LIBRARY PROC invertdata(VAR FLOAT aa[], xx[], bb[], xxx[], VALUE n)=
  VAR FLOAT tempa, tempb, tempx:
  VAR k:
  SEQ
    SEQ i = [0 FOR n]
      SEQ
        xx[i] := xxx[i]
        xxx[i] := 0.0
    k:= n-1
    SEQ i = [0 FOR (n/2)]
      SEQ
        tempb := bb[k]
        tempx := xx[k]
        bb[k] :=  bb[i]
        xx[k] :=  xx[i]
        bb[i] := tempb
        xx[i] := tempx
        k := k-1
    k:= (n*n) -1
    SEQ i = [0 FOR (n*n)/2]
      SEQ
        tempa := aa[k]
        aa[k] := aa[i]
        aa[i] := tempa
        k := k-1:
```

A.6 Display the results

```
EXTERNAL PROC num.to.screen(VALUE n):
EXTERNAL PROC str.to.screen(VALUE s[]):
EXTERNAL PROC fp.num.to.screen(VALUE FLOAT n):

-- *********************************************************
-- This routine prints the solution on the standard output *
-- *********************************************************

LIBRARY PROC putdata(VAR itera, VALUE FLOAT x[], VALUE fl, n) =
  SEQ
    IF
      fl
        SEQ
          screen!'*n'
          str.to.screen("The Solution was not found because: *n")
          str.to.screen("The number of iterations has exceeded ")
          num.to.screen(itera-1)
          screen!'*n'
          str.to.screen("The Solution so far is *n")
      TRUE
        SEQ
          screen!'*n'
          str.to.screen("The number of iterations performed = ")
          num.to.screen(itera)
          screen!'*n'
         str.to.screen("The Solution Vector is *n")
    SEQ
      SEQ i = [0 FOR n]
        SEQ
          fp.num.to.screen(x[i])
          screen!'*n'
      screen!'*n':
```