NOVEL ALGORITHMS FOR THE

SOFT-SYSTOLIC PARADIGM

BY

GRAHAM MARTIN MEGSON, B.Sc.(HONS.)

A Doctoral Thesis

submitted in partial fulfilment of the requirements

for the award of Doctor of Philosophy

of the Loughborough University of Technology

1987.

Supervisor: PROFESSOR D.J. EVANS, Ph.D.,D.Sc.
Department of Computer Studies.

DECLARATION

I declare that this thesis is a record of research work carried out by me, and that the thesis is of my own composition. I also certify that neither this thesis nor the original work contained therein has been submitted to this or any other institution for a higher degree.

GRAHAM MARTIN MEGSON.

DEDICATED TO

My Wife and Love Helena,

for her constant support during

the course of this work.

ACKNOWLEDGEMENTS

The author wishes to express his thanks to Professor D.J. Evans for his guidance, suggestions and advice throughout the preparation of this thesis and in the context of research generally.

The author also acknowledges the Science and Engineering Research Council (S.E.R.C.) for their financial support.

Thanks also to my parents for giving me the incentive to start and complete this project.

Finally, thanks to Mr. R.P. Stallard for his constant revision of the Loughborough OCCAM compiler and supplying the documentation in Appendix II.

APOTHEGM

Quarendo invenietis

"By seeking, you will discover"

NOVEL ALGORITHMS FOR THE SOFT SYSTOLIC PARADIGM

G.M. Megson

ABSTRACT

The soft-systolic paradigm, a framework of semi-formal axioms and heuristics, is introduced and used to develop a variety of novel systolic arrays and architectures for new and innovative numerical algorithms.

Designs presented include systolic arrays based on the so-called Quadrant Interlocking (QI) methods, Systolic Preconditioning strategies, Incomplete methods (for matrix factorisation and triangularisation), Table generation (including Interpolation, Extrapolation, Simplex and Assignments problems) and Systolic Group Explicit (GE) methods for Partial Differential Equations (PDEs).

A number of themes tie the different designs together within the new paradigm and illustrate its extensions over traditional methods of array construction. For instance, systolic arrays are represented as OCCAM programs whose execution and output implicitly confirm the correctness of the design. This adds flexibility, allowing formal (explicit) verification by already well developed program proof techniques. The systolic incomplete schemes allow optimal array structure to dictate the numerical method, while the QI methods use block partitioning strategies to improve array efficiency. The Table and GE arrays expose the close relationship between the algorithmic and geometric form of a problem and the use of templates or computational molecules (in the case of PDE's) for area efficient designs. While, preconditioning arrays illustrate that theoretically well-founded additional computation in the form of systolic preprocessing elements

can be used to produce overall area reduction with an accompanying speed-up of array operation.

The problems considered provide examples in which existing systolic arrays can be considered deficient in some respect, be it area, computation or efficiency and, for the most part these new methods and arrays provide some improvement.

Finally, a more general architecture is discussed in which systolic algorithms are considered purely as programs. A basic specification language is defined and a primitive simulator developed and tested with elementary examples. Hence bridging the gap between the soft-systolic paradigm and more general notions of parallel computation.

CONTENTS

PART I

INTRODUCTORY CONCEPTS AND DEFINITIONS

CHAPTER 1

INTRODUCTION:

THE ORIGINS OF SYSTOLIC ARRAYS

"A morsel of genuine history is a thing so rare as to be always valuable".

Thomas Jefferson.

The initial revelation of H.T. Kung and C.E. Leiserson (Circa 1979) indicated that highly parallel architectures and algorithms for computationally intensive and regular problems, could be implemented by area efficient, cost effective circuits; in a manner which permitted regular communication geometries and replicatable sub-circuits (or cells). Since then a tremendous volume of research in a variety of areas from signal and image processing, pattern matching, linear algebra, and recurrence evaluation, to graph algorithms, sorting, searching, and on to real time priority queues and relational data operations has occurred. Supporting the claim in Leiserson [81] that systolic systems had the desirable properties necessary to capture concepts of parallelism, pipelining and interconnection structures, in a unified framework which is known today as the systolic paradigm.

The relative youthfulness of the subject means that researchers entering the field, have more or less a free hand in the way they develop their ideas. A rough skeletal framework for systolic computation does exist, but sometimes it is difficult to define, and depends largely upon one's interests as to how restrictive it appears when encountered. H.T. Kung [80], Leiserson [81], Mead and Conway [79], Dew, Manning, Mcevoy [86] provide excellent introductory material and references, however one finds that a few simple designs have been developed before a more serious study is conducted. A version of the framework relevant to our discussions is given in Chapter 3.

More tangible and difficult to grasp are the philosophical and pragmatic threads that bind the framework together. Although, once understood a guiding philosophy provides exciting research opportunities, and denying or extending key assumptions to provide broader based

systems of design rules with wider applications is a worthy objective. By introducing the soft-systolic paradigm I feel we have moved some way towards this aim.

The aim of the current chapter however, is two-fold. Firstly, a brief historical review of the origins of systolic arrays is given, which is a fitting introduction to basic principles and requirements complementing the more detailed definitions in Chapter 3. Secondly, the structure of the thesis is outlined in relation to the themes followed throughout the work.

1.1 ORIGINS OF SYSTOLIC ARRAYS

We shall begin the journey into the soft-systolic paradigm with an informal analogy between the human body and systolic systems. The analogy is useful for two reasons, firstly it allows identification of the main characteristics of systolic computation in a simple and intuitive way, and second such a simple relation also implants in the mind a vivid picture of the mechanics involved, which can be embellished to support more complex discussions. Furthermore, the environment in which we live is inherently parallel in nature and our activities in some respects model the processes involved in systolic computation. In fact, the development of parallel systems in general, has been motivated by the difficulties encountered in implementing so-called Natural problems from the real world, on sequential (Von-Neumann) machines.

Consider the human body broken down to a simple circulatory and control system consisting of nerves, arteries and major organs. Each organ performs a unique function in parallel with the rest which is

necessary to keep the whole body alive. Organs co-operate and in a sense communicate with each other making the body a parallel system based on communication. The whole system is co-ordinated by a controller (the brain) and a pumping mechanism (the heart) driving blood around the circulatory system. Under normal circumstances the brain issues controls as nerve impulses and blood is pumped rhythmically at correct intervals and arrives at the required organs when necessary. The simpler functions of the organs complement each other to achieve the more complex goal of healthy life for the body. Taking the analogy further, we can dissect each organ which reveals that it too is a parallel system constructed from a vast number of simple cells, co-operating and communicating to achieve the function of the individual organ. Now, if nerve impulses are sent incorrectly or at wrong times, or blood is pumped erratically then an organ fails to produce the correct response. Alternatively individual cells within an organ can develop faults resulting in sickness of the organ, then the whole body, and ultimately death. Furthermore, the body includes built-in redundancy among the vital organs, so if one organ fails (e.g. kidney, lung) a second identical organ can cover for it but at a cost of reduced efficiency to the body. The process of failures may continue in some cases until faults are so numerous that the body dies.

Now, consider systolic systems, the above discussion illustrates all the essential features of systolic computation. The body becomes a machine, nerves and arteries, control and data paths, while, major organs are systolic array components performing dedicated or special purpose computations. The heart and brain form a host computer which orchestrates calculation according to control and data signals pumped

continuously around the system, arriving at the place required at the precise moment in time when they will be used, before eventually returning to the host. This pumping action is the origin of the word systolic, derived from the word systole or 'contraction of the heart' borrowed from physiology. At a high level any system which preserves this attribute may be termed systolic, our analogy however is closer than that. First of all, systolic systems retain the concept that a number of independent systolic components (or arrays) can be connected to achieve more complex tasks. Individual components can be further broken down into networks (arrays) of simple self-contained circuits (cells), which are replicated to construct the complete component. The cell collections work in parallel to fulfil the special function of the component, if controls are issued incorrectly or data arrives at the wrong speed, erroneous values are produced making the whole system faulty. On the lower cell level individual cells may develop faults with time, and in a normal computing system the machine would grind to a halt. However, an interesting feature of modern systolic arrays is the inclusion of fault tolerance, which like the body introduces redundancy and checking procedures so that system performance degrades gracefully as faults develop. Also, like the body systolic systems limit the major portion of memory to the host machine (brain), with cells confined to simple functional arrangements. In the case of cells in the lung the function is a chemical reaction, replacing carbon dioxide (CO_2) with oxygen (O_2) in the blood as it flows through the cells. Similarly, in a systolic array data flows through the cells which modify it according to simple computational rules akin to the reaction.

Now, the immediate task is to trace the origins of systolic arrays so that the constraints imposed by the framework may be better appreciated, and indirectly our extensions. The correlation above between body and machine although useful should not be taken too literally, because systolic arrays abstract away the overwhelming details of physiology. For instance, true systolic arrays exhibit severe restrictions on the regularity and dimension of the cell network, betraying the roots of systolic design but which are not apparent in the above analogy.

CELLULAR AUTOMATA: From a theoretical viewpoint, systolic arrays can be traced back to cellular automata of Von-Neumann, the Mealy machine (G.F. Mealy, 1955) and Moore Machine (E.F. Moore, 1956, from his Gedanken experiments on sequential machines in Automata Studies). During the development of the systolic framework, work has progressed independently on automata studies by Katona [83] and more recently Umeo [85a]. Automata theory is basically a mathematical theory about machines and what they can accomplish at a low level of computation; it has mainly been applied to the design of electrical circuits with digital hardware, the logic of nervous systems in man and animals, and the underlying logic of protein synthesis in cells. Hence, it is not surprising that our body analogy is a good one, and general reading about automata studies can be found in Hopcroft & Ullman [79], Minsky [67].

Automata theory is tremendously important for the comprehension of systolic arrays because it lends a ready-made theory about what such machines might achieve. Automatons themselves can be represented by labelled directed graphs, with machine states represented as nodes, and arcs defining state transitions. Consequently a simple cell function

(with little or no memory) can be represented by a simple graph. Responses to inputs (i.e. outputs) can also be encoded on arcs, and from here it is a small step to connect inputs and outputs of a number of (possibly identical) machines, and operate them in parallel to create a systolic array (see Leiserson [81]). The formal specification of such parallel networks involves the breakdown of the array to its constituent machines, which are then formally defined, and linked by connecting equations. For even small arrays formal specification and hence verification is both tedious, messy, and error prone. Consequently systolic arrays have their own simpler and relatively abstract graph specification, which collapses whole machines to nodes and input/output histories to sequences on arcs. See Weiser & Davis [81], Melhem & Rheinboldt [84], H.T. Kung & Lin [83]. Thus, obviating the need for detailed understanding of automata theory. Notice, that a well established theory about machines lends some formality to the discussion, expanding nodes to smaller machines presents a hierarchical structure to the design process allowing us to fix the abstract level of design, see Thompson & Tucker [85] while a theoretic description of arrays keeps the analogy with the body quite realistic as no claims on the dimensionability or regularity of arrays are made.

VERY LARGE SCALE INTEGRATION (VLSI): The second thread of systolic array realisation was the desire to construct fast, highly parallel computing structures at low cost, H.T. Kung first realised that rapidly developing chip industry and automaton theory together could achieve this. Until the advent of VLSI, the development of parallel computers with large numbers of processors had been limited by the prohibitively high costs of manufacture. Existing machines had been improved by

tinkering with the traditional Von-Neumann architecture, for instance cycle stealing, direct memory access (DMA), and pipelining of fetch and execute operations. Parallel machines were left as mainly special, one off productions, primarily for research interests.

The development of new manufacturing techniques for fabrication of small, dense, and inexpensive, semi-conductor chips created a revolution in the computer industry. With the use of VLSI in circuits, size and cost of processing elements and memory was reduced, and it became feasible to combine the principles of automata theory with the pipeline ideas of traditional architectures. The combination was especially attractive because device manufacture cost remained constant relative to circuit complexity, with most time and money invested in design and testing.

Now, if we endeavour to sketch a complex automata arrangement one is immediately confined to the two dimensional (2-D) plane defined by sheets of paper. In fact VLSI is achieved in a similar manner by a combination of circuit design with high resolution photographic techniques (Mead & Conway [79]), where it is convenient to place wires on rectangular grids, and limit the number of parallel layers of semi-conductor material containing wires and circuit elements. Hence, the problem of collapsing a three dimensional (3-D) graph structure onto a 2-D plane or chip, is simplified if the graph is as close to 2-D as possible. (A 2-D graph is termed planar if it can be drawn in the plane with no arcs intersecting at places other than nodes). Furthermore, an 'almost' planar graph based circuit is easier to design if it is modular - i.e. composed of many replicatable components (like cells), and reduces overall production time as only a single or few cells must be designed. VLSI presents

additional problems, as the size of wires and transistors approach the limits of photographic resolution, for it becomes impossible to achieve further miniaturization and actual circuit area becomes a key issue. Add the fact that chip area is limited in order to maintain high chip yield, and the number of connections to the outside world (pins) is limited by the finite size of the chip perimeter, and a highly restrictive set of constraints is imposed on the theoretical systolic array model. These restrictions form the basis of the systolic paradigm - and cull the large number of algorithms available to a select few for implementation. See Savage [81], Ullman [84] and Kung [82] for VLSI models and area considerations.

TIMELINESS: So far we have combined a theoretical idea of parallel computation on regular connection geometries, with the physical constraints necessary for cost effective manufacture. A final third thread remains, that of applicability. It is rare in a competitive industry for a methodology or product to survive unless there is sufficient demand for it. The emergence and consequent success of systolic arrays is not due only to H.T. Kung's foresight but also the timing. At the same time Kung revealed the systolic concept, the idea of using VLSI for signal processing was the major focus of attention in governmental, industrial, and university research establishments as a means of bridging the gap between theory, algorithms and implementation. The constantly increasing demands for high performance real-time signal processing illustrated the need for a vast computation capability in both volume and speed. Clearly, fast, low cost, high density VLSI devices promised practical cost effective, high speed parallel processing for large volumes of data with ultra-high throughput rates. Furthermore, the algorithms used

relied heavily on the solution of linear equations, and as Leiserson [81] demonstrates such computations are well suited to systolic arrays. The demand ensured the survival of the systolic concept at least for the present and near future.

1.2 APPLICATIONS OF THE SYSTOLIC PRINCIPLE

Virtually a decade has passed since the first systolic principles were introduced and in hindsight a number of important events have occurred to shape the field.

After the initial concept, what can only be described as a frenzy of research activity followed, in which the principle was applied to any algorithm or area with similar properties. The commotion propagated a wave through computer science, and although not all designs or areas considered were successful, many contributed to the rules of systolic design. For instance, it was soon found that systolic arrays work best for computationally intensive tasks with a recurrence type formulation (e.g. matrix multiplication), such problems are called compute-bound, and are measured by the amount of computation versus host communication. In contrast problems with a high ratio of communication to computation are termed Input-Output (I/O) bound (e.g. matrix addition) and the systolic principle is difficult to apply efficiently in these areas. Attempts at implementing a wide variety of algorithms (see Table 1.2) have identified a small number of 'standard' networks, the most famous being the hexagonal array of H.T. Kung & Leiserson. Broadly speaking the development of systolic principleshas remained polarized around the three threads of its inception, Theory, Implementation and Applications.

SIGNAL PROCESSING	Signal processor for recursive filtering, Implementation of Kalman filters, Discrete Fourier Transform (DFT), Convolution (multi-dimensional), Linear algebra machines in digital processing.
NUMERICAL PROBLEMS	Finite element analysis, Singular value decomposition, Linear time solution of Toeplitz systems, Orthogonal equivalence transformations, Least-squares (adaptive beam forming), Eigenvalues and generalized inverses, (symmetric matrices), Iterative algorithms.
SHAPES & PATTERNS	Pattern matching, Feature extraction and pattern classification, Stereo matching, Algorithms for recti-linear polygons.
WORDS & RELATIONS	Largest common subsequence problem, Dictionary machines, Relational Database operations, Connected word recognition.
AUTOMATA	Tree acceptors, Trellis automata, Binary tree automata, Design rule checker.
GENERAL	Shortest path problem, Algebraic path problem (including matrix inverse), Fundamental sorting problems, Linear-time Greatest Common Divisor (GCD) computation, Priority queues.

TABLE 1.2: Selection of systolic applications.

On the theoretical side relationships between other forms of parallel computation and architectures have been examined. For Single Instruction Multiple Data (SIMD) and Multiple Instruction Multiple Data (MIMD) machines, Kunde, Lang, Schimmler, Schmeck and Schroder [85], S.Y. Kung [84], Umeo [85], Saxe & Leiserson [83] have considered the mapping (or systolization) of existing algorithms into systolic arrays. Producing the concept that systolic systems are special cases of Reduced Instruction Set Computers (RISCs), and that the range of applications solvable on a particular architecture, in parallel, is a trade-off between general purpose data-flow multiprocessors and dedicated architectures (like systolic arrays). Graph theoretic models of systolic arrays have been developed as aids for formal verification but remain unwieldly and fraught with difficulties. Problems arise from the lack of ability to specify cells with complex internal arrangements, and the derivation of systems of mutually recursive equations - which are unsolvable, even for intuitive and obviously correct arrays. See Melhem & Rheinboldt [84]. While Turkedjiev [86] has examined methods for enumerating all the possible systolic networks for a given problem using the idea of cluster graphs derived from the necessary data flow of the problem, again even simple problems require detailed analysis, and so far only 1-D convolution and hexagonal matrix product have been examined thoroughly. This illustrates the fact that the inherently special purpose nature of systolic arrays, so far can only be captured in special theories for certain applications. What is lacking is a general intuitive feel implicit in the theory for what a systolic algorithm is - but systolic algorithms today are confined by the constraints of VLSI and so we must be certain that we are studying systolic computation not VLSI computation.

The ultimate aim of systolic theory is the automatic synthesis of systolic arrays. That is, given a problem encoded in some high-level (parallel) language, can we automatically choose the best (i.e. area efficient, fastest) array in a suitable form to produce a chip specification which can then be manufactured. This is a formidable and long term task and is related to the concept of a silicon compiler which given a high level description of a circuit produces the 'best' chip. In this context, a systolic array synthesizer forms the front end to the silicon compiler. However, until a strict and automatic theory for verifying systolic arrays is developed it is often easier to adopt a simple 'dry run' technique or some form of simple simulation for testing.

Allied to the theoretical side is applications where the search for new arrays is conducted informally, at present. This field attracts experts from different specialist fields eager to try systolic design, but only a limited number appreciate the underlying necessity for the constraints of the systolic paradigm. Consequently there is a widening gap between new proposed systolic algorithms and the means to implement them. Designers in an attempt to apply systolic principles adopt large numbers of cells, of increased complexity and allow complex data and control movements. This neglect, or lack of concern for implementation issues is worrying because if it continues could make systolic array design an abstract principle largely of academic interest, divorced from any means of implementation.

The implementation side has progressed slower than theory or applications, but perhaps in terms of real achievements and the establishment of a hard core of practical knowledge, is the leader.

Very few of even the earliest systolic algorithms have been implemented as chips. The first attempt at implementation was the pattern matcher of Foster & H.T. Kung [80], followed by the systolic 2-D convolution chip, H.T. Kung & Song [82], the pipelined Lattice Processor (PLP) S.Y. Kung & Hu [83] for Toeplitz systems and the programmable systolic chip (PSC), Fisher, H.T. Kung, Monier, Walker & Dohi [83]. With commercial companies taking up the challenge only recently with NCR'S Geometric Arithmetic Parallel Processor (GAPP), a 6×12 arrangement of single bit processing cells (each with ALU), and Marconi's SOS systolic array (a CMOS radiation hard bit-slice correlator), with applications in radar/sonar systems, beam forming, FIR filtering and medical electronics. Problems with implementations arise for a number of reasons, such as heat dispersion when the chip is densely packed and mistiming problems due to propagation delays down wires particularly as feature size diminishes. More complex designs may benefit from a two level approach for metal lines as used in the INMOS transputer (INMOS [85]). Systolic arrays however reduce the effectiveness of cheap implementation as they can only be used on a narrow set of problems, and design cost cannot be spread over large numbers of devices. While effective parallelism often is only achieved by expanding the host array interface, demanding a high number of wires (pins). Both cell area and pin count can be modified by serializing some of the computation i.e. essentially chopping it up into identical steps and performing them serially on small hardware, see Fisher [84]. Bit-serial or byte-serial implementations have potential because hardware is easier to design, allows flexible word lengths and smaller cell cycle time (due to increased clock speed), but has the disadvantages of additional area

for latches and reduced opportunities for optimization and trade-offs at the cell level. And, unless pipelining is used in conjunction with serialization throughput of data is often reduced. For examples of bit serial signal processing applications see McCanny & McWhirter ([82], [83]).

Because of the narrow range of problems for which systolic arrays can be implemented successfully, and the growing gap between abstract arrays and implementation capability, emphasis has been placed on programmability. In generic arrays a number of related problems can be solved by a more general cell type, and usually the individual arrays have a common structure. The leading development in programmable systolic arrays is the WARP processor at Carnegie-Mellon University (CMU) Pittsburgh (H.T. Kung [84]) and is based on the PSC, plans are to incorporate the array in a general purpose systolic array computer. But, Kung himself has noted, producing generic arrays allows memory to creep into basic cells, which then are no-longer simple. Problems also occur in defining a suitable programming language - as systolic arrays have difficulty with structures like while-loops which can run indefinitely until some test is satisfied. The design of a suitable generic cell component is crucial and still an active area of research. The INMOS Transputer (a true microprocessor) has a wider range of applications, being more general purpose in nature, and a simple MIMD array can be constructed quite easily from transputer components. The transputer itself is a language (OCCAM) based design, providing concurrency and communication as a basic feature and is an example of a RISC architecture. Hardware links contain built-in hand shaking circuits and represent the software construct of a communication channel.

This makes it possible to represent abstract array designs via the language OCCAM as networks of transputers. In general a loss of speed will be expected in an algorithm over its dedicated counterpart, due to the generic nature of the processor/cell, but allows networks derived theoretically to be implemented almost directly.

1.3 TOPICS OF DISCUSSION

The individual chapters of this thesis are bound together by a number of themes and arguments which raise important questions concerning the design of systolic arrays and their future.

Our first task at the end of Chapter 3 is to define the soft-systolic paradigm a more general set of rules and heuristics. This new paradigm replaces the old framework, and relaxes the more rigid constraints in a controlled manner which is sensitive to technological advances. To illustrate the point recall the analogy in (1.1). The organs of the body are essentially 3-D conglomerates of cells, so why not relax the 2-D constraint permitting 3-D systolic arrays? Such proposals are in line with technology research as Rosenberg [83] presents a case study of 3-D VLSI, indicating that overall wire length and design volume savings will be made over existing 2-D approaches, whereas in optical computing (Caulfield, Rhodes, Foster & Horvitz [81], and Goodman, Leonberger, S.Y. Kung, & Athale [84]) illustrate that free space and wave guided transmission of light can be used to overcome pin and long-wire restrictions with data and clock transmission at the speed of light, to implement systolic arrays.

A particularly strong theme throughout the work is the representation of systolic algorithms/arrays as OCCAM programs. The

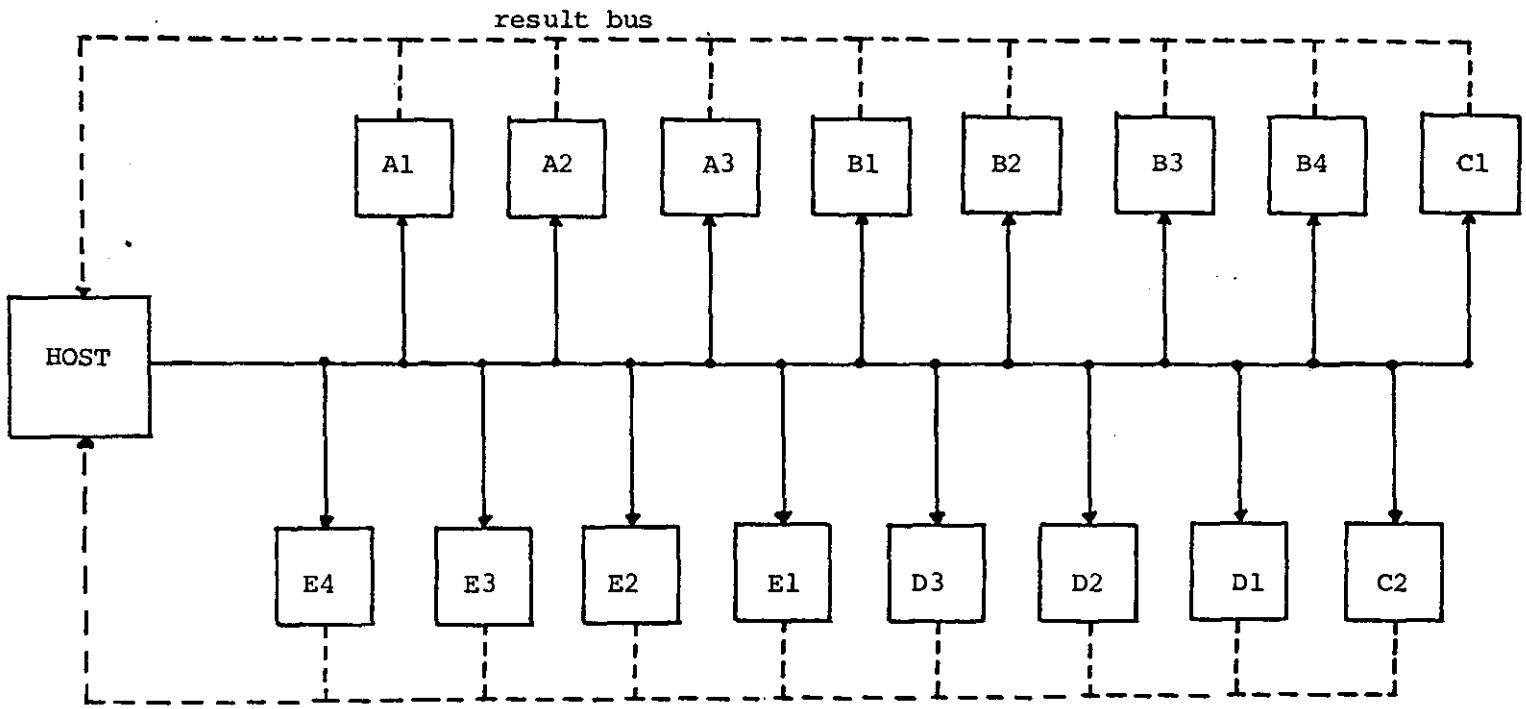
OCCAM language (see Appendices for synopsis of syntax) contains some useful features for mapping directed graph structures to parallel programs. OCCAM also doubles as a semi-formal verification tool; by semi-formal we mean that the usual error-prone hand testing of arrays is performed by program execution, hence reduced testing time facilitating quick debugging. We must be careful at this stage and observe Dikjistra's old adage - that program testing reveals only the presence not the absence of bugs. Program representation does more than correct initial design problems, it side steps the complex and problematical formal verification techniques mentioned above, while retaining a vestige of formality. To elaborate, OCCAM is based on the language CSP (Communication Sequential Processes) Hoare [78], Hehner & Hoare [83] and developed by David May at INMOS. The proof of correctness of programs is achieved by the use of invarient arguments, and these ideas can be extended via CSP to provide correctness proofs of OCCAM programs. Hence, the correctness of a systolic array can be defined implicitly, although this is not pursued in the text. Instead we adopt a semi-formal method where correct output of the program indicates array 'correctness', a selection of programs for major designs appears in the Appendices. The use of programs to represent systolic algorithms is extended in Chapter 9, to define a more general purpose architecture and its extensions, where, the role of the rhythmically recurrent pumping action is retained only at a low architectural level, facilitating program execution.

Emphasis is also placed on defining new efficient algorithms using relaxed constraints, as well as improving or redesigning arrays which suffer certain difficulties under the old paradigm. The general

theme is the improvement of systolic array efficiency and computation time. In this context, efficiency is taken to be the proportion of cycles a cell is active during the computation, or the number of cells used by different systolic designs. In order to achieve a measure of improvement, where possible new arrays have been compared with old ones. As the thesis develops it becomes apparent that efficiency improvement is related to algorithmic and geometric implementations of arrays together with various assumptions about the type of hardware available. In particular, new arrays are developed from cells based on simple computational rules and molecules implicitly defining the geometry of a problem, rather than the recurrence relations on which algorithmic array forms have been based to date.

The third strand running through the thesis, is the special purpose nature of systolic arrays. As arrays are inherently application dependent the philosophy for design must be decided at the outset, in the main we attempt to introduce generic arrays where possible. However, general purpose design is problematical because applications dictate structure. As a basic model to unify the designs, which are mainly numerical, it is appropriate to regard the thesis as an attempt to produce a systolic hardware library of components. Each component can be considered akin to a routine called from LINPACK or EISPACK, and we might call our component machine SYSPACK for reference. The structure of the hypothetical machine is shown in Fig. 1.1 and also defines the structure of the thesis.

The use of components raises the issue of granularity in systolic design. Granularity of an algorithm refers to the maximal amount of computation a typical module can perform before having to communicate



KEY:
- A1: Block hexs and double pipes
- A2: Rank annihilation
- A3: Banded and Toeplitz solver (BATS)
- B1: Matrix power generator
- B2: Systolic preconditioning
- B3: Incomplete factorisation
- B4: Incomplete elimination
- C1: Direct } QI
- C2: Iterative }
- D1: Table generators
- D2: Simplex
- D3: Assignment
- E1: Systolic marching
- E2: 1-D } GE methods
- E3: 2-D }
- E4: 1-D hyperbolic

FIGURE 1.1: Structure of the Syspack Machine

with other modules. Furthermore, the choice of granularity is often critical for the performance of an algorithm. The module size in Fig. 1.1 can be classed as medium sized, as each component solves a relatively complex task, using a collection of pipelined systolic arrays. A finer grain setup is to use a small set of common arrays which reduces the total number of arrays, but loses pipelining features due to increased communication requirements. This fine grain will be referred to as the 'bag-of' approach derived from the idea of having a bag of useful components from which you extract the most appropriate for each stage of a calculation. A larger grain size is exhibited in Chapter 9 where systolic algorithms are partitioned into blocks of program instructions on a general architecture.

1.4 OVERVIEW OF THE THESIS

The structure of the sys-pack machine gives a global view of the work contained in this thesis.

CHAPTER 2: here sections of basic mathematical definitions and concepts necessary for the description of algorithms are given.

CHAPTER 3: gives a broad foundation of the basic techniques and definitions for the design of systolic algorithms/arrays. Included is a treatment of the representation of arrays by a computational graph model together with a method for mapping graphs to OCCAM programs. Basic quantities such as area, cycle time, efficiency and the structure of common networks are given. Various trade-offs are examined in relation to area, efficiency and time, and the concept of two-level pipelining for improving the throughput introduced.

CHAPTER 4: presents the concept of double pipes for improving array efficiency by extending layout to a small number of layers in

which circuits remain planar on individual slices. The technique is modified to introduce block partitioning for traditional hexagonal patterns and basic theorems about the best block size and efficiency constructed. Finally, a stable and efficient systolic pipeline is developed for the solution of circulant matrices, using first the method of rank annihilation and second a novel factorisation scheme.

CHAPTER 5: considers the Quadrant Interlocking (QI) methods for factorisation, elimination and iterative methods and systolic arrays, based on the previous chapters, derived. It is shown that block partitioning affects efficiency only on systolic arrays with certain input formats. Applications of the arrays for periodic matrices are shown to be superior than those of existing designs.

CHAPTER 6: examines the theory of preconditioning, incomplete factorisation, and matrix triangularisation to produce area efficient designs. In preconditioning a new hex array pipeline, for repeatedly squaring a matrix is described and used as a preprocessing array to reduce the number of iterations, hence hardware required by systolic iterative methods. Whereas, the incomplete theory is used to derive new numerical algorithms based on optimal systolic arrays, which can take advantage of sparsity within the band of a matrix and for which standard systolic arrays produce fill-in and high area usage.

CHAPTER 7: extends the use of systolic arrays to the parallel construction of tables. The idea of a table template is introduced and used to derive arrays for extrapolation, solution of ordinary differential equations (ODE's), and an area efficient ring design produced. The main concepts are used to develop a unified generic array for differencing operations like rational function approximation

and Wynn's ε-algorithm (for converting slowly converging sequences to rapidly converging ones). The Quotient Difference algorithm for root finding is implemented, before more complex table algorithms like the simplex, revised simplex and assignment problems are discussed.

CHAPTER 8: extends the ideas of templating to computational molecules and representation of a problem in its geometric form. Systolic marching techniques based on asymmetric approximations of Saul'ev [64] are introduced for 1-D and 2-D parabolic partial differential equations and compared with iterative forms derived from purely algorithmic implementation. The group explicit methods are then used to develop fast arrays and the best is used to examine bit serial implementation with fixed point arithmetic. Area efficient versions of the arrays are introduced based on hopscotch techniques before the ideas are extended to 1-D hyperbolic equations.

CHAPTER 9: limitations of the syspack structure are discussed and as already indicated, a more general architecture for simulating systolic algorithms and its operation and extensions are described and illustrated with examples. In particular an area efficient design for the implementation is discussed based on collapsing 2-D arrays to 1-D alternatives, and program transformations to the new arrays with respect to SIMD and MIMD algorithms considered.

CHAPTER 10: concludes the thesis and gives an overview of techniques developed during the work and outlines areas of further study. In particular the Systolic Control Ring Instruction Processor (SCRIP) is proposed as a conglomerate of designs to replace the syspack machine and provide a general purpose systolic computer.

CHAPTER 2

BASIC MATHEMATICS

"A good Notation has a subtlety and suggestiveness which at times make it seem almost like a live teacher".

BERTRAND RUSSELL.

In this chapter basic definitions and theory about Linear Algebra, Linear Systems and Partial Differential Equations is given. The material presented is necessary for the ready comprehension of the Systolic Algorithms in later chapters.

Linear Algebra deals with the specification and solution of linear systems (of equations) which can be derived from a variety of problems in Engineering, Mathematics, Business and Economics. In fact differential equation problems themselves can be written in terms of Linear Systems. Furthermore the recurrence relations inherent in matrix formulations and subsequent computations make them suitable for Systolic Array applications. Consequently, matrix notation can be used as a cipher to translate numerical algorithms to systolic arrays.

The interested reader is referred to more advanced texts such as Varga[62] and Evans[83], while introductory material is given in Burden, Faires & Reynolds [81], Smith [85], Lipschutz [74] and Wu & Coppins [81].

First of all the chapter defines vectors and matrices together with relevant properties and relations. This base is then used to discuss direct and iterative methods for solving linear systems. Partial differential equations are defined next, and by the use of the finite difference technique the corresponding linear systems are derived. Finally, some brief definitions of convex sets are provided for discussion of topics in Chapter 7.

2.1 VECTORS

In mathematical terms a vector is an ordered n-tuple which can be represented as either a row or column of elements, viz,

$$a = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} \quad \text{or } b = [a_1, a_2, \ldots, a_n] \tag{2.1.1}$$

whether row or column forms are used depends largely on the context in which the vector appears. For instance, it is quite acceptable to write $a=b^t$ (or $b=a^t$), where the superscript 't' denotes transposition, and indicates that a row vector can be converted to a column vector (and vice versa). The values a_i, i=1(1)n are termed components and throughout the thesis we shall denote vectors by Roman lowercase letters, and their components with the same letter but subscripted to indicate position unless the meaning is clear. The vectors in (2.1.1) are said to be from n-space and we further define $\mathbb{R}$ to be the set of real numbers, with $\mathbb{R}^n$ the n-space of vectors with components from $\mathbb{R}$. Consequently (2.1.1) would be defined as $a,b \in \mathbb{R}^n$ and $a_i \in \mathbb{R}$ for i=1(1)n. Values like $\alpha \in \mathbb{R}$ which are not vector components in a problem specification are termed scalars and will be denoted by lowercase greek letters where confusion would otherwise occur. Vectors and the operations on them form vector spaces which are Algebraic structures involving fields (or commutative rings, see Section 2.6). The main operations are defined below and are used with the proviso that the vectors in an operation are all from the same space.

EQUALITY: if $a,b \in \mathbb{R}^n$ then $a=b$ if $a_i=b_i$, i=1(1)n (2.1.2)

ADDITION: for $a,b,c \in \mathbb{R}^n$,

$$c = (a_1+b_1, a_2+b_2, \ldots, a_n+b_n)^t \tag{2.1.3}$$

SCALAR MULTIPLICATION: for $a \in \mathbb{R}^n$ and $\alpha \in \mathbb{R}$

$$\alpha a = \alpha(a_1, \ldots, a_n)^t = (\alpha a_1, \alpha a_2, \ldots, \alpha a_n)^t \tag{2.1.4}$$

SUBTRACTION: for $a,b \in \mathbb{R}^n$ and $\alpha \in \mathbb{R}$

set $\alpha=-1$ form αb then add to a, $a+\alpha b=a+(-1)b$ (2.1.5)

INNER PRODUCT: for $a,b,c \in \mathbb{R}^n$ and $\alpha \in \mathbb{R}$

$$\alpha = a.b = a_1b_1+a_2b_2+ \ldots +a_nb_n = \sum_{j=1}^{n} a_jb_j$$

and also observes the following rules,

$$\left.\begin{array}{ll} \text{(i)} & (a+b).c = a.c+b.c = c.(a+b) \\ \text{(ii)} & (\alpha a).b = \alpha(a.b) \\ \text{(iii)} & a.b = b.a \\ \text{(iv)} & a.a \geq 0 \text{ and } a.a=0 \text{ iff } a=0 \\ \text{(v)} & (a+b).(c+d) = a.c+a.d+b.c+b.d \end{array}\right\} \quad (2.1.6)$$

BASIC ALGEBRA: for $a,b,c \in \mathbb{R}^n$ and scalars $\alpha,\beta \in \mathbb{R}$

$$\left.\begin{array}{llll} \text{(i)} & (a+b)+c = c+(a+b) & \text{(v)} & \alpha(a+b) = \alpha a+\alpha b \\ \text{(ii)} & a+0 = a & \text{(vi)} & (\alpha+\beta)a = \alpha a+\beta a \\ \text{(iii)} & a+(-a) = 0 & \text{(vii)} & (\alpha\beta)a = \alpha(\beta a) \\ \text{(iv)} & a+b = b+a & \text{(viii)} & 1.a = a \end{array}\right\} \quad (2.1.7)$$

using these basic formulae a number of basic definitions and concepts can be constructed, for the simpler formulas and choosing $\mathbb{R}^k$, $k \leq 3$ gives a geometric interpretation.

Definition 2.1.1: Vectors $a,b \in \mathbb{R}^n$ are said to be *orthogonal* (perpendicular) if $a.b=0$.

Definition 2.1.2: A *Vector Norm* is a mapping of a vector in $\mathbb{R}^n$ to an element of $\mathbb{R}$ and is denoted $||.||$. Intuitively, the norm measures the size (length) of a vector and satisfies the properties below:

$$\left.\begin{array}{ll} \text{(i)} & ||a|| \geq 0 \text{ for all } a \in \mathbb{R}^n \\ \text{(ii)} & ||a||=0 \text{ iff } a=(0,\ldots,0)^t = 0 \\ \text{(iii)} & ||\alpha a||=|\alpha|\ ||a|| \text{ for all } \alpha \in \mathbb{R},\ a \in \mathbb{R}^n \\ \text{(iv)} & ||a+b|| \leq ||a||+||b|| \text{ for all } a,b \in \mathbb{R}^n \end{array}\right\} \quad (2.1.8)$$

Any computation rule connecting components of a and satisfying (2.1.8) is a norm, different norms are denoted by a subscript e.g. $L_p=||a||_p$ for the p-norm and from a practical viewpoint the most useful norms are given by

Definition 2.1.3: The L_1, L_∞, and L_2 norms for all $a \in \mathbb{R}^n$ are defined respectively as,

$$L_1: ||a||_1 = |a_1|+|a_2|+ \ldots +|a_n| = \sum_{i=1}^{n} |a_i| \tag{2.1.9}$$

$$L_\infty: ||a||_\infty = \max_i |a_i| \tag{2.1.10}$$

$$L_2: ||a||_2 = (|a_1|^2+|a_2|^2 +\ldots+|a_n|^2)^{\frac{1}{2}} = [\sum_{i=1}^{n} |a_i|^2]^{\frac{1}{2}} \tag{2.1.11}$$

The L_2 norm gives the geometric interpretation of vector length (for $\mathbb{R}^k$, $k\leq 3$). Consequently the distance between two vectors a,b in general is given by the length of the vector between a and b or $||a-b||_2$ and motivates the following result.

Definition 2.1.4: A sequence $\{a^{(k)}\}_{k=1}^{\infty}$ of vectors from $\mathbb{R}^n$ is said to be *convergent* (or converges) to a vector a in the same space under norm $||.||$ if for $\varepsilon>0$ and some integer N_ε,

$$||a^{(k)}-a|| < \varepsilon \text{ for all } k>N_\varepsilon \tag{2.1.12}$$

The relation (2.1.12) indicates an error bound on the approximation, and due to the finite arithmetic in computers which introduce rounding errors, is often used as an algorithm termination criterion. In iterative algorithms the value ε is termed the tolerance representing the accuracy of solution, when (2.1.12) is satisfied the distance between an approximate and exact vector are considered close enough and the algorithm terminates having satisfied the tolerance. All norms on space $\mathbb{R}^n$ are equivalent with respect to convergence, meaning, that if the sequence converges for one norm, it does so for all the others, but possibly at different rates.

Extending the ideas of the inner product, results in the following important relationships.

Definition 2.1.5: A linear combination of vectors $a_1,\ldots,a_m \in \mathbb{R}^n$ and scalars $\alpha_1,\ldots,\alpha_m \in \mathbb{R}$ not all zero is a set of vectors which when combined by vector addition and scalar multiplication form a new vector $a_{m+1} \in \mathbb{R}^n$

$$a_{m+1} = \sum_{i=1}^{m} \alpha_i a_i \,. \tag{2.1.13}$$

If no a_i, i=1(1)m can be formed as a linear combination of the others the set is termed *Linearly Independent* otherwise *Linearly Dependent.*

In particular, linear independence implies orthogonality from (2.1.6) as,

$$\sum_{i=1}^{m} \alpha_i a_i = 0 \text{ otherwise } -\alpha_j a_j = \sum_{\substack{i=1 \\ i\neq j}}^{m} \alpha_i a_i \tag{2.1.14}$$

Definition 2.1.6: A *vector space* (e.g. $\mathbb{R}^1$, $\mathbb{R}^2$, $\mathbb{R}^3$, etc.) is any set of vectors closed under addition and scalar multiplication. In this context closed means that any operation performed on the vectors results in a vector from the same space.

A *subspace* of a vector space is simply a set of vectors contained in the vector space, and which are also closed.

Definitions (2.1.5) and (2.1.6) combine to form the concept of a basis.

Definition 2.1.7: A set of vectors $a_1,\ldots,a_m \in \mathbb{R}^n$ form a *spanning set* for $\mathbb{R}^n$ if every vector in $\mathbb{R}^n$ can be written as a linear combination of the a_i, i=1(1)m. If the a_i are also linearly independent the spanning set is termed a *Basis.*

It can further be shown (see Lipschutz[74]) that a Basis contains only n vectors to span the space $\mathbb{R}^n$, and that this set is a minimal

spanning set. That is, the smallest possible set of vectors which still spans the space. As a simple example, a Basis for $\mathbb{R}^3$ (3-D) is the set $s=\{(1,0,0),(0,1,0),(0,0,1)\}$ and a linear combination

$$a = \alpha_1(1,0,0)+\alpha_2(0,1,0)+\alpha_3(0,0,1) \ , \quad \alpha_i \in \mathbb{R}$$

generates any vector in $\mathbb{R}^3$ (hence s is a spanning set), now remove a vector to give $s_1=\{(1,0,0),(0,1,0)\}$ and

$$b = \alpha_1(1,0,0) + \alpha_2(0,1,0) \ ,$$

which generates vectors restricted to a 2-D space ($\mathbb{R}^2$), indicating (but not proving) that s has the minimal number of vectors for a spanning set of $\mathbb{R}^3$.

Incidentally s_1 is a Basis for $\mathbb{R}^2$, and we note that any subset of a linearly independent set must itself be a linearly independent set.

2.2 MATRICES

Matrices are important to Numerical Analysis because they provide a concise method for specifying and manipulating large numbers of linear equations. The collection of equations and their unknowns is called a linear system if each one can be expressed in the form,

$$a_1x_1 + a_2x_2 + \ldots + a_mx_m = b \ , \tag{2.2.1}$$

with x_i, a_i, $b \in \mathbb{R}$, i=1(1)m. The x_i, i=1(1)m are the unknowns, a_i the coefficients and b the constant or right hand side (RHS) term. Hence, for an n equation system we write,

$$\left.\begin{aligned} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1m}x_m &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2m}x_m &= b_2 \\ \vdots \qquad\qquad\qquad\quad & \ \ \vdots \quad \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nm}x_m &= b_m \end{aligned}\right\} \tag{2.2.2}$$

Definition 2.2.1: An n by m (or n*m) *matrix* is a rectangular array of

elements (or scalars) from $\mathbb{R}$, with n rows and m columns in which not only the elements value is important but also its position. When m=n, the matrix is *square*, and said to be order n (or m) when m=1 reduces to a column vector or when n=1 a row vector and with m=n=1 produces a single scalar value. Throughout the text we shall denote matrices by upper case Roman letters, pictorially or as a row vector of column vectors as illustrated below,

$$A = (a_{ij}) = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{bmatrix} = [c_1, c_2, \ldots, c_m] \qquad (2.2.3)$$

where $c_i = [a_{1i}, a_{2i}, \ldots, a_{ni}]^t$, $a_{ij} \in \mathbb{R}$ for i=1(1)n, j=1(1)m and a_{ij} locates the element at the intersection of the ith row and jth column. Thus, a linear system can be formally specified by matrices and vectors as,

Definition 2.2.2: A *Linear System* of equations can be represented by,

$$Ax = b \,, \qquad (2.2.4)$$

with A an n×m coefficient matrix, and x,b vectors. The system is *homogeneous* if the components of b, b_i=0, i=1(1)n and always has a *trivial solution* with x_i=0, i=1(1)m (the components of x), any solution with some $x_i \neq 0$ is termed a *nontrivial solution*. A non-homogeneous system has a *particular solution* if $u \in \mathbb{R}^n$ satisfies (2.2.4) when substituted for x, and the set of all vectors satisfying (2.2.4) gives the *general solution*.

In fact we shall restrict our attention to linear systems with mainly square coefficient matrices, and which often arise from physical problems. Fortunately such systems if solvable produce only a single or

unique solution obviating the need to deal with general solution sets.

Similar to vectors, matrices observe rules from an algebraic structure, this time a non-commutative ring. Many matrices have elements whose positioning form special structures and together with the value of the entries produces certain properties. Such quirks are essential for deriving relationships between matrices and identifying classes of matrices which in some sense are easier to solve than others. Special structures and properties are detailed below, but the solution of large and intricately related equations often needs some manipulation before a form corresponding to (2.2.4) is produced, consequently basic operations on matrices are required.

EQUALITY:Two n×m matrices A and B are equal iff $a_{ij}=b_{ij}$ for i=1(1)n, j=1(1)m and is denoted A=B.

ADDITION: For two n×m matrices $A=(a_{ij})$ and $B=(b_{ij})$ is written

$$\left.\begin{aligned} &C = A+B = (a_{ij}+b_{ij}) = (c_{ij}) \text{ for } i=1(1)n, j=1(1)m \\ &\text{and obeys the rules} \\ &A+B = B+A \text{ and } (A+B)+C = A+(B+C) \end{aligned}\right\} \qquad (2.2.5)$$

SCALAR MULTIPLICATION: Let A,B be n×m matrices and $\alpha,\beta \in \mathbb{R}$ then

$$\left.\begin{aligned} &\alpha A = \alpha(a_{ij}) = (\alpha a_{ij}) \quad i=1(1)n,\ j=1(1)m \\ &\text{and} \qquad \alpha(A+B) = \alpha A+\alpha B, \quad (\alpha+\beta)A = \alpha A+\beta A \end{aligned}\right\} \qquad (2.2.6)$$

MATRIX MULTIPLICATION: is possible for two matrices A and B only if A has the same number of columns as rows of B. Let A be an m×p and B a p×n matrix the product,

$$C = (c_{ij}) = AB = \sum_{k=1}^{p} a_{ik}b_{kj} \quad , i=1(1)m,\ j=1(1)n \qquad (2.2.7)$$

is an m×n matrix.

BASIC ALGEBRA: for compatible matrices A,B,C,

$$\left.\begin{array}{ll} \text{(i)} & AB \neq BA \\ \text{(ii)} & A(BC) = (AB)C \\ \text{(iii)} & (A+B)C = AC+BC,\ C(A+B) = CA+CB \\ \text{(iv)} & \alpha(AB) = (\alpha A)B = A(\alpha B) \end{array}\right\} \qquad (2.2.8)$$

Notice that for A and B n×m matrices setting n=1 or m=1 reduces scalar multiplication to the same definition as for vectors. Selecting B to be m×1 defines matrix vector multiplication, and with A also 1×m produces the inner product operation.

As mentioned earlier certain matrix structures are useful, below are some which recur throughout Linear Algebra problems.

S0: NULL (or zero) matrix denoted 0, with $a_{ij}=0$ for all i,j

S1: IDENTITY matrix denoted I, a square n×n matrix with

$$a_{ij} = \begin{cases} 0 & i \neq j \\ 1 & i=j \end{cases} \qquad i,j=1(1)n \qquad (2.2.9)$$

S2: DIAGONAL matrix (usually denoted D), n×n square matrix

$$a_{ij} = \begin{cases} 0 & i \neq j \\ \text{non-zero} & i=j \end{cases} \qquad i,j=1(1)n \qquad (2.2.10)$$

when a diagonal matrix has $a_{ii}=\alpha$, $\alpha \in \mathbb{R}$ it is sometimes called a *scalar* matrix. Non-zeros are situated on the main diagonal.

S3: UPPER TRIANGULAR MATRIX (U) is an n×n matrix in which all the elements below the main diagonal are zero.

S4: LOWER TRIANGULAR MATRIX (L) same as structure S3 but with zeros above the main diagonal, non-zeros below.

When the main diagonal is zero, S3 and S4 are termed *strictly* upper (lower) triangular respectively.

S5: a BAND MATRIX is an n×n matrix together with integers p and q $1<p,q<n$ such that $a_{ij}=0$ for $i+q \leq j$ or $j+p \leq i$ the

$$\textit{bandwidth } w=p+q-1. \qquad (2.2.11)$$

Common matrices arising from this structure are the *Tridiagonal* matrix (p=q=2) and *Quindiagonal* matrix (p=q=3).

A *subdiagonal* is defined as a line of elements parallel to the main diagonal with a_{ij} for j<i, in the above structure there are q-1 subdiagonals. For a_{ij} with i<j we define *Superdiagonals* (in a band matrix there are p-1). Notice that a band matrix restricts only the positioning of non-zero elements. Individual sub(super)diagonals can also contain zeros and adjacent groups of such diagonals allows the construction of *striped matrices*.

S6: a SPARSE MATRIX is produced when the elements are predominantly zero. Consequently band matrices of high order tend to be sparse e.g. the tridiagonal matrix. Usually however, no restrictions on the placement of the minority non-zeros are assumed, if a pattern exists (like a banded form) the matrix is termed *regularly sparse* otherwise *irregularly sparse*.

S7: a DENSE MATRIX in constrast to S6 contains predominantly non-zero elements and only a few zeros.

S8: a SYMMETRIC MATRIX is a square matrix of order n which is symmetric about the main diagonal, that is $a_{ji}=a_{ij}$, i,j=1(1)n. A square matrix symmetric about the opposite diagonal is termed *per-symmetric*. Alternatively, a matrix which is symmetric with $a_{ji}=-a_{ij}$, i,j=1(1)n is termed *skew symmetric (or Toeplitz)*.

S9: a CIRCULANT MATRIX A of order n has the form,

$$A = \begin{bmatrix} \alpha_0 & \alpha_1 & \cdots & \alpha_{n-1} \\ \alpha_{n-1} & \alpha_0 & \alpha_1 & \cdots & \alpha_{n-2} \\ \vdots & & & & \vdots \\ \alpha_1 & \alpha_2 & \cdots & \alpha_{n-1} & \alpha_0 \end{bmatrix}, \quad \alpha_i \in \mathbb{R}, \ i=0(1)n-1 \qquad (2.2.12)$$

related to the circulant form is a *Periodic Matrix* which is often a band matrix form of A.

S10: the TRANSPOSE of a matrix A is denoted A^t (or A^T) of a rectangular matrix and is formed by interchanging the rows and columns, that is $(a_{ji})^T=(a_{ij})$ and satisfies the following rules,

$$\left.\begin{array}{ll} \text{(i)} & (A^t)^t = A \\ \text{(ii)} & (A+B)^t = A^t+B^t \\ \text{(iii)} & (AB)^t = B^tA^t \\ \text{(iv)} & \text{IF } A^{-1} \text{ exists } (A^{-1})^t = (A^t)^{-1} \quad (A^{-1} \text{ see later}) \\ \text{(v)} & \det(A^t) = \det(A) \qquad\qquad (\det(A) \text{ also later}) \end{array}\right\} \qquad (2.2.13)$$

Notice that if $A=A^T$ the matrix is symmetric.

S11: PARTITIONED (BLOCK) forms of a matrix are constructed by dividing the elements into non-overlapping submatrices, if the submatrices are square k×k matrices the matrix is said to be in k×k block form and regularly partitioned, if submatrices are of different size we have an irregular partition.

S12: a PERMUTATION MATRIX (usually denoted P) is a square matrix which has precisely one entry in each row and column, with all other entries zero. e.g. 3×3 permutation matrix

$$P = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \qquad (2.2.14)$$

S13: a ROTATION MATRIX (R) is a matrix differing from the identity matrix (I) in at most four elements which have the form,

$$r_{ii} = r_{jj} = \cos\theta \quad \text{and} \quad r_{ji} = -r_{ij} = \sin\theta$$

for given angle θ. Sometimes the elements are adjacent where j=i+1 and the matrix can then be partitioned to produce a 2×2 submatrix

containing the elements for which the special notation below is often used,

$$\begin{bmatrix} r_{ii} & -r_{ij} \\ r_{ji} & r_{jj} \end{bmatrix} = \begin{bmatrix} c & -s \\ s & c \end{bmatrix} \tag{2.2.15}$$

Accompanying the structures SO-S13 are a number of properties but before these are given some additional concepts are required.

The idea of a vector norm (definition 2.1.2) can be extended to matrices yielding,

Definition 2.2.3: A Matrix Norm for a Real n×n matrix is a real valued function denoted $||.||$ defined for all matrices A,B and a scalar $\alpha \in \mathbb{R}$ satisfying

$$\left.\begin{array}{ll} \text{(i)} & ||A|| \geq 0 \\ \text{(ii)} & ||A||=0 \text{ iff A is a null matrix} \\ \text{(iii)} & ||\alpha A||=|\alpha|\ ||A|| \\ \text{(iv)} & ||A+B|| \leq ||A||+||B|| \\ \text{(v)} & ||AB|| \leq ||A||\ ||B|| \end{array}\right\} \tag{2.2.16}$$

The left-hand side of (2.2.4) uses matrix vector multiplication and indicates that both vector and matrix norms may appear together, in such situations it is imperative that the two norms are compatible or consistent. Compatibility means satisfying the condition,

$$||Ax|| \leq ||A||\ ||x|| \ , \quad x \neq 0 \ , \tag{2.2.17}$$

Now, if we take x from a set S (with $x \in S$ iff $||x||=1$), and further denoted $x_0 \in S$ as the vector which makes $||Ax||$ a maximum, then,

$$||A|| = ||Ax_0|| = \max_{||x||=1} ||Ax|| \tag{2.2.18}$$

a matrix norm satisfying this stronger condition is termed a *subordinate norm*. The condition is stronger than (2.2.17) because if ensures compatibility as,

$$||Ax_i|| \leq ||Ax_o|| = ||A|| = ||Ax_i|| \text{ for any } x_i \in S \quad (2.2.19)$$

Definition 2.2.4: A is a convergent matrix when,

$$\lim_{k\to\infty} (A^k)_{ij} = 0 \text{ for } i,j=1(1)n \quad 0=\text{null matrix} \quad (2.2.20)$$

As the coefficients of A are used to construct norms it can also be shown that $\lim_{k\to\infty}||A||^k=0$ when A is convergent.

Definition 2.2.5: The determinant of a square matrix A, *det(A)*, is given by,

(i) If $A=[\alpha]$ is a 1×1 matrix $\det(A)=\alpha$ $\alpha \in \mathbb{R}$

(ii) The *Minor* M_{ij} is the determinant of the (n-1)×(n-1) submatrix of A obtained by deleting the ith row and jth column

(iii) The *Cofactor* A_{ij} associated with M_{ij} is $A_{ij}=(-1)^{i+j}M_{ij}$

(iv) Thus the determinant of A for n>1 is

$$\left.\begin{aligned} \det(A) &= \sum_{j=1}^{n} a_{ij}A_{ij}, \quad i=1(1)n \\ \text{or} \\ \det(A) &= \sum_{i=1}^{n} a_{ij}A_{ij}, \quad j=1(1)n \end{aligned}\right\} \quad (2.2.21)$$

depending on whether the rows or columns are expanded.

An interesting question, (which leads to a very useful branch of Linear Algebra) is whether the matrix A in a system can be substituted by a simple scalar $\lambda \in \mathbb{R}$. Formally, can some λ for $x\neq 0$ be found for which,

$$Ax = \lambda x , \quad (2.2.22)$$

and is termed the eigen-problem. Alternatively we can use the following definition relating the eigen-problem to the matrix determinant.

Definition 2.2.6: If $P(\lambda)=\text{Det}(A-\lambda I)$ is a polynomial, with A an order n matrix and λ a scalar, the zeros (roots) of P are called *eigenvalues*, and associated with each value is an *eigenvector* $x\neq 0$ satisfying (2.2.22).

As A is of order n, $P(\lambda)$ can have at most n roots or eigenvalues λ_i, i=1(1)n and hence n eigenvectors, from Definition (2.2.2) each vector is the non-trivial solution of the homogeneous system $(A-\lambda I)x=0$. $P(\lambda)$ is often termed the *characteristic polynomial* and λ the characteristic value.

Definition 2.2.7: The spectral radius $\rho(A)$ of any n×n matrix A is defined as the maximum eigenvalue associated with A., i.e.,

$$\rho(A) = \max_{1\leq i\leq n} |\lambda_i| \tag{2.2.23}$$

Definition 2.2.8: The P-condition number of the matrix A is defined as,

$$P = b/a \ , \tag{2.2.24}$$

where $a,b \in \mathbb{R}$ satisfy $a\leq|\lambda_i|\leq b$, i=1(1)n and are the largest and smallest eigenvalues respectively.

The spectral radius is extremely useful (particularly in iterative solution of linear systems) because it allows the structure and properties of a coefficient matrix via the eigenvalues to influence the performance of the solution technique. To develop the use of the spectral radius more definitions are required including the most common matrix norms.

Definition 2.2.9: The L_1,L_2 and L_∞ matrix norms for an n×n matrix A are given by,

$$||A||_1 = \max_{1\leq j\leq n} \sum_{i=1}^{n} |a_{ij}| \quad \text{maximum column sum} \tag{2.2.25}$$

$$||A||_\infty = \max_{1\leq i\leq n} \sum_{j=1}^{n} |a_{ij}| \quad \text{maximum row sum} \tag{2.2.26}$$

$$||A||_2 = \rho(A^T A)^{\frac{1}{2}} \quad \text{Euclidean Norm} \tag{2.2.27}$$

REMARK: $||A||_2=\rho(A^H A)^{\frac{1}{2}}$ when A has complex matrix elements, where A^H is the Hermitian or complex conjugate transpose of A.

These norms can be used to place bounds on the eigenvalues of a coefficient matrix by using the following theorems.

Theorem 2.2.1: $\rho(A) \leq ||A||$ for the n×n matrix A.

Proof:

Let λ_i, i=1(1)n be the eigenvalues of A and x_i, i=1(1)n the associated eigenvectors, then,

$$Ax_i = \lambda_i x_i$$

and $$||Ax_i|| = ||\lambda_i x_i|| = |\lambda_i| \, ||x_i||$$

hence for compatible norms,

$$|\lambda_i| \, ||x_i|| = ||Ax_i|| \leq ||A|| \, ||x_i||$$

$$|\lambda_i| < ||A||$$

for an eigenvalue of A, and in particular the largest, thus by Definition 2.2.7,

$$\rho(A) \leq ||A|| \tag{2.2.28}$$

Corollary 2.2.1: If $||A||<1$ by definition (2.2.4) A is convergent and as $\rho(A)<1$ must be true $\lim_{k\to\infty} \rho(A)^k = 0$. Now $||A||<1$ is a necessary and sufficient condition for convergence but with $\rho(A)<1$ this does not follow as $||A||>1$ could be true. Consequently $\rho(A)$ is a tighter bound on the convergence rate of A.

Theorem 2.2.2: (Gerschgorin Disk theorem)

Let D_s be the sum of the absolute values of elements along the sth row excluding the element a_{ss} of an n×n matrix A. Then each eigenvalue of A lies inside or on the boundary of at least one of the n circles with centre a_{ii} and radius D_i, i=1(1)n.

Proof:

Let λ_i be an eigenvalue of A then,

$$Ax_i = \lambda x_i$$

and with $x_i=(\beta_1,\beta_2,\ldots,\beta_n)^t$ and expanding to the form in (2.2.2) row s

has the appearance,

$$a_{s1}\beta_1 + a_{s2}\beta_2 + \ldots + a_{sn}\beta_n = \lambda_i\beta_s$$

giving,

$$\lambda_i - a_{ss} = a_{s1}(\frac{\beta_1}{\beta_s}) + a_{s2}(\frac{\beta_2}{\beta_s}) + \ldots a_{s,s-1}(\frac{\beta_{s-1}}{\beta_s}) + a_{s,s+1}(\frac{\beta_{s+1}}{\beta_s}) + \ldots$$

$$+ a_{sn}(\frac{\beta_n}{\beta_s})$$

Thus,

$$|\lambda_i - a_{ss}| = |a_{s1}(\frac{\beta_1}{\beta_s}) + \ldots + 0 + \ldots + a_{sn}(\frac{\beta_n}{\beta_s})|$$

and if s has the largest row sum,

$$\left|\frac{\beta_i}{\beta_s}\right| < 1, \quad i=1(1)n \quad i \neq s.$$

Hence,

$$|\lambda_i - a_{ss}| \leq |a_{s1} + \ldots + 0 + \ldots a_{sn}| = D_s \qquad (2.2.29)$$

Corollary 2.2.2: If r of the circles form a connected region isolated from all other circles the region contains exactly r eigenvalues.

Finally, one special bound for a matrix with special structure

Theorem 2.2.3: A symmetric matrix A satisfies $\rho(A) = \max_{1\leq i\leq n} |\lambda_i| = ||A||_2$

Proof:

$$||A||_2 = \rho(A^t A)^{\frac{1}{2}} = [\rho(A^2)]^{\frac{1}{2}} = [\rho^2(A)]^{\frac{1}{2}} = \rho(A) \qquad (2.2.30)$$

Indicating that $\rho(A)<1$ is a necessary and sufficient condition for convergence of A for a symmetric matrix.

We can now consider some special properties of matrices

PO: the INVERSE of a square matrix A is a square matrix A^{-1} such that $AA^{-1}=A^{-1}A=I$ (i.e. commutative matrix product). Matrices with inverses are termed *nonsingular* those without *singular*. A matrix is singular if $\det(A)=0$. Also $\det(A)\det(A^{-1})=\det(I)$.

P1: Cramer's Rule. The inverse of a 2×2 matrix is given as follows,

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}^{-1} = \frac{1}{ad-bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

and det(A)=ad-bc, so A is singular if a=c=0 or d=b=0, etc.

REMARK: Although similar rules are available for large matrices for n>3 they are very complex and faster methods are available.

P2: a DIAGONALLY DOMINANT n×n matrix A is one where,

$$|a_{ii}| \geq \sum_{\substack{j=1 \\ i \neq j}}^{n} |a_{ij}| \qquad i,j=1(1)n.$$

If $\geq$ is replaced by >, A is *strictly diagonally dominant.*

P3: a POSITIVE DEFINITE matrix A, is a real symmetric matrix for which $x^t Ax>0$ for every $x \neq 0$ from $\mathbb{R}^n$.

P4: an IRREDUCIBLE n×n matrix for n≥1 and two non-empty disjoint subsets S and T of W (a set comprising the first n positive integers) such that S+T=W satisfies $a_{ij} \neq 0$ with $i \in S$, $j \in T$.

P5: ORTHOGONALITY a matrix A is orthogonal if $A^{-1}=A^T$.

P6: SIMILARITY. Two matrices A and B are said to be similar if a non-singular matrix S exists such that $A=S^{-1}BS$.

If we denote λ and eigenvalue of A and x the corresponding eigenvector,

$$Ax = \lambda x ,$$

and

$$SAS^{-1}Sx = \lambda Sx$$

$$B(Sx) = \lambda(Sx)$$

Thus if B is similar to A, it has eigenvalue λ and eigenvector Sx.

2.3 DIRECT METHODS FOR THE SOLUTION OF LINEAR SYSTEMS

The previous two sections have introduced a mathematical basis for Linear Systems of the form given in (2.2.4), i.e.,

$$Ax = b \ . \qquad (2.3.1)$$

We now turn to methods of constructing the solution of the system, and assume for convenience that A is non-singular (A^{-1} exists) so that the solution is unique. The methods discussed are intended for use on computers and so the solution of a system is generally only approximate due to rounding errors introduced by the finite word (length) calculations. However, the growth of errors is bounded in practice and results are acceptable especially if double precision arithmetic is used.

The choice of solution method depends on a number of factors including structure and size of the matrix A, the number of arithmetic operations required to construct the solution, the amount of storage available/required for (2.3.1), and the control of rounding error growth (or stability).

In this section *direct methods* of solution are considered, which are applicable to small dense matrices, and have the advantage of producing a solution after a fixed number of operations proportional to the matrix order. Furthermore, in most cases the accuracy of the solution is usually stable and adequate for our purposes.

Equation (2.3.1) is an example of an *implicit* system, the solution vector cannot be derived without modifications to the system, which preserve the solution, and also give access to the unknowns.

A brute force approach is to solve (2.3.1) by converting it to an equivalent *explicit* form, using the fact that A is non-singular. This yields,

$$x = A^{-1}b \ , \qquad (2.3.2)$$

and x is constructed explicitly by matrix vector multiplication. This implicit-explicit conversion is fine if A^{-1} is already known, or easily constructed but generally this is not the case. Instead direct methods

are aimed at a compromise which manipulates A and b to produce a *semi-explicit* form,

$$\bar{A}x = \bar{b} \ , \tag{2.3.3}$$

where x can be derived from an ordered substitution process, and $\bar{A}$ is a matrix with an easily solvable structure, and $\bar{b}$ is a modified RHS.

2.3.1 Forward/Backward Substitution

Linear systems which have upper or lower triangular matrix structures (i.e. S3 and S4) automatically form semi-explicit solution schemes and so are easily solvable. With A=L we write,

$$Lx = b \ , \tag{2.3.1.1}$$

and with A=U

$$Ux = b \tag{2.3.1.2}$$

and the process of forming x is termed forward substitution for (2.3.1.1) and backward substitution for (2.3.1.2). As an example in the latter case, the problem can be expanded to give a coefficient form,

$$\begin{bmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ & \ddots & & \vdots \\ 0 & & & u_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \tag{2.3.1.3}$$

$$u \qquad\qquad x \qquad b$$

and the backsubstitution formula is,

$$\left.\begin{aligned} x_n &= b_n/u_{nn} \\ x_i &= \{b_i - \sum_{j=i+1}^{n} u_{ij}x_j\}/u_{ii} \ , \quad i=n-1,\ldots,1. \end{aligned}\right\} \tag{2.3.1.4}$$

The method could fail if some $u_{ii}=0$, i=1(1)n, but implies from Definition (2.2.5) that Det(U)=0 which with property P0 contradicts the assumption

that A was non-singular. The amount of work involved in back-substitution is assessed by counting the number of scalar operations, and from (2.3.1.4) this is,

$$\left.\begin{array}{ll} \text{Mults/Divs} & 1 + \sum_{i=1}^{n-1} ((n-i)+1) = \frac{n}{2}(n+1) \\ \text{Adds/Subs} & \sum_{i=1}^{n-1} ((n-i-1)+1) = \frac{n}{2}(n-1) \end{array}\right\} \qquad (2.3.1.5)$$

A similar formula with the same operation counts can be derived for the forward substitution process (with A=L).

Given this simple technique of solving triangular systems, direct methods are developed from the simple supposition that converting a general matrix to an easily solvable triangular form, by operations preserving the solution will minimise the amount of computational work. Consequently direct methods are divided into two forms:

(i) Triangularisation Methods: which convert A to L or U form.

or (ii) Factorisation (or decomposition) Methods: which replace A by a product of Triangular matrices.

2.3.2 Matrix Triangularisation

The most popular method for triangularising a matrix is *Gaussian Elimination* which is based upon the use of three operations on the rows of A with form like (2.2.1) which preserve the solution vector. The rules are:

(i) row_i can be multiplied by any non-zero constant α and the result used in place of row_i (i.e., scalar multiplication of a row vector).

(ii) Two rows, row_i and row_j can be added together and used to replace one of the rows, that is, $row_i = row_i + row_j$, or $row_j = row_i + row_j$ (alternatively vector addition).

(iii) Two rows can be interchanged.

The rules are applied to an augmented matrix $\bar{A}$, (which is an $n \times n+1$ matrix constructed by making b an additional column of A), to form a sequence of modified matrices $\bar{A}^{(1)}, \bar{A}^{(2)}, \ldots, \bar{A}^{(k)}$ for $k=1(1)n$ and when $k=1$, $\bar{A}^{(1)} = A$

$$k>1, \quad a_{ij}^{(k)} = \begin{cases} a_{ij}^{(k-1)} & i=1(1)k-1,\ j=1(1)n+1 \\ 0 & i=k(1)n,\ j=1(1)k-1 \\ a_{ij}^{(k-1)} - \left(\dfrac{a_{i,k-1}^{(k-1)}}{a_{k-1,k-1}^{(k-1)}}\right) a_{k-1,j}^{(k-1)} & i=k(1)n,\ j=k(1)n+1 \end{cases} \quad (2.3.2.1)$$

when $k=n$, $\bar{A}^{(k)}$ is upper triangular, when the $(n+1)$th column the modified b vector is removed.

The value $\left(\dfrac{a_{i,k-1}^{(k-1)}}{a_{k-1,k-1}^{(k-1)}}\right)$ is termed the *multiplier* and there is a separate value associated with each element set to zero. Thus, multipliers can be stored in the strictly lower triangular portion of A assumed zero, this is particularly useful when the same matrix is to be used to solve a number of different right hand sides, as only the new vector has to be modified.

Generally, the $A^{(k)}$ matrix has the form,

$$\left[\begin{array}{ccccccc|c} a_{11}^{(1)} & a_{12}^{(1)} & \cdots & a_{1,k-1}^{(1)} & & & a_{1,n}^{(1)} & a_{1,n+1}^{(1)} \\ 0 & a_{22}^{(2)} & & & & & & \\ & & \ddots & & & & & \\ & & & a_{k-1,k-1}^{(k-1)} & a_{k-1,k}^{(k-1)} & \cdots & a_{k-1,n}^{(k-1)} & a_{k-1,n+1}^{(k-1)} \\ & & & 0 & a_{kk}^{(k)} & \cdots & a_{kn}^{(k)} & a_{k,n+1}^{(k)} \\ & & & & \vdots & & \vdots & \vdots \\ 0 & & \cdots & 0 & a_{n,k}^{(k)} & \cdots & a_{nn}^{(k)} & a_{n,n+1}^{(k)} \end{array}\right] \quad (2.3.2.2)$$

at the end of the modifications in (2.3.2.1), and it is trivially observed that if any $a_{ii}^{(i)}=0$, i=1(1)n the method breaks down as multipliers are impossible to form. The element $a_{kk}^{(k)}$ of (2.3.2.2) is the crucial value in forming $A^{(k+1)}$ and is termed the *pivot*, it follows that if the pivot is zero the method fails, and the use of methods to avoid failure are termed pivoting strategies. Notice also, that (2.3.2.1) requires the application of only the first two solution preserving rules, the essential idea of pivoting is to replace zero pivots by non-zero values, using the third rule to interchange rows. Adopting pivoting ensures that Gaussian elimination fails only if A was originally singular, and we assumed A was non-singular. The extensions of the pivoting strategy can be used to control the stability of the method, in this case the pivot value is swapped if a better one (i.e. compresses rounding error more) can be found, and gives rise to a number of pivoting strategies.

(i) MAXIMAL COLUMN (or PARTIAL) PIVOTING:

This is the simplest method and selects an element in the same column as the pivot but below it with the largest absolute value, and swaps its associated row with the one containing the pivot.

(ii) SCALED COLUMN PIVOTING:

This is the same as (i) but scales the rows first by dividing the row entries by the maximum row element before choosing the element.

(iii) MAXIMAL (total) PIVOTING:

This is the most general method, selecting an element by scaling the remaining rows, and then using row and column interchanges to produce the best pivot value.

Although pivoting is useful for avoiding breakdown and controlling rounding error they involve extra work and it is desirable to keep this

to a minimum, hence the simplest strategy that can be used is adopted. The total number of operations for Elimination without pivoting is

$$\left.\begin{array}{ll} \text{Mults/divs} & \sum_{i=1}^{n-1} (n-i)(n-i+2) = \frac{2n^3+3n^2-5n}{6} \\ \text{Adds/subs} & \sum_{i=1}^{n-1} (n-i)(n-i+1) = \frac{n}{3}(n^2-n) \end{array}\right\} \qquad (2.3.2.3)$$

Thus, the total number of operations to solve (2.3.1) requires the addition of (2.3.1.5) and (2.3.2.3) yielding the expressions,

$$\left.\begin{array}{ll} \text{Mults/divs} & \frac{n^3+3n^2-n}{3} \\ \text{Adds/subs} & \frac{2n^3+3n^2-5n}{6} \end{array}\right\} \qquad (2.3.2.4)$$

A number of variations to Gaussian Elimination are available which minimise time and storage by making use of the special structure of the matrix. The above method works for general non-singular matrices, matrices with banded structures for instance rarely use as many operations as (2.3.2.4).

Other methods for general problems include the Gauss-Jordan algorithm and the Givens (orthogonal) Rotation method. The former scheme follows (2.3.2.1) by a second sequence of matrices which eliminate in the reverse direction to produce a diagonal matrix, which is trivially solved by n divisions and requires a total of

$$\left.\begin{array}{ll} \text{Mults/divs} & \frac{n^3}{2} + n^2 - \frac{n}{2} \\ \text{Adds/subs} & \frac{n^3}{2} - \frac{n}{2} \end{array}\right\} \qquad (2.3.2.5)$$

operations. The Givens orthogonal triangularization method is identical to Gaussian Elimination except that the equation,

$$a_{ij}^{(k)} = a_{i,j}^{(k-1)} - \frac{a_{i,k-1}^{(k-1)}}{a_{k-1,k-1}^{(k-1)}} a_{k-1,j}^{(k-1)} \quad \text{in} \quad (2.3.2.1)$$

is replaced by the 2×2 rotation matrix operation

$$\begin{bmatrix} c & -s \\ s & c \end{bmatrix} \begin{bmatrix} a_{k-1,j}^{(k-1)} \\ a_{i,j}^{(k-1)} \end{bmatrix} = \begin{bmatrix} a_{k-1,j}^{(k)} \\ 0 \end{bmatrix}$$

with $$sa_{k-1,j}^{(k-1)} + ca_{ij}^{(k-1)} = 0$$

$$s^2 + c^2 = 1$$

and $$c = a_{k-1,j}^{(k-1)}/\Delta \quad , \quad s = a_{ij}^{(k-1)}/\Delta$$

$$\Delta = \{[a_{ij}^{(k-1)}]^2 + [a_{k-1,j}^{(k-1)}]^2\}^{\frac{1}{2}} \qquad (2.3.2.6)$$

which requires more basic operations than the elimination scheme, but requires no pivoting to remain stable.

2.3.3 MATRIX FACTORISATION:

Factorisation is an alternative to triangularisation which avoids modification of the rhs of (2.3.1) and so is more convenient for multiple rhs solutions. Generally it also avoids pivoting. The idea is to replace A by the product of a lower and upper triangular matrix,

$$A = LU \ , \qquad (2.3.3.1)$$

which on substitution for A and the introduction of an auxiliary vector y produces an answer by solving the two coupled systems,

$$\begin{aligned} &a) \quad Ly = b \\ &b) \quad Ux = y \end{aligned} \qquad (2.3.3.2)$$

by forward and backward substitution respectively. A number of methods for producing L and U factors which satisfy (2.3.3.1) are known and can be classified according to whether the diagonal element ℓ_{ii} or u_{ii} are set to 1 or equal ($\ell_{ii}=u_{ii}$), they are:

(i) Doolittles' method which ($\ell_{ii}=1$, i=1(1)n) formulated as

$$\left.\begin{aligned} u_{ij} &= a_{ij} - \sum_{k=1}^{i-1} \ell_{ik}u_{kj} \ , \quad j\geq i \\ \ell_{ji} &= \frac{1}{u_{ii}} \left[a_{ji} - \sum_{k=1}^{i-1} \ell_{jk}u_{ki}\right] \quad j>i \ , \ i=1(1)n \end{aligned}\right\} \qquad (2.3.3.3)$$

(ii) Crouts' method with ($u_{ii}=1$, i=1(1)n) given as,

$$\left.\begin{aligned} u_{ij} &= \frac{1}{\ell_{ii}} \left[a_{ij} - \sum_{k=1}^{i-1} \ell_{ik}u_{kj}\right] \quad j\geq i \\ \ell_{ji} &= a_{ji} - \sum_{k=1}^{i-1} \ell_{jk}u_{ki} \quad j>i, \ i=1(1)n \end{aligned}\right\} \qquad (2.3.3.4)$$

(iii) Choleski's method ($\ell_{ii}=u_{ii}$ effectively)

$$\left.\begin{aligned} \ell_{ii} &= \left(a_{ii} - \sum_{k=1}^{i-1} \ell_{ik}^2\right)^{\frac{1}{2}} \ , \quad i=j \\ \ell_{ij} &= \left(a_{ji} - \sum_{k=1}^{i-1} \ell_{ik}\ell_{jk}\right)/\ell_{ii} \ , \ i>j, \ j=1(1)n \end{aligned}\right\} \qquad (2.3.3.5)$$

Note that if any ℓ_{ii} or u_{ii} is zero the methods breakdown. The formulae (2.3.3.3) and (2.3.3.4) are general methods for non-singular matrices, while (2.3.3.5) is applicable only for symmetric positive definite matrices and $A=LL^T$. A root-free form of the Choleski with $A=LDL^T$ (D a diagonal matrix) is also possible. Due to the fact that only the entries of L are computed in (2.3.3.5) savings in memory and computation time can be made over Gaussian Elimination. The number of operations are,

$$\left.\begin{aligned} &\text{square roots} && n \\ &\text{Mults/divs} && \frac{n^3+9n^2+2n}{6} \\ &\text{Adds/subs} && \frac{n^3+6n^2-7n}{6} \end{aligned}\right\} \qquad (2.3.3.6)$$

but requires the relatively complex square root calculation, while Doolittle's and Crout's method use approximately the same amount of

computation as Gaussian Elimination.

To define the types of matrices which can be solved using triangularisation and factorisation methods consider the following two theorems.

Theorem 2.3.1: If A is an n×n strictly diagonally dominant matrix or positive definite A is non-singular and Gaussian Elimination can be performed without pivoting and remains stable against the growth of rounding errors.

Theorem 2.3.2: If Gaussian elimination can be performed on a system Ax=b without row interchanging, A can be factorised into the form A=LU.

Further improvements can be made to the methods if A has a regular structure (like symmetry) or is banded. But as A becomes larger and less dense the above methods produce fill-in (replacing zeros by non-zeros) increasing the amount of computation and storage. If the matrix structure is irregular fill-in occurs in predictable places and we look for alternative optimised methods of solution.

2.4 ITERATIVE SOLUTION OF LINEAR SYSTEMS

Iterative methods preserve the sparse structure of a matrix, but do so by computing a sequence of approximations which converge (Definition (2.1.4)) to the solution. The methods can provide arbitrary accuracy depending on the number of iterations performed, and are terminated usually when the difference between successive approximations satisfies some tolerance.

To form an iterative scheme we split A (from (2.3.1)) into matrices E and F such that,

$$A = E-F \ , \qquad (2.4.1)$$

which produces, $$Ex = Fx + b \ . \qquad (2.4.2)$$

Now if $x^{(0)}$ is some arbitrary selected start vector, and $x^{(n)}$ denotes the n^{th} approximation to x (the exact solution), the iterative scheme is,

$$Ex^{(n)} = Fx^{(n-1)} + b \qquad (2.4.3)$$

and providing E^{-1} exists, $x^{(n)}$ can easily be found. The amount of work to construct $x^{(n)}$ depends on the structure of E and F, in the light of Section 2.3, the sensible thing to do is to restrict E and F to easily solvable matrices. If E is a Diagonal matrix the so-called *simultaneous displacement methods* (e.g. Jacobi, Richardson) result, when E is lower triangular *successive displacement methods* (e.g. Gauss-Seidel, SOR) are produced. In the simultaneous case the order that the components of $x^{(n)}$ are updated is unimportant, while in the successive case a sequential modification order is imposed.

2.4.1 Simultaneous Displacement Methods

Consider the system A=D-L-U=D-B with B=L+U, put E=D and F=B and substitute in (2.4.2)

$$Dx^{(n)} = Bx^{(n-1)} + b$$

and $$x^{(n)} = D^{-1}Bx^{(n-1)} + D^{-1}b \qquad (2.4.1.1)$$

The Jacobi method. Now substitute for B=D-A to give,

$$x^{(n)} = (I-D^{-1}A)x^{(n-1)} + D^{-1}b \qquad (2.4.1.2)$$

which is also,

$$x^{(n)} - x^{(n-1)} = D^{-1}(b-Ax^{(n-1)}) \qquad (2.4.1.3)$$

illustrating that the difference between successive approximations is proportional to the difference between the true solution (x) and the $x^{(n-1)}$ estimate. If we define $\alpha \in \mathbb{R}$ a scalar and $r^{(n-1)} = D^{-1}(b-Ax^{(n-1)})$ convergence to x can be accelerated by the formula,

$$x^{(n)} = x^{(n-1)} + \alpha r^{(n-1)} \qquad (2.4.1.4)$$

known as the Simultaneous Displacement method. Choice of the 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 (2.4.1.4) which is more sensitive to the errors in approximation is to define $\alpha_i \in \mathbb{R}$ for each iteration giving Richardsons method,

$$x^{(n)} = x^{(n-1)} + \alpha_n r^{(n-1)} \; . \qquad (2.4.1.5)$$

The simultaneous method (2.4.1.4) is termed *stationary* because the error of approximation is always affected by the same amount (α), while (2.4.1.5) is nonstationary as the error at each iteration is affected differently due to the changing α_i. Consequently the choice of α_i is generally more difficult than in the stationary case.

2.4.2 Successive Displacement Methods

To derive the Gauss-Seidel method set A=D-L-U and E=D-L with F=U to give,

$$(D-L)x^{(n)} = Ux^{(n-1)} + b \; , \qquad (2.4.2.1)$$

consequently,

$$x^{(n)} = (D-L)^{-1}Ux^{(n-1)} + (D-L)^{-1}b. \qquad (2.4.2.2)$$

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

As for simultaneous methods, an acceleration parameter $\omega \in \mathbb{R}$ can be introduced deriving the Successive Overrelaxation (SOR) scheme

from (2.4.2.1)

$$D(x^{(n)}-x^{(n-1)}) = Lx^{(n)}+Ux^{(n-1)}+b-Dx^{(n-1)}$$

so

$$(x^{(n)}-x^{(n-1)}) = D^{-1}[Lx^{(n)}+Ux^{(n-1)}+b-Dx^{(n-1)}]$$

introducing the acceleration parameter gives,

$$(x^{(n)}-x^{(n-1)}) = \omega D^{-1}[Lx^{(n)}+Ux^{(n-1)}+b-Dx^{(n-1)}]$$

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

and

$$x^{(n)} = (I-\omega D^{-1}L)^{-1}\{\omega D^{-1}U+(1-\omega)\}x^{(n-1)}+(I-\omega D^{-1}L)^{-1}\omega D^{-1}b \quad (2.4.2.3)$$

2.4.3 Convergence of Iterative Schemes

Now (2.4.1.1), (2.4.2.2) and (2.4.2.3) can be represented by the general iterative form,

$$x^{(n)} = Mx^{(n-1)} + c \ , \quad (2.4.3.1)$$

where M is called the *iteration matrix* and c is a vector, which for the above schemes take the forms,

$M = (I-D^{-1}A) = D^{-1}B$, $c=D^{-1}b$ Jacobi

$M = (D-L)^{-1}U$, $c=(D-L)^{-1}b$ Gauss-Seidel

$M = (I-\omega D^{-1}L)^{-1}\{\omega D^{-1}U+(1-\omega)\}$, $c=(I-\omega D^{-1}L)^{-1}\omega D^{-1}b$ SOR

The error vector $e^{(i)}$ associated with the ith iterate $x^{(i)}$ is

$$e^{(i)} = x-x^{(i)} \quad (2.4.3.2)$$

and with x the exact solution substituted in (2.4.3.1) we have,

$$x = Mx + c \ , \quad (2.4.3.3)$$

Subtraction of (2.4.3.1) from (2.4.3.3) produces,

$$x-x^{(i)} = M(x-x^{(i-1)}) \ . \quad (2.4.3.4)$$

Hence, $e^{(i)} = Me^{(i-1)}$

and consequently

$$e^{(n)} = Me^{(n-1)} = \ldots = M^n e^{(0)}$$

using consistent and compatible norms produces,

$$\begin{aligned} ||e^{(n)}|| &\leq ||M^{(n)}||\ ||e^{0}|| \\ &\leq ||M||^n ||e^{0}|| \end{aligned} \qquad (2.4.3.5)$$

Thus from Definition (2.2.4) and Corollary (2.2.1) $||M||<1$ for the error to decrease and the sequence of approximations $x^{(i)}$ to converge to x for an arbitrary starting vector $x^{(0)}$.

The error vectors of (2.4.1.4) and (2.4.1.5) satisfy,

$$\left.\begin{aligned} e^{(n)} &= (I-\alpha A)^n e^{(0)} \\ \text{and} \quad e^{(n+1)} &= \prod_{i=0}^{n} (I-\alpha_i A) e^{(0)} \end{aligned}\right\} \qquad (2.4.3.6)$$

indicating the stationary and non-stationary nature.

Different methods produce different rates of convergence and this together with the amount of work required for each iteration dictates which method is used for particular problems. For instance, a more complicated iteration method may converge significantly faster than a simple one, but involve much more work per iteration; unless the amount of work in two competing methods is approximately the same a simpler iteration scheme could out-perform a complex one in terms of total number of operations. Choice of iterative method is further complicated by the selection of a good initial approximation $x^{(0)}$, a bad choice, some distance from x can force even an efficient method to perform large amounts of computation. The definitions and theorems below formalise these concepts allowing a numerical method of analysis.

Definition 2.4.1: Average Rate of Convergence

Let A and B be two n×n matrices. If, for some positive integer $m, ||A^m||<1$ then,

$$R(A^m) \equiv -\ell n[(||A^m||)^{1/m}] = -\frac{\ell n||A^m||}{m} \qquad (2.4.3.7)$$

is the average rate of convergence, for m iterations of A. If $R(A^m) < R(B^m)$, B is iteratively faster for m iterations. Note also that the number of iterations is inversely proportional to the average rate of convergence. We can also distinguish which methods are better for large numbers of iterations when $m \to \infty$ by

Theorem 2.4.1: Let A be a convergent n×n matrix. For all m sufficiently large the average rate of convergence for m iterations $R(A^m)$ satisfies,

$$\lim_{m\to\infty} R(A^m) = -\ell n\rho(A) = R_\infty(A) \qquad (2.4.3.8)$$

where $R_\infty(A)$ denotes the *Asymptotic Rate of Convergence*.

Proof:

$$R(A^m) \equiv -\frac{\ell n||A^m||}{m} \approx -\frac{\ell nV}{m} - \frac{\ell n\binom{m}{p-1}}{m} - \left[\frac{m-p+1}{m}\right]\ell n\rho(A)$$

with $$\binom{m}{p-1} = \frac{m!}{(p-1)!(m-p+1)!} \quad \text{and} \quad \frac{\ell n\binom{m}{p-1}}{m} \to 0 \text{ as } m\to\infty.$$

Thus,

$$\lim_{m\to\infty} R(A^m) = -\ell n\rho(A) \equiv R_\infty(A)$$

Furthermore $R_\infty(A) \geq R(A^m)$ for any $||A^m|| < 1$

An indication of the relationship between (2.4.3.1) and (2.3.2) the explicit solution of (2.3.1) is contained in the following proofs:

Theorem 2.4.2: If the spectral radius of an n×n matrix M satisfies $\rho(M) < 1$ then $(I-M)^{-1}$ exists and

$$(I-M)^{-1} = I+M+M^2+ \ldots \qquad (2.4.3.9)$$

(the righthand side of (2.4.3.9) is called the *Neumann expansion*.)

Proof:

(i) Let λ be an eigenvalue of M then $1-\lambda$ is an eigenvalue of I-M.

(ii) If $|\lambda| \leq \rho(M) < 1$ then I-M can have no zero eigenvalues and hence is nonsingular (see characteristic polynomial Definition (2.2.6) and Property PO).

(iii) Put $S_m = I+M+M^2+\ldots+M^m$

and $(I-M)S_m = I-M^{m+1}$

as $\rho(M)<1$, M is convergent (Definition 2.2.4) hence,

$$\lim_{m\to\infty} (I-M)S_m = \lim_{m\to\infty} (I-M^{m+1}) = I$$

$$\lim_{m\to\infty} S_m = (I-M)^{-1}$$

(iv) Now

$$x^{(k)} = Mx^{(k-1)} + c$$

and $$x^{(k)} = M^k x^{(0)} + (M^{(k-1)}+M^{(k-2)}+\ldots+M+I)c \qquad (2.4.3.10)$$

for k large enough $M^k=0$ hence,

$$x \approx S_{k-1}c = (I-M)^{-1}c \qquad (2.4.3.11)$$

using (2.4.3.5) a rough bound on the number of iterations is given by,

$$||x-x^{(k)}|| \leq ||M^k||\ ||x-x^{(0)}||$$

and with $\rho(M)\leq||M||$

$$||x-x^{(k)}|| \approx \rho(M)^k||x-x^{(0)}||$$

and with the initial guess $x^{(0)}=0$ the relative error is derived

$$\frac{||x-x^{(k)}||}{||x\ \ ||} \approx \rho(M)^k$$

and with a tolerance of 10^{-t} defining the error of approximation termination occurs with $\rho(M)^k \leq 10^{-t}$ and

$$k \geq \frac{-t}{\log_{10}\rho(M)} \qquad (2.4.3.12)$$

Consequently if the iteration matrix is convergent, it will converge to the correct solution for $x^{(0)}=0$, which gives a simple starting vector. From (2.4.3.12) we conclude that the smaller the spectral radius the faster convergence will be. For the Jacobi and Gauss-Seidel methods we can prove the following.

Theorem 2.4.3: If A is strictly diagonally dominant then for any choice

of $x^{(0)}$ both Jacobi and Gauss-Seidel methods give vector sequences $\{x^{(k)}\}_k^{\infty}$ which converge to x the solution of Ax=b.

Proof:

(i) Jacobi: Iteration matrix $M=D^{-1}(L+U)$ and for convergence $||M||<1$

thus,
$$||D^{-1}(L+U)|| \leq ||D^{-1}|| \; ||(L+U)|| < 1$$

and,
$$||L+U|| < \frac{1}{||D^{-1}||}$$

$$||L+U|| < ||D|| \text{ as } ||D|| \; ||D^{-1}|| > ||I|| = 1$$

and this is a norm representation of property P2.

(ii) Gauss-Seidel: $M=(D-L)^{-1}U$ so for $||M||<1$

$$||(D-L)^{-1}U|| \leq ||(D-L)^{-1}|| \; ||U|| < 1$$

$$||U|| < ||(D-L)|| < ||D|| - ||L||$$

hence, $$||L+U|| \leq ||L|| + ||U|| < ||D||$$

Thus from Theorem (2.4.2) part (iv) both methods converge for any $x^{(0)}$.

Finally the last few theorems indicate bounds on the acceleration parameters of (2.4.1.4) and (2.4.2.3).

Theorem 2.4.4: The optimal value α for the simultaneous replacement method $x^{(n)} = x^{(n-1)} + \alpha r^{(n)}$ with $r^{(n)} = D^{-1}(b-Ax^{(n-1)})$ is $\alpha=2/(a+b)$ where a and b are the largest and smallest eigenvalues of A.

Proof:

Assume $(I-D^{-1}A)$ has n-linearly independent eigenvectors v_i associated with n distinct eigenvalues λ_i and let μ_i be eigenvalues of $D^{-1}A$, for i=1(1)n. Then the method converges with $\rho(I-\alpha D^{-1}A)<1$ and by definition $a \leq \mu_i \leq b$ and $\lambda_i = 1-\alpha\mu_i$ for i=1(1)n.

Consequently, $$|1-\alpha\mu_i| < 1 \; ,$$

and $$0 < \alpha < \frac{2}{b} . \qquad (2.4.3.13)$$

Now to achieve optimal convergence we minimise $\rho(I-\alpha D^{-1}A)$ resulting in

$$|1-\alpha a| = -|1-\alpha b| \Rightarrow \alpha = \frac{2}{a+b} \qquad (2.4.3.14)$$

and so,

$$|1-\alpha\mu_i| \leq \frac{b-a}{b+a} = \frac{(\frac{b}{a})-1}{(\frac{b}{a})+1} < 1 , \qquad (2.4.3.15)$$

and from Definition (2.2.8) minimising the spectral radius is related to the P-condition number of the iteration matrix. For the SOR method ω is characterised by the results.

Theorem 2.4.5: (Kahan) If $a_{ii} \neq 0$ i=1(1)n for a matrix A and iteration matrix M_ω, $\rho(M_\omega) > |\omega-1|$ hence,

$$\rho(M_\omega) < 1 \quad \text{iff } 0<\omega<2$$

Proof: (omitted).

Theorem 2.4.6: (Ostrowski-Reich) If A is a positive definite matrix and $0<\omega<2$ the SOR method converges for any initial approximation $x^{(0)}$.

Let M_j and M_g be the iteration matrices of (2.4.1.1) and (2.4.2.2) respectively.

Theorem 2.4.7: If A is a positive definite tridiagonal matrix $\rho(M_g) = [\rho(M_j)]^2 < 1$ and the optimal value of ω in SOR is,

$$\omega = \frac{2}{1+\sqrt{1-\rho(M_j)^2}}$$

with $\rho(M_\omega) = \omega - 1$,

which indicates that the Gauss-Seidel method is iteratively faster than Jacobi's method.

Theorem 2.4.8: (Stein & Rosenberg) If $a_{ij} \leq 0$ for $i \neq j$ and $a_{ii} > 0$ for i,j=1(1)n then one and only one of the following is true:

(i) $0 \leq \rho(M_g) < \rho(M_j) < 1$

(ii) $1 < \rho(M_j) < \rho(M_g)$

(iii) $\rho(M_j) = \rho(M_g) = 0$

(iv) $\rho(M_j) = \rho(M_g) = 1$

Finally, we have made one critical and implicit assumption about the iterative schemes discussed, and which is termed the 'consistency condition'. That is, when x is substituted for $x^{(n)}$, $x^{(n+i)}$ $i \geq 1$ are also solutions. Consequently, when the method converges it is assumed that it does not diverge on subsequent iterations.

2.5 PARTIAL DIFFERENTIAL EQUATIONS

Almost all problems involving rates of change of two or more independent variables representing some physical quantity (e.g. time, length, etc.), leads to a partial differential equation (PDE) which can be written in a general form as,

$$a\frac{\partial^2 u}{\partial x^2} + b\frac{\partial^2 u}{\partial x \partial y} + c\frac{\partial^2 u}{\partial y^2} + d\frac{\partial u}{\partial x} + e\frac{\partial u}{\partial y} + fu + g = 0. \qquad (2.5.1a)$$

The variables a,b,c,d,e,f and g are called coefficients and can be zero, or functions of the independent variables x and y and also the dependent variable u. When coefficients are composed only of functions involving x and y (2.5.1a) is termed *linear*, but if they also contain terms with u or its derivatives they are called *non-linear* equations. It is possible to classify PDE's further and (2.5.1a) is termed,

$$\left.\begin{array}{l} \text{elliptical when } b^2-4ac < 0 \\ \text{parabolic when } b^2-4ac = 0 \\ \text{hyperbolic when } b^2-4ac > 0 \end{array}\right\} \qquad (2.5.1b)$$

We also make the implicit assumption that all terms in (2.5.1) can be formed and this implies that the solution function u is twice

differentiable and continuous in a bounded space called a region (denoted R). Associated with R is a boundary (denoted C) which defines the limits of R, and generally we are not interested in solving (2.5.1) beyond the boundary. Corresponding to (2.5.1) in R, a set of boundary conditions are attached to C which are functions describing the behaviour of u at the periphery of the region. The two main types of boundary conditions are *specific* and *general* boundary conditions. With specific boundary conditions, the dependent variables can be assigned specific values at specific points on C, and can be further partitioned into *homogenous* and *non-homogenous* types. Specific homogenous boundary conditions are such that if $u=f_1$, (where f_1 is a function on the boundary) $\alpha u=\alpha f_1$ is satisfied for some parameter α, any other specific condition not obeying the relation is non-homogenous. On the other hand, general boundary conditions arise when the behaviour is unpredictable on C. For instance, we might want a solution to u in R_3, a region made up of two subregions R_1 and R_2 which in physical terms constitute different mediums, the boundary between R_1 and R_2 may be uncertain but usually functions can be derived using values not on but adjacent to the boundary in both regions, these functions are then interpreted as C. In this thesis the partial differential equations will possess specific boundary conditions.

Given a partial differential equation we can define four main types of boundary value problems (which amount to solving the equation under different boundary conditions), they are:

(i) Dirichlet problem where u is specified at every point on C

(ii) Neumann problem where only values of the normal derivative are given on C.

(iii) Robbins problem where a linear combination of u and its derivatives is given on C.

(iv) Mixed problem where u is given for part of C and the normal derivative for the remainder leading to a discontinuous solution near the boundary.

In addition to the boundary conditions many problems also define *initial conditions* which describe the state of the physical problem at some stage. In problems where time (t) is one of the independent variables, t=0 (zero-time) is adopted for the instant when the initial conditions are valid, and is the starting point for the solution of the equation. The boundary and initial conditions arise from the physical constraints of the problem rather than from the form of eqn. (2.5.1) and are sometimes termed 'auxiliary conditions'.

Definition 2.5.1: A partial differential equation is said to be *well-posed* if its auxiliary conditions are specified in such a way that there exists a unique solution, and that small changes in the auxiliary conditions transmit only small changes to the solution.

Parabolic and hyperbolic equations are derived mainly from problems which can define time as an independent variable and so possess initial conditions. For instance, the simplest parabolic problem is,

$$\frac{\partial u}{\partial t} = k \frac{\partial^2 u}{\partial x^2} \qquad (2.5.2)$$

derived from the theory of heat diffusion, u is the temperature at a distance x units from one end of a thermally insulated metal bar with length (ℓ) at time t. The initial conditions are clearly the temperature of the rod at t=0, and the boundary conditions the temperature at the ends of the bar (x=0, x=ℓ say).

The simplest hyperbolic equation is that of a vibrating string more generally the wave equation given by,

$$\frac{\partial^2 u}{\partial t^2} = c^2 \alpha^2 \frac{\partial^2 u}{\partial x^2} , \tag{2.5.3}$$

where u is the transverse displacement (of the string) at distance x units from one end of the vibrating string again of length ℓ at time t. This time initial conditions are the initial displacements of the string and its shape and velocity at various points from x=0 to x=ℓ given by u and $\frac{\partial u}{\partial t}$, the boundary conditions are the displacements of the string at its ends (i.e. x=0, x=ℓ).

REMARK: Hyperbolic equations arise generally in vibration problems where there are discontinuities over time (e.g. shock waves with discontinuities in speed, pressure and density).

The simplest and best known elliptic equations are the two-dimensional Poisson and Laplace equations given by,

$$\left.\begin{aligned} &\text{a)} \quad \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + g(x,y) = 0 \\ &\text{and} \\ &\text{b)} \quad \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \end{aligned}\right\} \tag{2.5.4}$$

and associated with steady state or equilibrium problems. The standard physical examples being:

(i) velocity potential for the steady flow of an incompressible non-viscous fluid (modelled by (2.5.4b))

(ii) The electric potential associated with a two-dimensional electron distribution of given charge density (modelled by (2.5.4a)).

Throughout the text we will assume that our problems are well-posed.

The usual requirement for hyperbolic/parabolic equations to be well-posed is that the region R is open in the direction of one of the independent variables (for our purposes t), giving an infinite region. While for elliptical equations all the points on the boundary must be specified and R must be a closed region (see Fig.2.1).

2.5.1 Solution of PDE's Using Finite Differences

In solving an equation of the form (2.5.1) we integrate to produce the function u which is twice differentiable and continuous, and when suitably differentiated produces (2.5.1). There are two main ways to recover the function - the Analytic and Numeric approaches. A purely analytical method is symbolic and attempts to derive an exact finite mathematical formula. The numeric approach, approximates the function with numeric values at various points in R. A cross between purely analytic and numerical approximations are the approximate analytic methods, these replace the finite exact formula of analytic methods by an easier derived formulation often an infinite series, the methods then become approximate as some terms in the series must be neglected producing truncation errors. Analytical methods have the advantage that the character of the solution (at key positions in R) can be easily extracted but are hampered by difficulties in representing boundary value information as regions and boundaries become more complex. We shall use only numerical methods and in particular the finite difference technique, where the solution for a number of points in R is written in tabular form.

The first step towards a finite difference solution is to discretize the region R, this means selecting specific points at which

(2.5.1) will be solved numerically, the solution is then discrete rather than continuous as in the analytic approach. The easiest method of specifying points which cover the whole region is to envisage R as a Cartesian space with independent variable axes (of the space) as illustrated in Fig.(2.1a). Partitioning the axes into uniformly spaced points separated by distances Δx (in the x direction) and Δt (in the t direction) produces a discrete set of points, with $x_o=t_o=0$

$$\left.\begin{aligned} x_i &= x_o+i\Delta x\ , \quad i=0(1)\ell \\ t_j &= t_o+j\Delta t\ , \quad j=0,1,\ldots \end{aligned}\right\} \qquad (2.5.1.1)$$

Drawing the abscissa and ordinates of all these points defines a rectangular grid over R, and the set of solution points are simply the intersecting points of horizontal and vertical lines, and termed grid (nodal or pivotal) points.

The next step is to approximate the solution of each grid point in R, the most popular method is to employ the Taylor expansion. When a function u and its derivatives are single valued, finite and continuous values of x, the Taylor expansion is,

$$u(x\pm h) = u(x) \pm hu'(x) + \frac{h^2}{2!}u''(x) \pm \frac{h^3}{3!}u'''(x) + \ldots \qquad (2.5.1.2)$$

and for a value t

$$u(t\pm k) = u(t) \pm ku'(t) + \frac{k^2}{2!}u''(t) \pm \frac{k^3}{3!}u'''(t) + \ldots \qquad (2.5.1.3)$$

with $h=\Delta x$, $k=\Delta t$, and $u'(x)=\frac{\partial u}{\partial x}$, $u''(x)=\frac{\partial^2 u}{\partial x^2}$, etc.

Forming $u(x+h)+u(x-h)$ and rearranging to isolate $u''(x)$ produces,

$$\left.\begin{aligned} &\text{a)} \quad u''(x) = \frac{1}{h^2}\left\{u(x+h)-2u(x)+u(x-h)\right\} + T \\ &\text{with} \\ &\text{b)} \quad T = -\left\{\frac{2h^2}{4!}u^{(4)}(x) + \frac{2h^4}{6!}u^{(6)}(x) + \ldots\right\} \end{aligned}\right\} \qquad (2.5.1.4)$$

known as the central difference formula. Considering $u(x+h)$ and $u(x-h)$

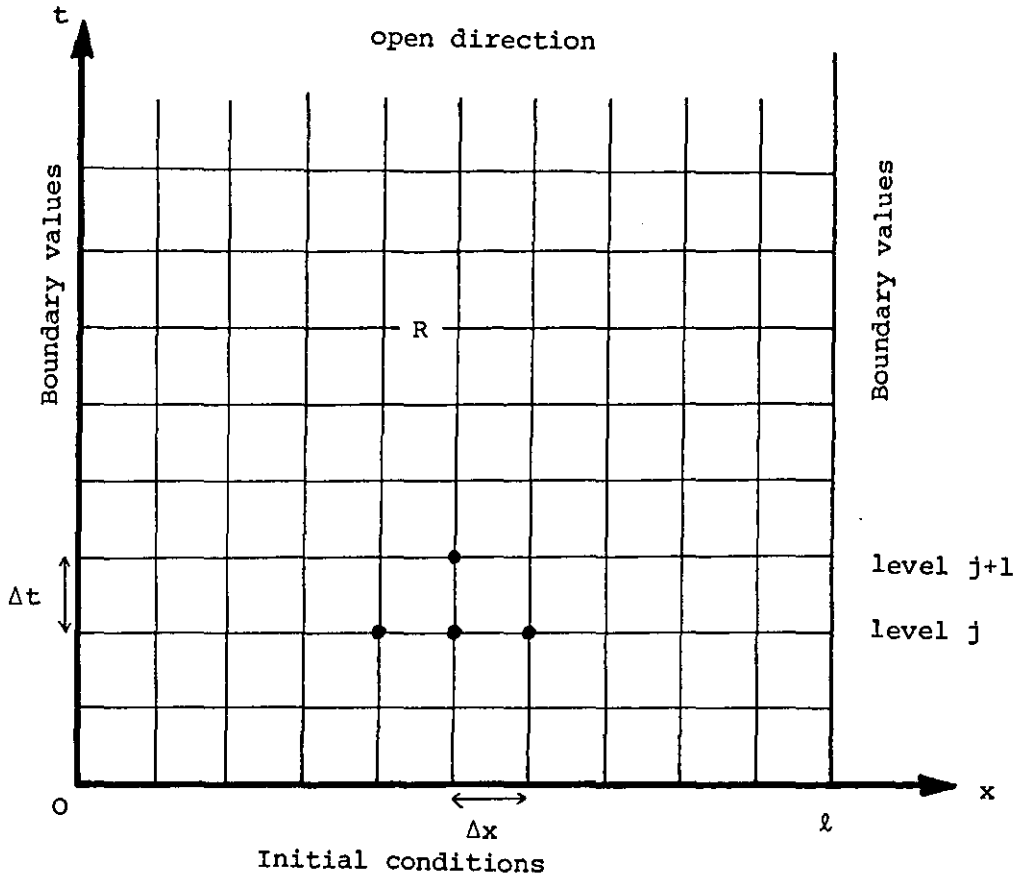


FIGURE 2.1a: Open region for parabolic/hyperbolic equations

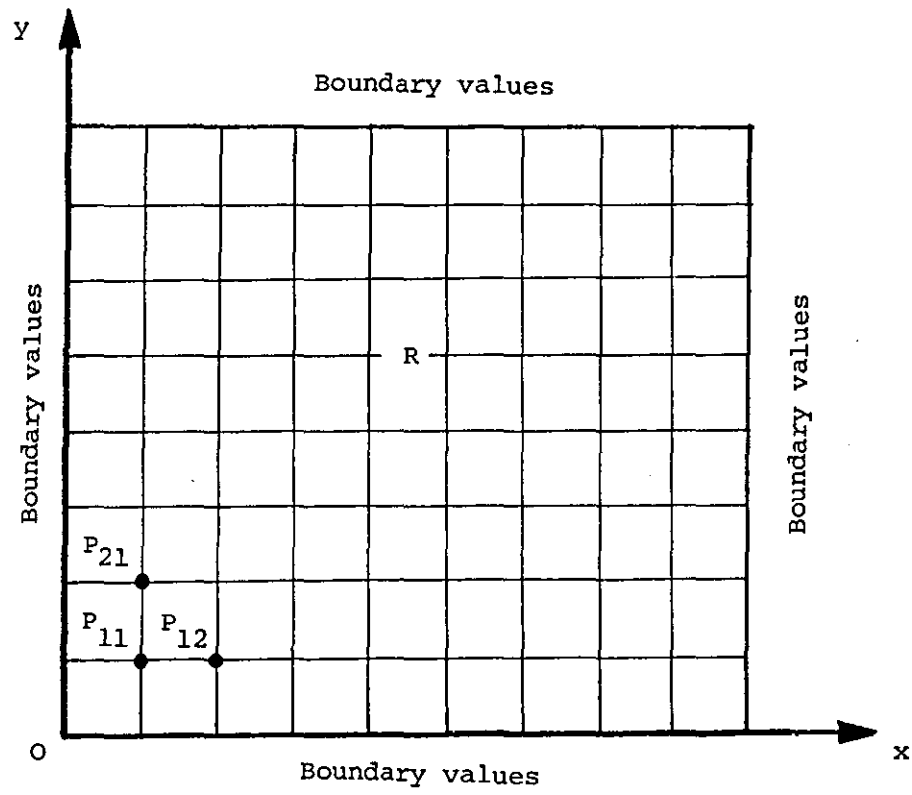


FIGURE 2.1b: Closed region for elliptical problems

individually we have the forward and backward differences

$$\left.\begin{aligned} &\text{a)} \quad u'(x) = \frac{1}{h}\{u(x+h)-u(x)\} + T \\ &\text{b)} \quad T = -\{\frac{h}{2!}u''(x) + \frac{h^2}{3!}u^{(3)}(x) + \ldots\} \end{aligned}\right\} \qquad (2.5.1.5)$$

and

$$\left.\begin{aligned} &u'(x) = \frac{1}{h}\{u(x)-u(x-h)\} + T \\ &T = \{\frac{h}{2!}u''(x) + \frac{h^2}{3!}u^{(3)}(x) + \ldots\} \end{aligned}\right\} \qquad (2.5.1.6)$$

are derived.

The partial derivatives are formed by applying (2.5.1.2) in the x-direction and (2.5.1.3) in the t-direction. If P_{ij} is the grid point at coordinate (ih,jk) then the approximation of u at this point is $u(ih,jk)=u_{ij}$ and it follows that,

$$\begin{aligned} (\frac{\partial^2 u}{\partial x^2})_{ij} &\approx \frac{1}{h^2}\{u((i+1)h,jk)-2u(ih,jk)+u((i-1)h,jk)\} \\ &\approx \frac{1}{h^2}\{u_{i+1,j}-2u_{i,j}+u_{i-1,j}\}\ , \end{aligned} \qquad (2.5.1.7)$$

neglecting the T term from (2.5.1.4) makes the formulation approximate and T is called the *Truncation error*. We denote the error above as $O(h^2)$ indicating that the largest (or principal) term in the truncated part is dominated by h^2. This assumes that the higher derivatives are small relative to powers of h. Likewise,

$$(\frac{\partial u}{\partial x})_{ij} \approx \frac{1}{h}\{u_{i+1,j}-u_{i,j}\} \quad \text{with } O(h) \text{ error} \qquad (2.5.1.8)$$

and,

$$(\frac{\partial^2 u}{\partial t^2})_{ij} \approx \frac{1}{k^2}\{u_{i,j+1}-2u_{i,j}+u_{i,j-1}\} \text{ with } O(k^2) \text{ error} \qquad (2.5.1.9)$$

$$(\frac{\partial u}{\partial t})_{ij} \approx \frac{1}{k}\{u_{i,j+1}-u_{i,j}\} \quad \text{with } O(k) \text{ error} \qquad (2.5.1.10)$$

Now consider the parabolic equation of the form,

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}$$

$$\left.\begin{array}{ll}\text{Initial condition} & u(x,0) = f(x) \\ \text{Boundary conditions} & u(x,0) = u(\ell,0) = 0\end{array}\right\} \qquad (2.5.1.11)$$

At point P_{ij} (2.5.1.11) is approximated by substituting (2.5.1.10) and (2.5.1.7) for the partial derivatives, yielding,

$$\left.\begin{array}{l}\frac{1}{k}\{u_{i,j+1}-u_{ij}\} = \frac{1}{h^2}\{u_{i+1,j}-2u_{ij}+u_{i-1,j}\} + T_{ij} \\ T_{ij} = \frac{h^2}{12}\left(\frac{\partial^4 u}{\partial x^4}\right)_{i+\theta_i h,j} - \frac{k}{2}\left(\frac{\partial^2 u}{\partial t^2}\right)_{i,j+\phi_j k} + \cdots\end{array}\right\} \qquad (2.5.1.12)$$

with $0<\theta_i<1$ and $-1<\phi_j<1$. T_{ij} is the *Local Truncation* error, neglecting this term gives an approximate value for u_{ij} at the point P_{ij} and after re-arrangement and with $r=k/h^2$, we have,

$$u_{i,j+1} = ru_{i+1,j}+(1-2r)u_{ij} + ru_{i-1,j}\ , \qquad (2.5.1.13)$$

known as the *classical explicit formula*, its structure is shown by the molecule on Fig.(2.1a) and it follows that given three points on the jth abcissa (the jth time level) then one point of the (j+1)th level can be computed. The initial conditions of (2.5.1.11) gives us all the points on t=0 hence we can compute all the points on t=1,2 etc. by repeated application of (2.5.1.13). Further suppose that there are n internal points along t=0 (i.e. not counting x=0, x=ℓ) then a complete time level can be formulated as n applications of (2.5.1.13). Numbering the points from left to right on the jth time level produces the n linear equations,

$$\begin{aligned}(1-2r)u_{1,j}+ru_{2,j} &= u_{1,j+1}-ru_{0,j}\\ ru_{1,j}+(1-2r)u_{2,j}+ru_{3,j} &= u_{2,j+1}\\ &\vdots\\ ru_{n-1,j}+(1-2r)u_{n,j} &= u_{n,j+1}-ru_{n+1,j}\end{aligned}$$

and as $u_{0,j}=u_{n+1,j}=0$ (boundary conditions) this yields a linear system of the form (2.4.31) given by,

$$u_i^{(j+1)} = Au_i^{(j)} + b \ , \qquad (2.5.1.14)$$

with,

$$A = \begin{bmatrix} 1-2r & r & & & \\ r & 1-2r & r & & \\ & \ddots & \ddots & \ddots & \\ & & r & 1-2r & \end{bmatrix}_{n\times n} \ , \quad u_i^{(j)} = \begin{bmatrix} u_{1j} \\ u_{2j} \\ \vdots \\ u_{nj} \end{bmatrix} \ , \quad u_i^{(j+1)} = \begin{bmatrix} u_{1,j+1} \\ u_{2,j+1} \\ \vdots \\ u_{n,j+1} \end{bmatrix}$$

$$b_i = \begin{bmatrix} -ru_{0,j} \\ 0 \\ \vdots \\ 0 \\ -ru_{n+1,j} \end{bmatrix}$$

From the structure properties SO-S13 and for large n,A is symmetric sparse banded and tridiagonal and is already in explicit form and solved simply by a matrix vector multiplication. Successive levels are constructed by repeatedly solving (2.5.1.14) replacing the old level by the new at each iteration. If (2.5.1.10) had been a backward difference, substitution into (2.5.1.11) would have produced the linear system,

$$Au_i^{(j+1)} = u_i^{(j)} + b \ , \qquad (2.5.1.15)$$

with,

$$A = \begin{bmatrix} 1+2r & -r & & & \\ -r & 1+2r & -r & & \\ & \ddots & \ddots & \ddots & \\ & & -r & 1+2r & -r \\ & & & -r & 1+2r \end{bmatrix} \qquad b = \begin{bmatrix} ru_{0,j} \\ 0 \\ \vdots \\ 0 \\ ru_{n+1,j} \end{bmatrix}$$

which is diagonally dominant, symmetric, sparse and tridiagonal, but yields an implicit system which requires direct or iterative solution techniques at each level. The formula corresponding to (2.5.1.12) is called the *classical implicit formula.*

2.5.2 Convergence, Stability and Consistency

Given a solution method, how can we be sure that the approximations at successive levels will stay close to the true solution of the problem.

Convergence: Suppose U is the exact solution of the PDE, and that u is the exact solution of the finite difference formula. If u approaches U along a time level or at a point as Δx and Δt tend to zero the method is convergent. This corresponds to introducing more and more points, and hence finer and finer grids, and at some stage the points will be so close together that the discretized solution will look very much like a continuous one. Consequently, the difference U-u is termed the *discretization* (or global *truncation*) error. The choice of grid size is critical to the success of the approximation, and can be analyzed using local truncation terms, we might also consider improving the result by estimating the error but this usually involves evaluating unknown derivatives leading to a more complicated process.

Stability: We actually solve the difference equations on a computer

with inherent rounding errors. The initial conditions themselves introduce additional errors associated with the gathered data from a physical process. Now if the finite difference formula was solved exactly (i.e. no rounding errors) and they limited the amplification of errors in all components of the initial conditions the formula would be stable. Successive time levels can consider the previous level as initial conditions and so limiting error growth also limits rounding error.

If we assume that $h\to 0$ and $k\to 0$ convergence and stability can be related using,

Theorem 2.5.1: (Lax's equivalence theorem)

Given a properly posed linear initial-value problem and a linear finite-difference approximation to it that is consistent, stability is a necessary and sufficient condition for convergence.

Now suppose that u_0 is the exact and $\bar{u}_0$ the estimated initial conditions given that $e_j=\bar{u}_j-u_j$ is the error on the jth time level, substitution into (2.5.1.14) yields $e_j=Ae_{j-1}$ which by repeated substitution produces $e_j=A^j e_0$ and using (2.4.3.2)-(2.4.3.5) indicates that convergence and stability of the methods occur if $||A||<1$. Likewise (2.5.1.15) is diagonally dominant and symmetric indicating that solution by direct or iterative methods at each level will also provide a stable hence convergent method.

This leaves only the problem of consistency. A finite-difference method may be stable but may converge to the solution of a different differential equation than the one intended as $k\to 0$ and $h\to 0$, such a method is said to be inconsistent. We assume throughout the work that equations are consistent and so matrix theory and convergence can

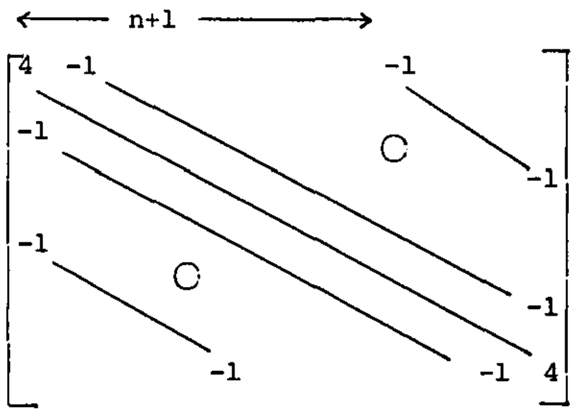
be applied to finite difference methods using Theorem(2.5.1).

Finally the technique described for solving the parabolic form (2.5.1.11) can be applied to other equations deriving other structured coefficient matrices and corresponding solution methods.

For the 2-D elliptic P.D.E., i.e. Laplace equation (2.5.4b) with Dirichlet boundary conditions $u(x,y)=0$ on C, corresponding to Fig.(2.1b) a five point formula,

$$4u_{i,j}-u_{i+1,j}-u_{i-1,j}-u_{i,j+1}-u_{i,j-1} = 0\ , \qquad (2.5.1.16)$$

is derived and assuming n^2 internal grid points with a columnwise ordering $(P_{ij}\rightarrow(j-1)n+i)$ of points and hence equations produces a coefficient matrix A of order n^2 of the form,

A =  , $b=0$, (2.5.1.17)

a symmetric sparse banded matrix which illustrates simple striped features. Similarly, the 3-D problem of the form,

$$\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}+\frac{\partial^2 u}{\partial z^2} = 0\ , \qquad (2.5.1.18)$$

$$(x,y,z) \in \mathbb{R} \equiv (0,1,)\times(0,1)\times(0,1)$$

and boundary conditions,

$$u(x,y,t) = 0$$

produces a cube dissected by a three-dimensional grid with spacing $\Delta x,\Delta y,\Delta z$ and when $\Delta x=\Delta y=\Delta z$ produces a seven-point finite difference formula,

$$6u_{i,j,k}-u_{i+1,j,k}-u_{i-1,j,k}-u_{i,j+1,k}-u_{i,j-1,k}-u_{i,j,k+1}-u_{i,j,k+1} = 0 \qquad (2.5.1.19)$$

specified at the n^3 grid-points in R with $u(i\Delta x, j\Delta y, k\Delta z)=u_{i,j,k}$ ordering these points produces the linear system,

$$A = \begin{bmatrix} A_1 & -I & & \\ -I & & \ddots & \\ & \ddots & & -I \\ & & -I & A_n \end{bmatrix}_{n^3\times n^3} \qquad (2.5.1.20)$$

with,

$$A_i = \begin{bmatrix} B_{j+1} & -J & & \\ -J & & \ddots & \\ & \ddots & & -J \\ & & -J & B_{j+n} \end{bmatrix}_{n^2\times n^2}, \quad B_j = \begin{bmatrix} 6 & -1 & & \\ -1 & & \ddots & \\ & \ddots & & -1 \\ & & -1 & 6 \end{bmatrix}_{n\times n}$$

$i=1(1)m$
$j=(i-1)(1)m.$

with I and J the $n^2\times n^2$ and $n\times n$ identity matrices respectively, and u,d $n^3\times 1$ vectors.

Clearly, when n is large these matrices produce large sparse banded matrices, where even the band is sparse in structure.

2.6 MISCELLANEOUS ITEMS

2.6.1 Convex Sets

Definition 2.6.1.1: A convex combination of points $a_1,a_2,\ldots,a_n \in \mathbb{R}^n$ is a linear combination $a=\alpha_1 a_1+\alpha_2 a_2+\ldots+\alpha_n a_n$ where $\alpha_i \in \mathbb{R}$ non-negative and satisfy $\sum_{i=1}^{n} \alpha_i=1$.

Definition 2.6.1.2: A vector space (or a set of points) A is convex if for all pairs of points $a_1, a_2 \in A$ and scalar $\alpha \in \mathbb{R}$ then any convex combination $a_3 = \alpha a_1 + (1-\alpha) a_2 \in A$, (i.e. closed under convex combination).

A point is called an *extreme point* of a convex set if it cannot be represented by more than one pair of points in A.

Definition 2.6.1.3: A convex polyhedron is the set of all convex combinations of a finite number of points. A *Simplex* is a convex polyhedron generated by n+1 points which do not lie in a plane formed from point vectors in $\mathbb{R}^n$, e.g. a Simplex in $\mathbb{R}^2$ is a triangle, in $\mathbb{R}^3$ it is a tetrahedron.

2.6.2 Rings and Fields

Let K be a non-empty set with two binary operators, say addition denoted + and multiplication by juxtaposition, K is a ring if it satisfies the axioms:

1. For any $a,b,c \in K$ $(a+b)+c=a+(b+c)$
2. There is an element $0 \in K$ called the zero element and $a+0=0+a=a$ for $a \in k$.
3. For each $a \in K$ there is an element called the negative of a, denoted $-a \in K$ such that $a+(-a)=(-a)+a=0$.
4. For any $a,b \in K$ $a+b=b+a$
5. For any $a,b \in K$ $(ab)c=a(bc)$
6. For any $a,b,c \in K$, $a(b+c)=ab+ac$ and $(b+c)a=ba+ca$

K is called a commutative ring if $a.b=b.a$ for all $a,b \in K$, and a ring with a unit element has $1 \in K$ such that $a.1=1.a=a$ for all $a \in K$. A field is a commutative ring with a unit element if every non-zero element has a multiplicative inverse $a^{-1} \in R$ such that $a.a^{-1}=a^{-1}.a=1$.

2.6.3 O-Notation

We use a technique involving asymptotic notation to compare two competing algorithms which concentrates on the basic number of operations or component cells in an array design. The notation we adopt is the O-notation.

Definition 2.6.3.1: $f(n)=O(g(n))$ is used to represent the relationship $|f(n)| \leq c|g(n)|$ for all $n \geq n_o$, where c and n_o are constants.

For operation counts (2.3.2.4) and (2.3.2.5) we can put,

$$f(n) = [n^3+3n^2-n] \text{ and } g(n) = [n^3+2n^2-n]$$

for mults/divs

hence $|f(n)| \leq c|g(n)| \quad c=\frac{3}{2} \quad n_o=2$

as $n\to\infty$ $f(n)\to n^3$ $g(n)\to n^3$ consequently $g(n)=O(f(n))$ and $f(n)=O(g(n))$ and the methods are asymptotically equivalent.

Futhermore $f(n)=O(n^3)$ and $g(n)=O(n^3)$ and we say the algorithms have $O(n^3)$ complexity. Backsubstitution is an $O(n^2)$ complexity problem from (2.3.1.5) and as $O(n^2)<O(n^3)$ solving an upper/lower triangular form is always computationally easier than a general matrix.

We can now substantiate the claim that converting a linear system to its explicit form (2.3.2) requires more computation than triangularisation. If A^{-1} exists implying A is a non-singular matrix, say of order $n\times n$,

$$AA^{-1} = I \ , \qquad (2.6.3.1)$$

taking each column of A^{-1} as an unknown vector with the corresponding column of I as the rightside produces n linear systems. Each system requires $O(n^3)$ operations and so forming A^{-1} and solving (2.3.2) has $O(n^4)$ complexity, it follows as $O(n^3)<O(n^4)$ that Gaussian Elimination is better. $O(n^4)$ is a very rough bound, by making use of the sparsity

in I and saving multipliers on the first solution, with subsequent solutions obtained by modifying rhs and backsubstitution operations, the time can be reduced to $O(n^3)$, this makes the two methods asymptotically equivalent but is rather misleading. In practice, when A^{-1} is unknown and A is a general matrix Gauss elimination or Factorisation is preferred, asymptotic analysis ignores the value of c which in this case is large enough to affect the choice of method. For parallel algorithms extra processors are incorporated to perform some operations simultaneously, and although the same notation is used the number of processors and value of c become important. In comparing parallel with sequential algorithms an order of magnitude drop in complexity is expected due to simultaneous operations, while comparison of two parallel algorithms with the same asymptotic number of processors indicates changes in c. Consequently improvements to existing parallel algorithms seem less dramatic.

CHAPTER 3

FOUNDATIONS OF SYSTOLIC ALGORITHMS

"You can observe a lot just by watching"

Yogi Berra.

"This is the awe-inspiring universe of magic:
There are no atoms, only waves and motions
all around..."

-The Atreides Manifesto
extract from "Heretics of
Dune", by Frank Herbert.

In this chapter the focus of attention shifts to systolic algorithms and their corresponding arrays. Section 3.1 defines basic concepts such as systolic spaces,processor geometries, wavefronts and types of systolic arrays. A basic set of axioms is given for systolic design which play a similar role to the algebraic structures in the previous chapter. Complementing these 'systolic' structures is a set of technology conscious heuristics which define practical design limits. Common networks are identified, and in Section 3.2, are used to illustrate systolic principles for the basic numerical methods from Chapter 2 forming a reference set for later discussions. The snapshot method for tracing systolic array operation is adopted for hand testing, before more theoretical techniques for manipulating systolic spaces and structures by means of re-timing and replacement are examined in Section 3.3. A mapping technique is used to translate abstract designs into OCCAM code providing automatic snapshot generation by program execution and implicit algorithm verification. Section 3.4 considers constraints imposed by VLSI technology validating the existing design heuristics and briefly examines area/time tradeoffs. Finally in Section 3.5 we propose new design heuristics in the light of recent innovations in 3-D VLSI design and optical computing and propose an alternative framework, the soft-systolic paradigm, used in later chapters.

3.1 SYSTOLIC SPACES AND STRUCTURES

At the most abstract level systolic computation demands only that moving data and instruction sequences interact to achieve some computation in parallel, while preserving a pumping action. This high level view motivates the following definitions which define hierarchical

levels of design complexity, illustrating that technological constraints imposed on systolic arrays limit systolic algorithms to a narrow design domain, and consequently small problem space.

Definition 3.1.1: A *Data (flow) sequence* (D_s) is a sequence of data elements all of the same type which has direction and speed.

The sequence can be represented by a triple $D_s=(\bar{a},b,\beta)$ where $\bar{a}=\langle a_1,a_2,\ldots,a_m\rangle$ is a sequence of length m, $b \in \mathbb{R}^n$ is an n-space direction vector and β is a speed (velocity) parameter.

Definition 3.1.2: An *Instruction (flow) sequence (I_s)* is a sequence of instruction procedures consisting of instructions from some finite set of instructions, which has direction and speed.

Instruction sequences are also represented by triples $I_s=(\bar{p},b,\beta)$ where $\bar{p}=\langle p_1,p_2,\ldots,p_k\rangle$ is a finite sequence of procedures and b, β have the same meanings as above. Individual procedures are considered similar to communicating sequential processes (Hoare [78]). For a systolic computation to occur data and instruction sequences must move in a common space.

Definition 3.1.3: A *systolic space* is a cartesian space of dimension m, where m is greater than or equal to the maximum dimension of direction vectors taken over all data and instruction sequences flowing in the space.

By the property that the systolic space is cartesian it can be discretized to produce points (locations, sites or co-ordinates) by using vectors with integer components. Because the space is at least as large as the space required by flow sequences, they can be mapped into it by assigning sequence elements to contiguous points in the direction of the direction vector. The velocity parameter preserves

the pumping action by defining a beat or pulse.

Definition 3.1.4: A systolic beat or pulse of a sequence (a,b,β) is the number of points it moves in unit time along direction b. If we assume a uniform notion of time throughout the space all the beats of sequences can be normalized to give a global or synchronous clocking mechanism. This mechanism can be used for timing systolic computations.

Now imagine we have a number of data sequences passing through the same space (S) then two possibilities are apparent, either:

(i) Some sequences collide, that is elements of the sequences occupy the same point in S at the same beat, or

(ii) No elements of sequences occupy the same point in S on the same beat.

In the first case it is clear that co-habiting elements may interact while in the second case no interaction is possible. Yielding

Definition 3.1.5: A *potential computation site* is produced at a point in a systolic space where elements from different data sequences occupy the same point at the same instance in time. Hence a collection of data sequences can be assigned a 'computational potential' according to the number of sites formed from the time they enter the space to the time when they leave it and this in some way measures the implicit parallelism of the dataflow. In order to fulfil this computational potential a catalyst is required and takes the form of instruction sequences. Instruction sequences are assigned to the systolic space just like data sequences with procedure elements occupying points. A single systolic beat is then divided into a systole phase for movement between points and diastole phase where the procedure in the node is executed. The cycle time is the total number of global beats required to complete the

most complex procedure in the space.

Definition 3.1.6: The union of a systolic space, data and instruction sequences which preserve a pumping action, with instruction sequences passing through potential computation sites is termed a *systolic algorithm.*

Some simple characteristics of systolic algorithms follow from this general definition. Firstly, the pumping action is only preserved if at least one (possibly an instruction) sequence has non-zero velocity. Second, elements of distinct instruction sequences cannot occupy the same point at the same time. Otherwise distinct procedures could interfere with each other's computations on the co-habiting data elements generating erroneous results. We shall call systolic algorithms possessing these qualities well-posed and otherwise ill-posed. The computation performed at points in systolic space by a well-posed algorithm are unique, with data modified according to the procedures residing at each point. The time of a systolic algorithm T can then be measured by the number of cycles from the first creation of a computational site to the last.

To simplify discussions about systolic algorithms it is useful to consider whole collections of sequences together motivating the definition below.

Definition 3.1.7: A data or Instruction *(flow) group* is a group of sequences which share a common direction and speed and also occupy non-overlapping regions of points in a systolic space.

Notice that once elements of a group are assigned points in space they remain stationary with respect to each other. The task of reasoning about groups is then simplified if their elements are regarded as wavefronts.

Definition 3.1.8: Let $d_i=(\bar{a}_i,b,\beta)$ i=1(1)k define k sequences forming a (k-ary) group. A *systolic wavefront* is defined by connecting points containing the elements with the same location in different sequences, i.e. $\bar{a}_i(j)$ j=1(1)m for sequences of length m. When j=1 the principal or leading wavefront is defined.

The stationary property of groups ensures that wavefronts move and act according to the Huygens principle whereby no two waves from the same source (group) can interfere or cross one another. Borrowing further ideas from Wave Theory in Physics provides a natural terminology for data and instruction flow in systolic algorithms. For instance, the concept of wave duality can be adopted for flow sequences. During the systole or flow phase of a beat data and instructions become indistinguishable, but in the diastole phase take on different properties with instructions capable of performing actions and data only capable of being acted upon. Systolic wave duality is strengthened if we consider a uniform (e.g. binary) representation for both data and instructions and further limit instruction procedure elements to single operations. We might also consider coupling different systolic designs by methods such as pipelining. For two algorithms s_1 and s_2 boundaries between the systolic spaces exist where changes in beat and direction of flow could occur. In a similar manner to the refractive index for the change in the speed of light and Snell's law for the angle of refraction in physics we might define a systolic index for coupled algorithms defining changes in wave speed and direction. Generally, however, it is a non-trivial task to speed up or slow down waves already flowing in a space and re-scaling the global beat mechanism over both algorithms is often wiser.

Systolic computation itself can be envisaged as the interaction of systolic wavefronts and leads naturally to the idea of *computational interference*. Individual elements of sequences arriving at the same point are portions of their respective wavefronts and must satisfy these restrictions:

(i) waves must be *coherent:* data sequences must be of the same type, which instruction sequences must be capable of manipulating.

(ii) speed of waves must be *consistent* that is the same speed or multiples or simple fractions of each other.

(iii) sequences at a point must be *compatible* i.e. data and instruction sequences cannot be combined to yield valid data or instructions.

This latter point is interesting because the duality of systolic waves means that sequence flow can fool a point into modifying an instruction sequence interpreted as a data sequence; by using a second instruction sequence to carry out modification. Thus systolic arrays which modify instructions are also well-posed.

Wavefronts satisfying the above properties lead to three types of computational interference

(i) Constructive Interference: Instruction and data elements combine to modify data creating true computations.

(ii) Neutral Interference: Data sequences are preserved and a null computation is achieved.

(iii) Destructive Interference: Instruction and data elements combine to produce erroneous results or fallacious computations.

The most difficult and exciting part of systolic algorithm design is the

arrangement of data and instruction sequences to form waves that constructively interfere to produce recognizable computations. Reasoning about systolic algorithms at this abstract level is further complicated by the limited processing power of the human brain. In order to picture any real examples we immediately place restrictions on the dimensionality of the systolic space as well as the number, direction and speed of sequences in the space.

Definition 3.1.9: A *processor geometry* is a directed graph whose nodes correspond to potential computation sites and whose arcs are defined by direction vectors of sequences; and which itself has direction and speed.

By allowing the geometry direction and speed, computation can be achieved by a smaller processor graph than the total number of potential computation sites providing that all the sites yielding constructive computations are visited during the course of the calculation.

Consequently an alternative definition of algorithm computation time can be given as the number of cycles required by the geometry to traverse the locus of constructive computation sites. As sequences and geometry move relative to each other it is usually possible to give the geometry zero speed and modify sequence flow to preserve computation. The task of adjusting data and instruction sequences is simplified if we allow variable direction sequences.

Definition 3.1.10: A variable direction (flow) sequence is a sequence whose direction vector can vary with time, e.g.

for $d_s=(\bar{a},b,\beta)$ from Definition(3.1.1), a variable direction equivalent is $d_s=(\bar{a},b_t,\beta)$ for t=1,2,3,... with b_t direction vectors in the systolic space.

Definition 3.1.11: A *systolic array* is a processor geometry with zero velocity covering all constructive computation sites together with a set of data and instruction sequences.

The processors of the geometry are assumed capable of performing any of the operations in the instruction sequence procedures. Consequently if an instruction group is homogeneous meaning that all the procedure elements are identical, it can be made stationary such that elements and processes coincide producing a dedicated systolic array. Furthermore, if instruction sequences are composed only of a few types of simple operations the complexity of processors is significantly reduced producing simple cells. As a result it is often the case that the terms systolic algorithm and systolic array are used interchangable in the context of array diagrams and operation, this arises from the need to visually represent algorithms in order to understand their dataflow.

Combining all the above features permits the following definition of a *systolic frame* over a systolic space:

R[1]: There must be an underlying structure of processors with connections

R[2]: Data and instructions must flow through the processors like waves (pumping action preserved).

R[3]: Processors perform only simple operations.

R[4]: Flow of data and instruction should be simple and regular

R[5]: Connections are nearest neighbour

R[6]: The processor geometry consists of only a few types of simple cells.

We can refer to systolic frames satisfying R[1-3] as *irregular frames* and those also including R[4-6] as *regular frames*. A systolic *semi-frame* can then be defined as a regular frame in which R[5] is relaxed to allow almost or next nearest-neighbour and limited fanout connections. Furthermore, we can say that a systolic frame (F) is a frame with a

neutral element preserving the real data operands - i.e. neutral interference.

Defining systolic arrays to have zero-speed has practical merit because they can then be mapped onto physical computing structures. Assuming that processors provide methods of performing operations implies some physical structure, and consequently, the geometry consumes area or volume. By using VLSI technology as a means to restrict systolic algorithms/arrays to semi-conductor or chip surfaces further heuristics can be defined to facilitate easier implementation.

H[1]: The systolic space hence frame is restricted to 2-D

H[2]: The processor geometry is planar or almost planar

H[3]: The number of input and output sequences to a point is limited (this ensures fixed sized processors)

H[4]: Broadcasting to a number of processors simultaneously is avoided.

H[5]: Longwires are undesirable as for wires over a certain length propagation delays can become significant, causing mistiming.

Systolic frames also possessing these properties will be termed *constrained frames*. Notice that usually designs in regular frames contain designs in constrained frames as a subset, it follows that it may not be feasible to implement designs from an unconstrained frame in VLSI technology.

We may attempt to map designs in a regular frame with a high dimensional systolic space into designs for a space of lower dimension to create a constrained frame for implementation. There is no guarantee however, that the resulting designs fit a regular or constrained frame of the smaller space.

The definition of a systolic array implies that designs in a systolic frame can be further classified according to their sequence movement and processor geometry.

(i) Sequence Flow Patterns:

The number, speed, direction and structure of sequences and groups of sequences can all be used to characterize a systolic design. When instruction sequences are homogeneous and have zero speed emphasis is placed on systolic dataflow, and designs are divided into stationary and non-stationary arrays. An array is *stationary* if a selection of data sequences has zero speed, if no sequences have zero speed the array is *non-stationary*. Normally a result sequence or group is present in a design whose purpose is to collect partial and complete results as it moves through the array. These result sequences are often the ones made stationary, in principle however, any set of sequences can be made stationary, but can result in larger geometries. These two categories can be further sub-divided according to the direction of flow. For instance, there are uni-directional arrays with data flow in only one direction, and bi-directional (two-way) flow in two directions, and in general k-directional with flows in k-directions. Structure of groups and sequences can be assessed according to the position of neutral elements (if any) and the sequence elements. Many problems are given in terms of matrix computations with groups representing matrices and sequence vectors, in these cases the complexity of the function for producing subscripts for successive sequence elements is a useful measure of flow complexity.

(ii) Processor Geometries:

A static processor geometry implicitly defines the number of data

sequences and their direction by the inputs/outputs on the boundary of the geometry and interconnecting arcs. If we assume a regular constrained systolic frame three types of array topology can be used. They are:

(i) Two dimensional (2-D) geometries:- a selection of commonly used forms is shown in Fig(3.1).

(ii) Collapsed (or degenerate) 2-D geometries:- These are obtained from full 2-D forms like Fig(3.1b) and c by collapsing the array onto only a single row and column.

(iii) Linear (1-D) geometries:- obtained from collapsed geometries by restricting input/output to the left most and right most cells only.

Fig(3.1c) indicates an array using boundary cells, these cells are added to an otherwise homogeneous network to perform on different and often more complex tasks than the other processors. Clearly a two cell stationary geometry requires two stationary instruction groups. A final criteria for assessing geometries is the ratio of computation to communication (input/output). Suppose the design in Fig(3.1b) has n cells, there are $\sqrt{n}$ connections on each of the four boundaries. Hence on a particular cycle there can be at most $4\sqrt{n}$ inputs/outputs and n computations giving a ratio of $\frac{1}{4}\sqrt{n}$ or $O(\sqrt{n})$. For a collapsed form of Fig.(3.1b) there are $\sqrt{n}$ computations and $O(\sqrt{n})$ communications giving a ratio O(1), while for a linear array with n cells only a constant number of communications occur on the boundary giving a ratio of O(n). Notice that these geometries are produced by a 2-D systolic space and satisfy properties of a regular constrained systolic frame and so are considered amenable to VLSI implementation.

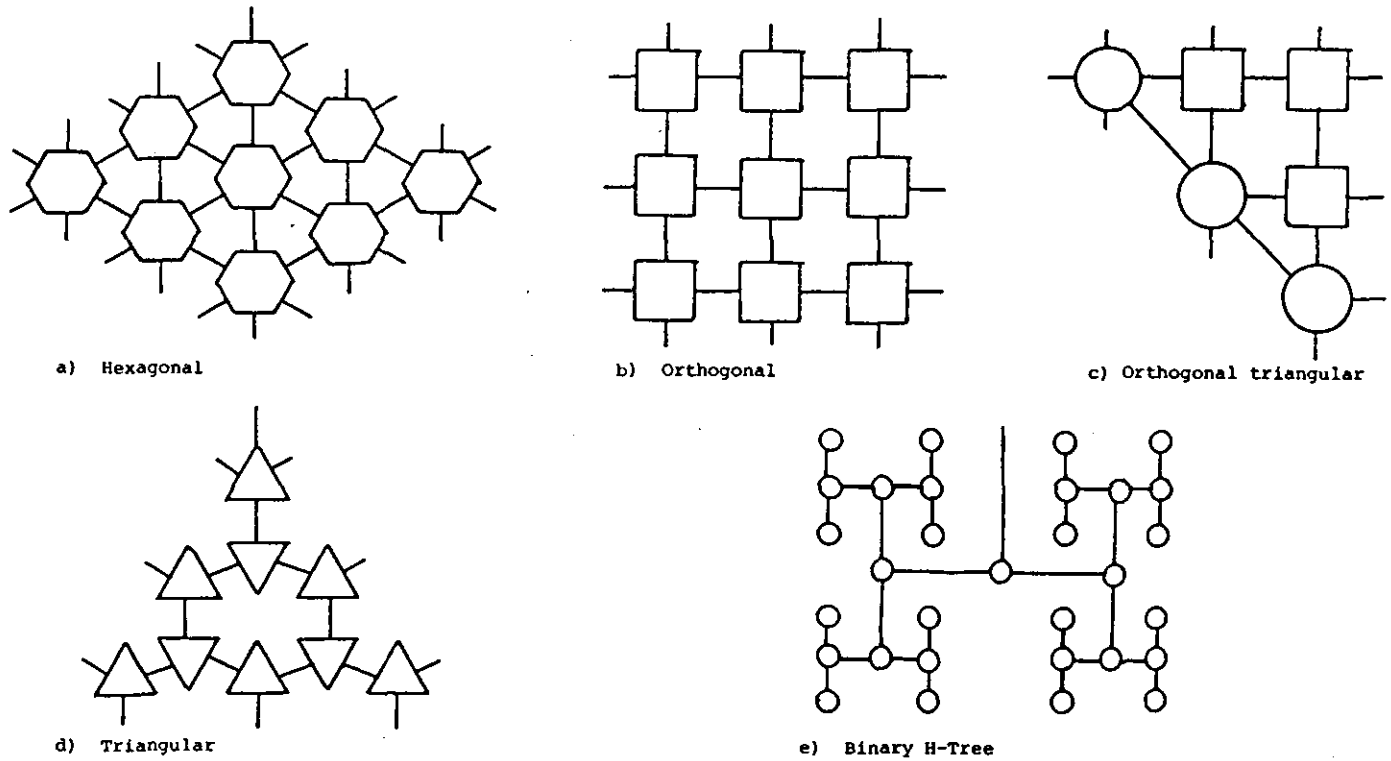


FIGURE 3.1: Selection of common 2-D processor geometries

Finally, the contribution of this section can be enumerated as follows:-

1. A simple terminology is defined for describing and relating different systolic designs.
2. Systolic algorithms are described as abstract objects which sit on processor geometries to create systolic arrays.
3. Constructive interference of wavefronts is defined as a necessary condition for recognizable computation.
4. The systolic structure of frames is introduced to define designs satisfying certain properties and constrained frames for designs sensitive to implementation problems.

We will see in the next section that a sufficient condition for useful computations is a mixture of neutral and constructive interference, where designs have a neutral element (zero).

3.2 STANDARD (OR TRADITIONAL) ARRAYS

Now consider some real systolic algorithms/arrays derived from constrained regular systolic frames over a 2-D systolic space. The designs are well publicised and can be found in a number of references such as Leiserson [81] and Mead & Conway [79] with the latter also providing basic VLSI knowledge. This reference set illustrates how the traditional numerical methods for solving linear systems (in Chapter 2) can be implemented as systolic arrays. Designs from the set will be referred to as traditional arrays and accompanying theorems on computation time and cell count can be used as a benchmark for new designs. In addition, the traditional arrays form a fine grain set of components which allow a 'bag-of' approach to more complex problems,

whereby a computational task is broken down into smaller tasks which are solved by selecting the most suitable array for each subtask, from a bag of standard arrays.

The fundamental unit of computation for these designs is the inner product step (y=y+a*x, with y,a,x $\in$ $\mathbb{R}$ scalars) the internal structure of the basic cell (processor) for different geometries are shown in Fig. (3.2) below.

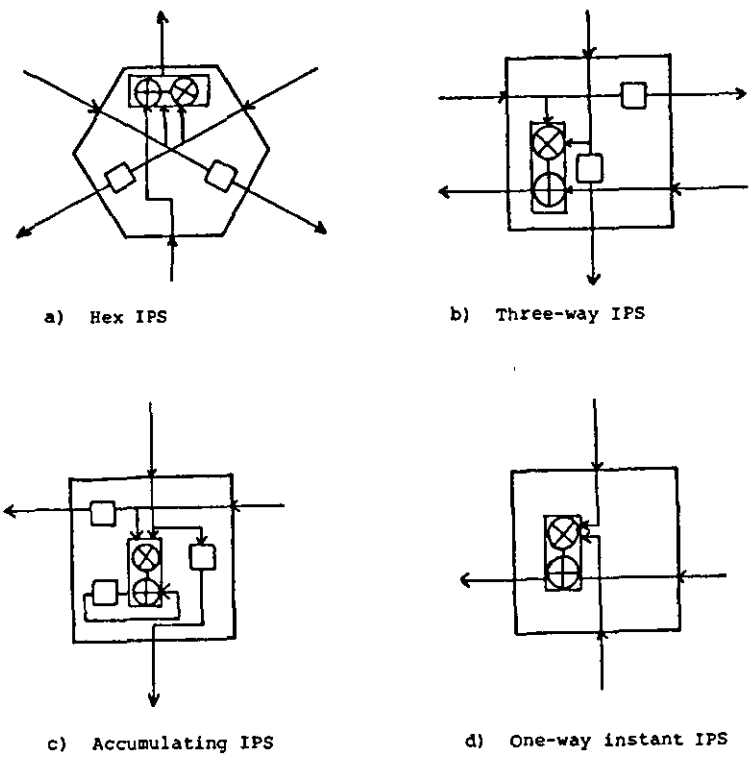


FIGURE 3.2: IPS geometries

The small empty boxes indicate latches preventing overwriting of data values before their replacements are valid. Circles indicate operations and where necessary are latched internally. Fig. (3.2a) and (3.2b) occur most frequently and define unit area and unit (cycle) time (i.e. the area and time required by one multiplication and one addition) for all designs.

As new cell designs are introduced their complexity will be graded according to the number of IPS equivalents required to implement them. This means counting additions and multiplication circuits and denoting cycle time by the proportion of IPS cycles required for cell operation. As the arithmetic portions dominate the cell area latches are omitted from calculations and for simplicity multipliers and dividers, adders and subtractors are considered to have equivalent area.

In general, the definition of a new cell can be achieved in two ways:

(i) A shape with labelled inputs and outputs is given together with a procedure defining its internal computation.

(ii) A sketch of the internal structure is shown.

(ii) is used for relatively simple cells with very little control but is replaced by (i) as the complexity of the cell increases. Sometimes (i) and (ii) are adopted simultaneously to illustrate special features of cells which can be used to reduce overall area or time, using techniques like the pipelining of internal operations.

3.2.1 Matrix and Vector Multiplication

We start with some very simple designs which illustrate stationary and non-stationary arrays, wavefronts, and why an IPS cell is adopted as the basic unit of computation for matrix orientated calculations.

Consider the inner product of two n-component vectors defined by (2.1.6) we require three data flow sequences one for each vector and a third for results, hence a design stationary with respect to results has flow sequences,

$$d_s^{(1)} = (\langle a_1, a_2, \ldots, a_n \rangle, (0,-1), \beta)$$

$$\left.\begin{aligned} d_s^{(2)} &= (\langle b_1, b_2, \ldots, b_n \rangle, (-1,0), \beta) \\ \text{and,} \qquad d_s^{(3)} &= (\langle \alpha \rangle, (0,0), 0) \end{aligned}\right\} \qquad (3.2.1.1)$$

direction vectors are from 2-D due to the assumed constrained frame. $d_s^{(3)}$ is the stationary result sequence and its direction is irrelevant, while $d_s^{(1)}$ and $d_s^{(2)}$ are orthogonal and hence must collide with each other, creating a single potential computation site. Embedding a single accumulating IPS processing element into this site defines a dedicated stationary processor geometry as the corresponding instruction sequence is homogeneous. The systolic array is given in Fig.(3.2.1.1) and is essentially sequential.

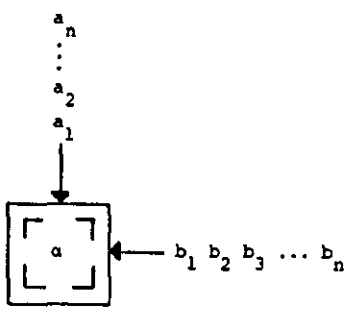


FIGURE 3.2.1.1: Stationary inner product

The 2-D systolic space is defined by the plane of the paper and moving sequences have the same velocity β equivalent to a single IPS cycle. Observing the array over a number of cycles proves the following theorem.

Theorem 3.2.1.1: The inner product of two n×1 vectors, $a,b \in \mathbb{R}^n$ can be computed using a single accumulating inner product cell in T=n IPS cycles.

The result is not very exciting, we hardly need an elaborate framework to extract this type of behaviour from (2.1.6), more

interesting properties are observed by constructing a non-stationary version. For the non-stationary case redefine vectors a and b to be groups and make result α a non-stationary sequence as follows, let,

$$\left.\begin{aligned} \bar{a}_i &= \langle a_i \rangle, \quad b = (0,-1),\beta \\ \bar{b}_i &= \langle b_i \rangle, \quad b = (0,1),\beta \end{aligned}\right\} \; i=1(1)n \qquad (3.2.1.2)$$

be the component sequence of the groups common direction, and speed of the two groups with the result sequence,

$$r = (\langle\alpha\rangle,(-1,0),\beta) \qquad (3.2.1.3)$$

where β=1 is the IPS cycle time. This design is shown in Fig.(3.2.1.2a) and consists of n one-way instant IPS cells, connecting all the components of vectors a or b defines the principle wavefronts for the two groups. Notice how sequences in the group are delayed in time to synchronise with the result sequence moving right to left accumulating a single term of the result at each cell. The dashes signify don't care or neutral elements, after the principal wavefront computational interference can be neutral or destructive and due to Huygens principle will not affect the result. Thus the following theorem is valid.

Theorem 3.2.1.2: The inner product of two vectors a and b can be computed by n one-way instant IPS cells in T=n cycles. Now comparing the two schemes we immediately notice that computation time is the same but the non-stationary scheme requires an additional n-1 cells. Furthermore, the stationary case requires only 3 boundary inputs/ outputs (allowing one for the result output) while the non-stationary version uses 2(n+1). Finally, the stationary array computes a constructive calculation every cycle, but in the non-stationary case each cell produces only one constructive computation.

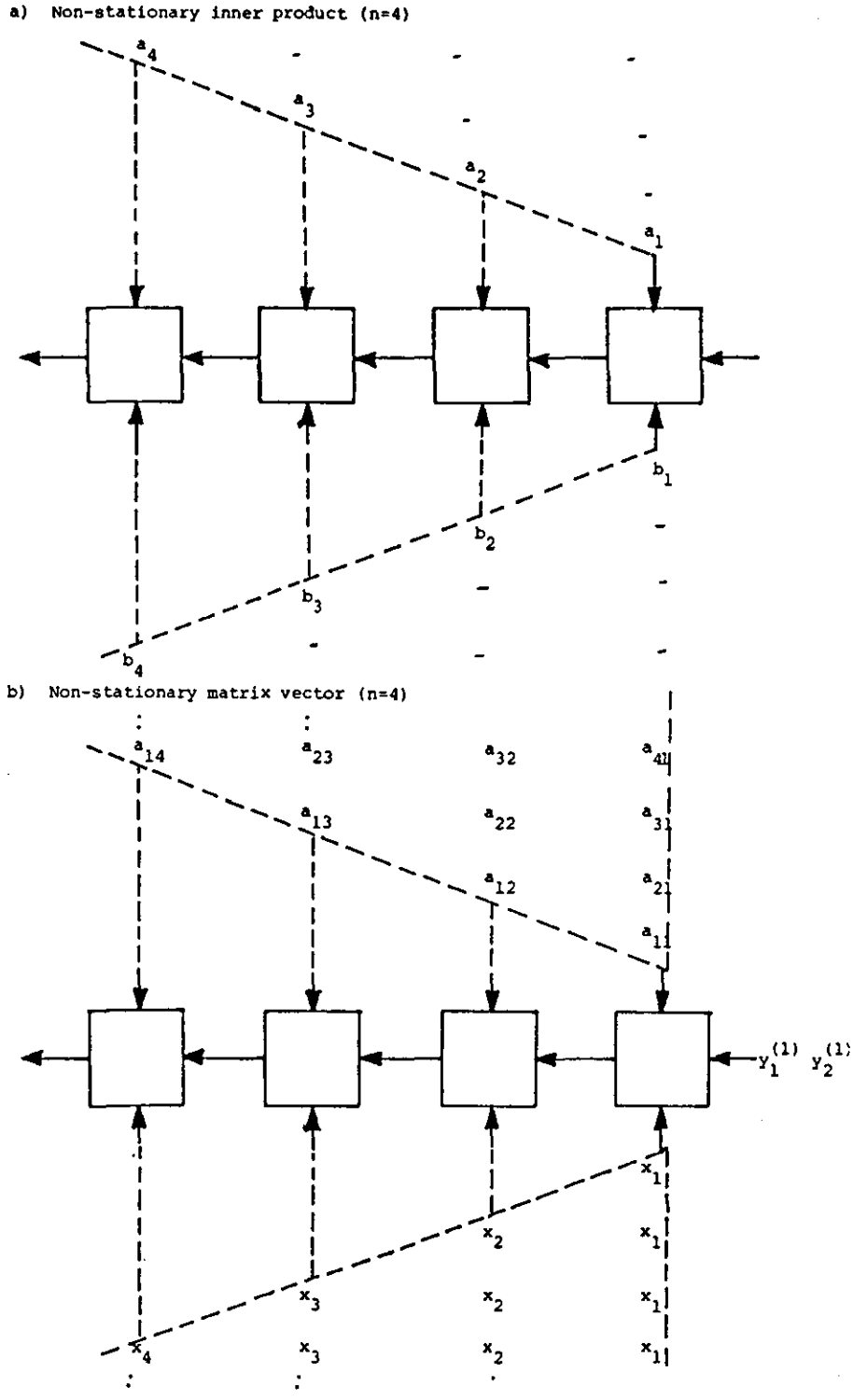


FIGURE 3.2.1.2: Matrix vector and inner product arrays

Definition 3.2.1.1: The number of constructive computations a typical or representative cell of an array performs during the course of an algorithm is termed the efficiency denoted e.

Normally $0<e\leq 1$ holds and for the above designs the stationary case has e=1 (the ideal value) while the non-stationary case is e=1/n indicating that its efficiency decreases as the problem size increases. It appears that the one cell design is best, and is not surprising due to the inherent sequential nature of equation (2.1.6).

Now consider the matrix vector multiplication problem Ax=y from (2.2.7) when B and C are reduced to n×1 vectors x,y respectively. Each component of y is produced by computing the inner product of a row from A and the vector x. More formally in recurrence notation,

$$\left.\begin{aligned} y_i^{(1)} &= 0 \\ y_i^{(k+1)} &= y_i^{(k)} + a_{ik}x_k, \quad i=1(1)n \\ y_i &= y_i^{(n+1)} \quad . \end{aligned}\right\} \qquad (3.2.1.4)$$

For a non-stationary solution, groups for A and x are formulated with component sequences,

$$\left.\begin{aligned} \overline{a}_i &= (\langle a_{1i},\ldots,a_{ni}\rangle,(0,-1),1) \qquad (3.2.1.5a) \\ \overline{x}_i &= \langle x_1,\ldots,x_n\rangle,(0,1),1) \qquad (3.2.1.5b) \end{aligned}\right\} \quad i=1(1)n$$

and the result sequence is,

$$\overline{c} = (\langle y_1^{(1)},y_2^{(1)},\ldots,y_n^{(1)}\rangle,(-1,0),1) \quad . \qquad (3.2.1.5c)$$

The systolic array is given by Fig.(3.2.1.2b) for n=4. As the $y_i^{(1)}$ values shift left each one collects a term for its inner product, and the computation of the components of y are pipelined. Tracing successive cycles of the array operation yields the following result.

Theorem 3.2.1.3: The matrix vector problem Ax=y for an n×n matrix A and n×1 vectors x and y can be computed in T=2n IPS cycles using n one-way instant IPS cells.

Proof:

Using Fig(3.2.1.2b), after n cycles $y_1^{(1)}$ has collected all its terms and is about to leave the array at the same time $y_n^{(1)}$ sits in the rightmost cell. Hence an additional n cycles are required for all the results to leave the array, giving T=2n.

The efficiency of the array for the new algorithm is e=½ as each cell performs a total of n constructive computations, a great improvement over the single inner product computation as e is now independent of n.

Corollary 3.2.1.1: An array of n one-way instant IPS can outperform a single accumulating IPS for multiple inner products even though it is twice as efficient.

Proof:

(i) The matrix vector problem is a sequence of n independent inner products requiring T=2n.

(ii) From Thm.(3.2.1.1) a single inner product on a one cell array requires T=n, as the cell has e=1 the n problems must be computed sequentially giving $T=n^2$ in total.

This Corollary illustrates the power of systolic arrays for utilising pipeline and parallel computations, the difference in cell count is out-weighed by the improvement in computation time.

A number of variations on the structure of Fig.(3.2.1.2b) which preserve the timing but modify the dataflow exist, two possibilities are:

(i) Each of the $\overline{x}_i$, i=1(1)n sequences are homogeneous thus a stationary design w.r.t. x can be constructed with x_i's preloaded into cells (the typed cell of Fig.(3.2) is augmented with a loadable register.)

(ii) The result sequence $\overline{c}$ is made stationary, and a non-stationary sequence $\overline{x}=(<x_1x_2\ldots x_n>,(-1,0),1)$ for vector x instead of a group is defined and the group for A is modified to give,

$$\overline{a}_i = (<a_{i1}\ldots a_{in}>,(0,-1),1) \ .$$

Each cell is now an accumulating IPS, with $\overline{x}$ using the former input for $\overline{c}$. Both these arrays have the same timing and efficiency as Thm.(3.2.1.3) but reduce the number of boundary input/outputs. The original matrix vector scheme demanded that the x vector components were repeatedly pumped into the array, the latter schemes require that they are input only once and from this viewpoint are superior.

REMARK: It is generally considered good practice to avoid repetition of inputs wherever possible.

Returning to the single inner product 1 cell design for a moment observe that not only does the cell have efficiency e=1 but that the array size is also independent of problem size n. On the other hand, the matrix vector (pipelined inner product) array is dependent on the problem size, changing the order of the matrix alters the size of the array. Fortunately, a problem size independent array can be derived by considering A to be banded.

The new banded array is constructed by considering another ordering of the coefficients in A as a flow group. The first design (Fig.3.2.1.2b) allocated coefficients to sequences in a column order

forming a row ordered wavefront pattern, and the stationary result scheme would require a sequence row ordering with column ordered wavefronts. A final possibility is to allocate coefficients in diagonal order such that each flow sequence contains elements from the same sub(super) diagonal which are separated by neutral elements. Using this diagonal group format allows the modification of the array stationary w.r.t. to x to make a non-stationary array with $\overline{x}=(<x_1x_2,\ldots,x_n(1,0),1)$ a moving sequence for x. The array is pictured in Fig.(3.2.1.3), its operation is slightly more complex incorporating a three-way IPS cell giving two-way flow of x and y.

The first 6 cycles of array operation are shown in Fig.(3.2.1.4) and a full trace gives this theorem.

Theorem 3.2.1.4: The matrix vector problem Ax=y for an n×n bandmatrix A with bandwidth w=p+q-1 and n×1 vectors x and y requires T=2n+w IPS cycles and w IPS cells.

Proof:

(i) From structure property S5 (Chapter 2) diagonals outside the band are all zero, hence cells with these inputs can be removed as they perform neutral computations leaving only w cells for necessary constructive computations.

(ii) The longest sequence in the flow group has length 2n, all these elements must be input giving lower bound T=2n.

(iii) The results of y_i are accumulated right to left giving an additional delay of w cycles for the first result to emerge hence T=2n+w.

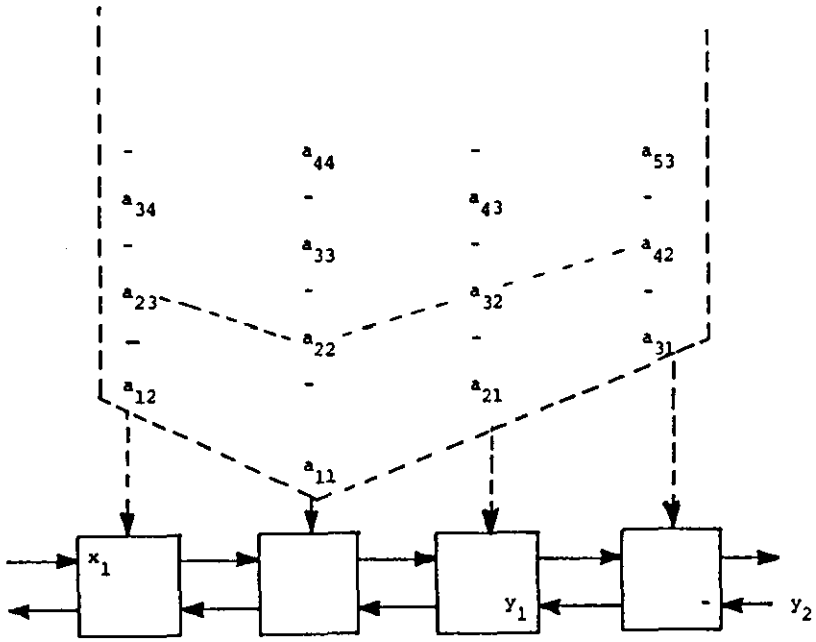


FIGURE 3.2.1.3: Banded matrix vector array (w=4, p=2, q=3)

In this design wavefronts are defined by a row ordering on the left of the main diagonal sequence and column ordering on the right. By definition neutral elements preserve operands and results and their use is extended to act as synchronising delay elements to permit the non-stationary sequence for x. The efficiency of the array is e=½ but the distribution of constructive computations is much more balanced than in Fig.(3.2.1.2b). In the latter scheme cells compute in a contiguous block of n constructive calculations and the portion of active or constructive cells moves left with time, in the new scheme active cells spread out from the main diagonal and alternate between constructive and neutral computations. Hence by nature of its problem size independence the band scheme appears superior - the Corollary below indicates a significant restriction.

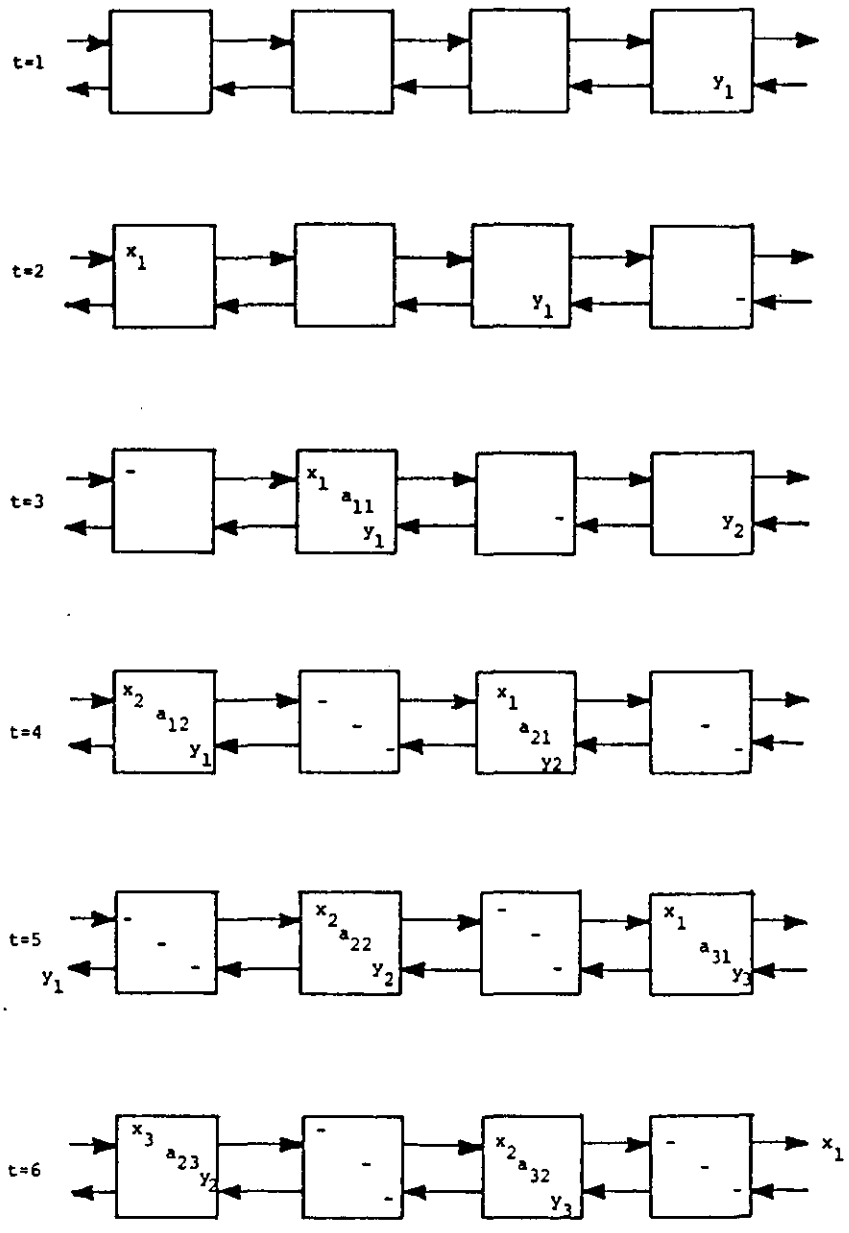


FIGURE 3.2.1.4: Snapshots of matrix vector computation

Corollary 3.2.1.2: A band matrix vector array independent of problem size is superior to an equivalent problem dependent array only when $w \ll n$, where w is the bandwidth of A an $n \times n$ matrix.

Proof:

(i) When $w \ll n$ e.g. tri- or quin-diagonal w is constant the additional cycles in T for the banded case are negligible for large n, making cell count the main consideration. Thus

the banded array is superior.

(ii) When A is full w=2n-1 and banded scheme requires T=4n-1 and 2n-1 cells compared with T=2n and n cells for the problem dependent case.

The result w<<n follows immediately.

The band matrix vector scheme can be improved by taking into account additional properties of the matrix coefficients. For instance in Toeplitz matrices of FIR filtering and symmetric matrices from discrete Fourier transform (DFT) matrices with constant diagonals (i.e. the same value in each location) or simple powers of each other are produced. The flow group associated with the matrix can then be made stationary w.r.t. A by preloading the constant values, and leads to the well studied convolution arrays of H.T. Kung [82b] .

In the above cases we adopted a hand testing method for tracing the array operation, essentially the image of the array is drawn on each cycle giving snapshots of the dataflow over time. This snapshot method is adopted throughout the thesis to support timing and dataflow arguments. So far, however snapshot traces have been simple but considering a more complex problem like matrix multiplication is more difficult.

From (2.2.7) the matrix product of two n×n matrices A and B can be formulated as recurrences,

$$\left.\begin{aligned} c_{ij}^{(1)} &= 0 \\ c_{ij}^{(k+1)} &= c_{ij}^{(k)} + a_{ik}b_{kj} \\ c_{ij} &= c_{ij}^{(n+1)} \end{aligned}\right\} \qquad (3.2.1.6)$$

for i=1(1)n, j=1(1)n. This formula can be manipulated in a similar

way as the simple matrix vector scheme to derive 2-D orthogonal and hexagonal arrays using geometries a) and b) in Fig.(3.1) and are illustrated in Fig.(3.2.1.5) and Fig.(3.2.1.6) respectively. A hint to deriving these arrays is to notice that a matrix product is a sequence of n independent matrix vector computations using A and the columns of B as the x vectors. Using the matrix vector array stationary

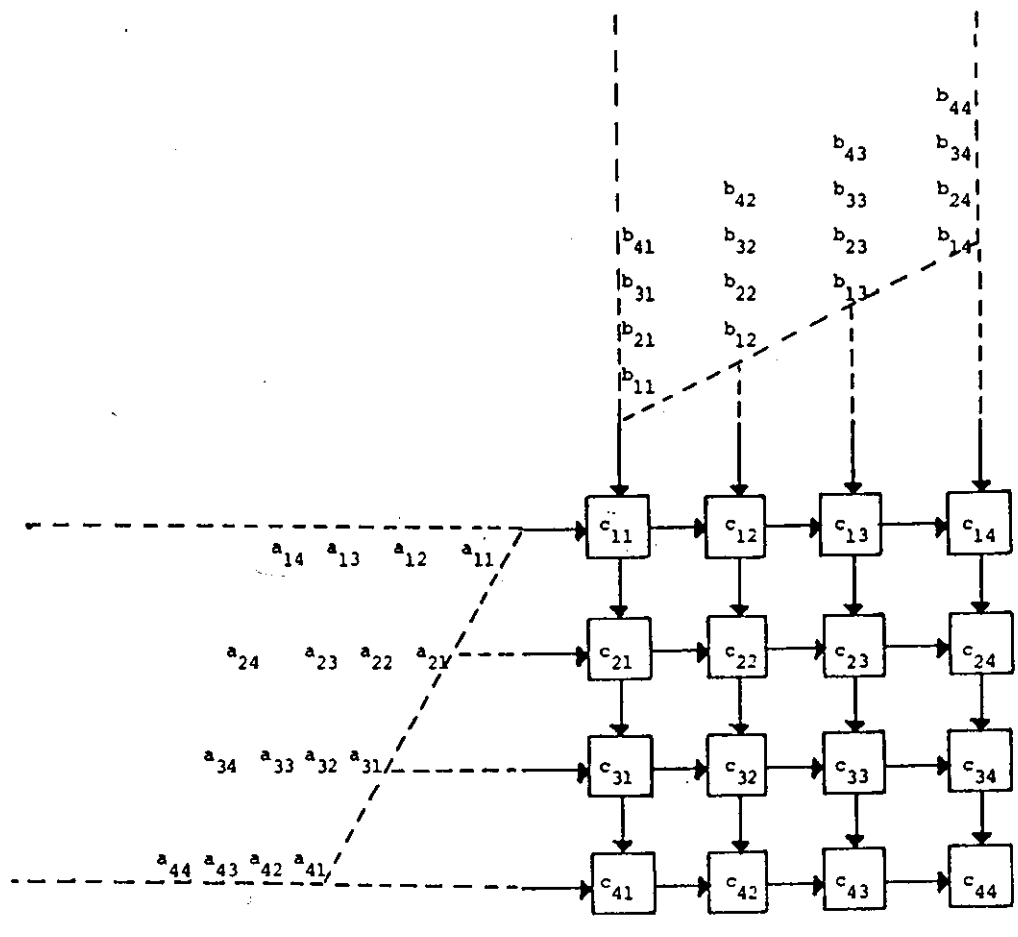


FIGURE 3.2.1.5: Stationary matrix product array (n=4)

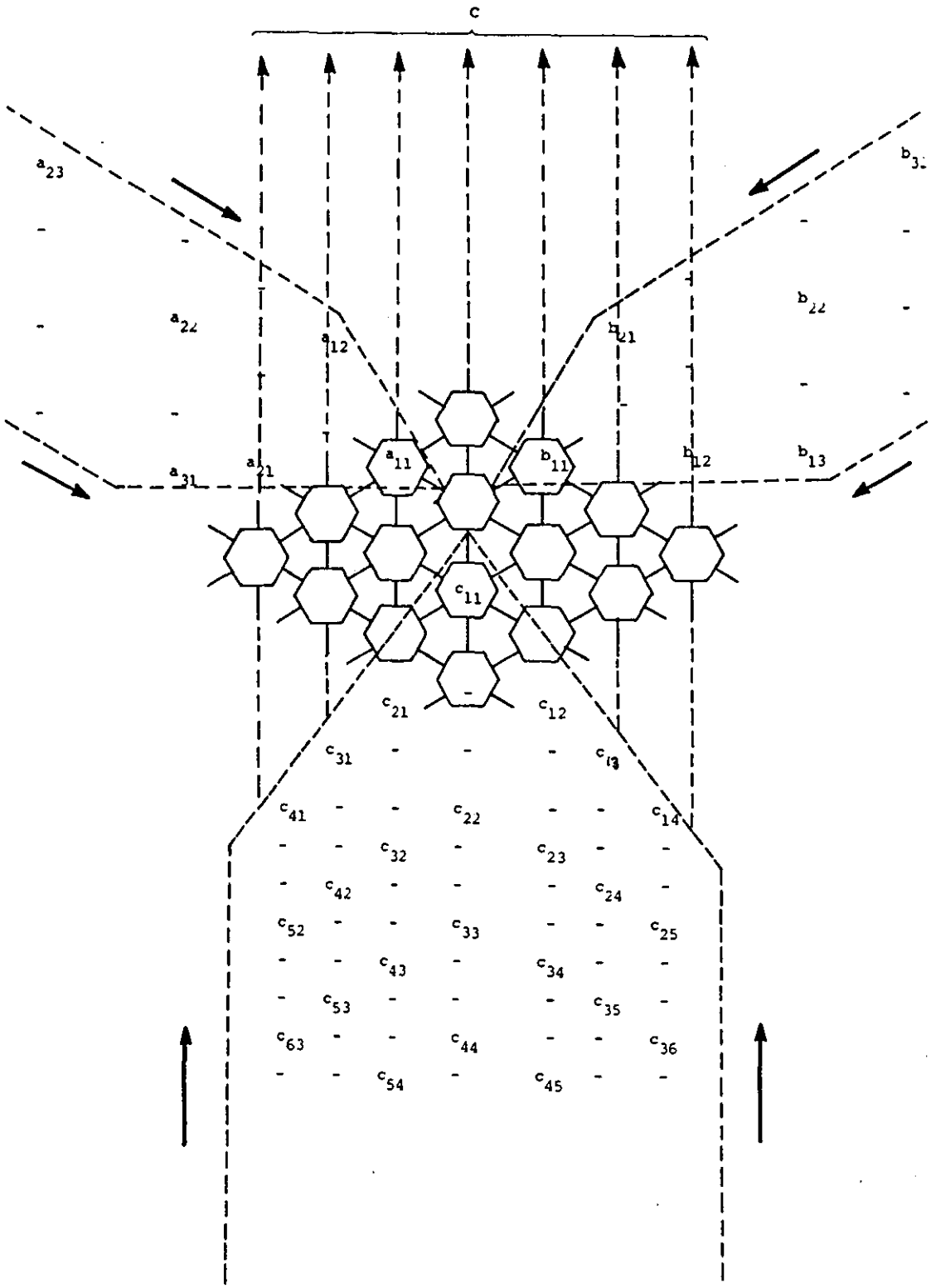


FIGURE 3.2.1.6: Band matrix multiplication ($w_1=w_2=4$)

w.r.t. results replicated n times will lead to the orthogonal array which is problem size dependent. The hexagonal array is problem size independent and based on the banded array. By tracing snapshots the following theorems corresponding to those for matrix vector arrays can be derived.

Theorem 3.2.1.5: The product of two dense n×n matrices A and B can be computed on an orthogonally connected mesh of n^2 cells in T=3n IPS cycles.

Proof: (from Fig.(3.2.1.5) data flow)

(i) After n cycles the result c_{11} is completed.

(ii) After 2n cycles all the results on and above the anti-diagonal are complete and all inputs have been read.

(iii) After 3n cycles the last result c_{nn} is completed and the product resides in the mesh.

Note: An extra n cycles will be required to read out the result but this is omitted.

Theorem 3.2.1.6: The product of two n×n band matrices A and B with bandwidths w_1 and w_2 respectively can be computed on a hexagonally connected network of w_1w_2 hex IPS cells in $T=3n+\min(w_1,w_2)$ cycles including input output time.

Proof: (From Fig.(3.2.1.6))

(i) The length of the longest input sequence is 3n giving a lower bound of 3n, as all elements must be input and output.

(ii) The c_{ij} results move north through the array and the first result c_{11} is delayed by $\min(w_1,w_2)$ cycles.

Corollary 3.2.1.3: A hexagonal band matrix product array is superior to a dense mesh connected array only when w_1 or $w_2 \ll n$.

Proof:

(i) when $w_1, w_2 \ll n$ the computational overhead $\min(w_1, w_2)$ is negligible and we save n cycles because no output time is required in the hexagonal case and require only $w_1 w_2$ cells.

(ii) when one matrix is full $w_1 = 2n-1$ or $w_2 = 2n-1$ hence time is unaffected due to minimum condition and requires $w(2n-1)$ hex cells compared with n^2 orthogonal cells (w is the bandwidth of the sparser matrix).

(iii) when $w_1 = w_2 = 2n-1$ both A and B are full $T = 5n-1$ and we use $(2n-1)^2$ hex cells making the orthogonal scheme superior.

Finally notice that both the orthogonal and hex scheme have e=1/3 but that the distribution of constructive computations is more balanced in the hexagonal array, as each cell works once every 3 cycles. In the orthogonal case cells perform constructive computations in a block of n cycles spreading out as a wavefront from the top left to bottom right corner. In fact for dense matrix multiplication the orthogonal scheme is the most efficient scheme known.

3.2.2 Arrays for Direct Solution of Linear Systems

Now consider systolic arrays corresponding to the equations (2.3.1.4) (2.3.2.1) and (2.3.3.3) which are back substitution, matrix triangularisation and LU-factorisation procedures respectively.

For backsubstitution (2.3.1.4) can be reformulated as the recurrence,

$$\left.\begin{aligned} y_i^{(1)} &= 0 \\ y_i^{(k+1)} &= y_i^{(k)} + a_{ik}x_k \quad , \; i=1(1)n \\ x_i &= (d_i - y_i^{(i)})/a_{ii} \end{aligned}\right\} \qquad (3.2.2.1)$$

using $y_i^{(k)}$ as an extra sequence collecting partial results. The array is shown in Fig.(3.2.2.1) below for a lower triangular matrix with bandwith w=q

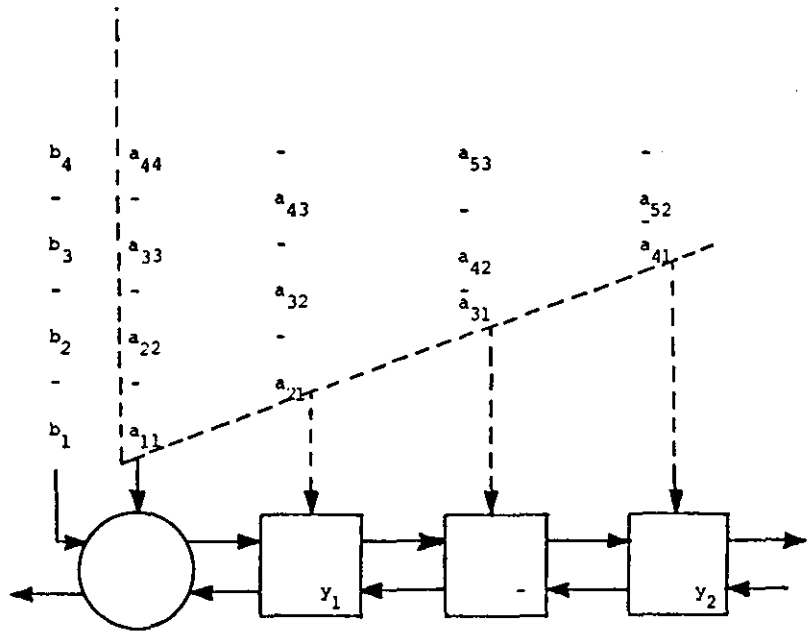
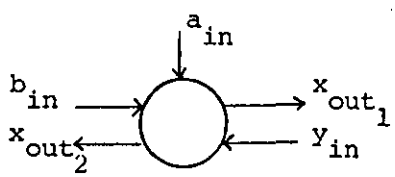


FIGURE 3.2.2.1: Backsubstitution (q=4)

and consists of the three-way IPS and a boundary cell on the left with the following cell definition,

Shape



Procedure

$$\left.\begin{aligned} x_{out_1} &= (b_{in}-y_{in})/a_{in} \\ x_{out_2} &= x_{out_1} \end{aligned}\right\} \qquad (3.2.2.2)$$

Notice that x_{out_1} relies on y_{in} and forms a feedback loop.
It follows that y_{in} values can arrive only once every two cycles, otherwise it would be impossible for them to collect all their required terms, and explains the positioning of synchronising neutral elements.

Theorem 3.2.2.1: A n×n band triangular system A with bandwidth w=q can be solved in T=2n+q IPS cycles using q IPS cell equivalents.

Proof: [Tracing dataflow of Fig(3.2.2.1)]

(i) The length of the longest data sequence is 2n requiring 2n IPS cycles to output results.

(ii) Each y_i moves left picking up one term per cell and on reaching the boundary cell has accumulated the expression

$$a_{i1}x_1+a_{i2}x_2+\ldots+a_{ii-1}x_{i-1}$$

which has at most q-1 terms. An additional delay occurs in the boundary cell producing x_i giving a total delay of q cycles for the first output.

In this proof we have invoked the assumption that subtract and divide has the same cost as an inner product cell in area and time. Modifications to the design are more difficult due to the feedback loop and the relations between successive x_i elements. A minor improvement is to write (3.2.2.1) as,

$$\left.\begin{aligned} y_i^{(1)} &= b_i \\ y_i^{(k+1)} &= y_i^{(k)} - a_{ik}x_k \ , \ i=1(1)n \\ x_i &= y_i/a_{ii} \end{aligned}\right\} \qquad (3.2.2.3)$$

which simplifies the boundary cell by removing the subtraction and b_{in} input.

Matrix triangularisation can be performed by using the array in Fig(3.2.2.2) which consists of an orthogonal triangular array for triangularizing the matrix and a collapsed (linear) array to modify the righthand side vector, the whole array computes using the augmented coefficient matrix A from (2.3.2.1).

Row 1 of the array forms the input boundary and accepts every row of A, every row that arrives has its first entry set to zero and the

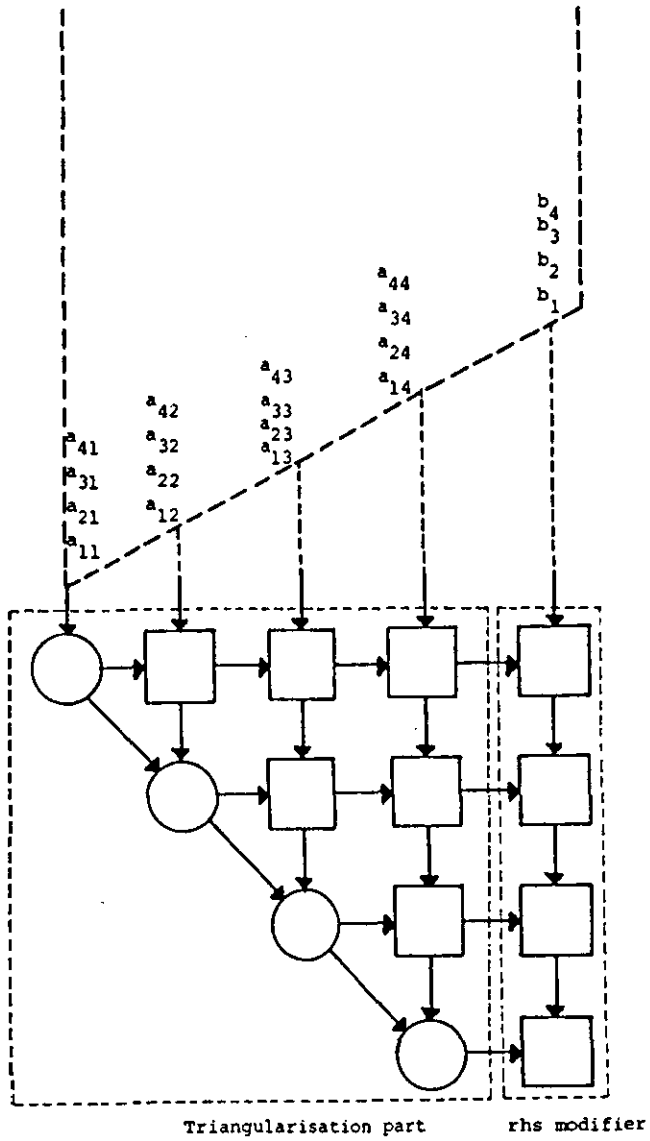
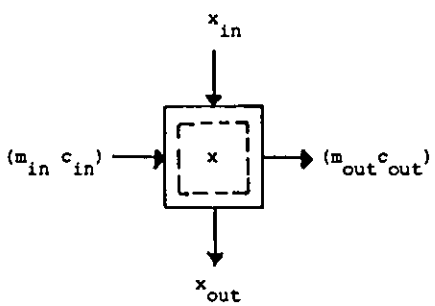


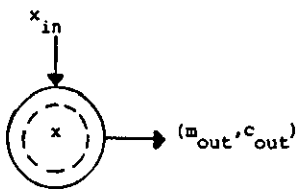
FIGURE 3.2.2.2: Array for matrix triangularisation

rest of the row updated, this corresponds to removing the first column of A. Likewise row 2 of the array accepts the resulting modified rows from row 1 of the array and zeroes their first entries corresponding to removing column 2 of A. In general, row i accepts modified rows from row i-1 and zeroes their first entries - corresponding to removing column i of A. Notice that the data sequences are skewed so that modification information in the form of a multiplier can be pumped right and meet with all elements of a particular row. A limited

pivoting strategy known as pairwise pivoting can be incorporated which only interchanges adjacent rows and is still numerically stable (Sorenson [85]). When pivoting is used during the computation row i of the array holds only the current row i of A and final placement of rows is obtained only on the last step of the algorithm. The basic cells for Gaussian elimination are defined below,

Shape





Procedure

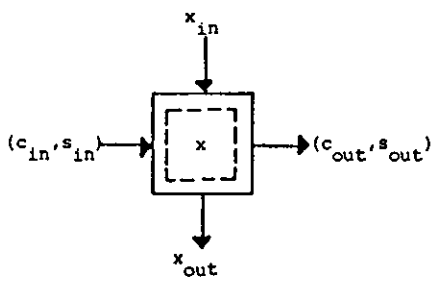
IF c_{in} THEN
$\{x_{out}=x+m_{in}x_{in}$
$x=x_{in}$
$\}$
ELSE
$x_{out}=x_{in}+m_{in}\ x$ (3.2.2.4)

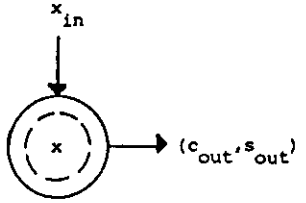
IF $|x_{in}|\geq|x|$ THEN
$\{$IF $x_{in}<>0$ THEN $m_{out}=-x/x_{in}$
ELSE 0
$x=x_{in}$, $c_{out}=1$
$\}$ (3.2.2.5)

ELSE
$\{m_{out}=-x_{in}/x$, $c_{out}=0\}$

and corresponding cells for the orthogonalisation process in (2.3.2.6) are given below with pivoting no longer necessary as

Shape





Procedure

$x_{out}=c_{in}x_{in}-s_{in}x$
$x=s_{in}x_{in}+c_{in}x$. (3.2.2.6)

IF $x_{in}=0$ THEN
$\{c_{out}=1$, $s_{out}=0\}$
ELSE
$\{\Delta=(x^2+x_{in}^2)^{\frac{1}{2}}$
$c_{out}=x/\Delta$
$s_{out}=x_{in}/\Delta$
$x=\Delta$
$\}$ (3.2.2.7)

Notice that only (3.2.2.4) can be computed in a straightforward IPS cycle. For (3.2.2.5) we must assume that we can negate x_{in} in parallel with the test $|x_{in}| \geq |x|$, and that the comparison takes the same time as addition/subtraction, before the cycle time for elimination is equal to an IPS cycle. For the Givens rotation cells, (3.2.2.6) is bounded by 2 IPS cycles but (3.2.2.7) requires the complicated square root. Hence a basic cycle is bounded by the cost of the boundary cell computation.

Theorem 3.2.2.2: The triangularisation of an augmented matrix A consisting of a full n×n matrix A and n×1 vector d requires $O(n^2)$ basic cells and T=3n basic cycles.

Proof: [from Fig.(3.2.2.2)]

(i) After n cycles the first column has been eliminated.

(ii) After 2n cycles the last rhs component enters the array and filters down to the last cell hence,

(iii) After 3n cycles the final modification of the rhs occurs. An additional n cycles for outputting results is neglected as they play no part in computation.

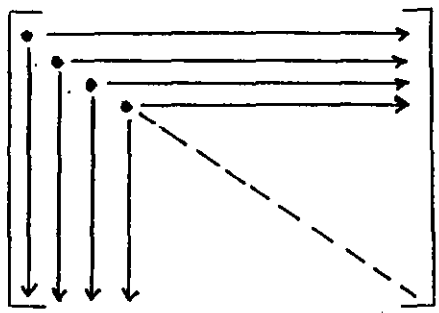
A second array which is suitable for triangularising a band matrix can also be constructed and is given in Gentleman & H.T. Kung [81], it is based on a diagonal sequence ordering like the arrays for matrix vector and product operations given above and a version appears in Chapter 6.

Next consider matrix factorisation defined by (2.3.3.3) and in equivalent recurrence form is given by,

$$\left.\begin{aligned} a_{ij}^{(1)} &= a_{ij} \\ a_{ij}^{(k+1)} &= a_{ij}^{(k)} + \ell_{ik}(-u_{kj}) \end{aligned}\right|$$

$$\left.\begin{aligned} \ell_{ik} &= \begin{cases} 0 & i<k \\ 1 & i=k \\ a_{lk}^{(i)}/u_{kk} & i>k \end{cases} \\ u_{kj} &= \begin{cases} 0 & k>j \\ a_{kj}^{(k)} & k\leq j \end{cases} \end{aligned}\right\} \qquad (3.2.2.8)$$

Schematically the order of formation for ℓ_{ik} and u_{kj} is



with ℓ_{ik} represented as columns and u_{kj} as rows. The factorisation procedure computes a row and column then updates the submatrix of $a_{ij}^{(k)}$ left, before starting the next row and column. A corresponding systolic algorithm uses a 2-D systolic array shown in Fig (3.2.2.3). The circular cell receives u_{kk} from the south and sends it north and computes the reciprocal $1/u_{kk}$ outputting it south west, and represents the '●' in the above sketch. The remaining cells are all hex IPS cells except for those on the upper boundary which are connected differently. On the upper boundary left of the circule hex cells are rotated 120° clockwise and perform the ℓ_{ik} computations denoted by the vertical lines, hex cells on the upper boundary right of the circle are rotated 120° anti-clockwise and form $-u_{kj}$ terms denoted by horizontal lines in the sketch. The wavefronts of the input data correspond to the chevrons formed by the horizontal and vertical lines together. Thus, the upper boundary computes the factors with the rest of the hex modifying the

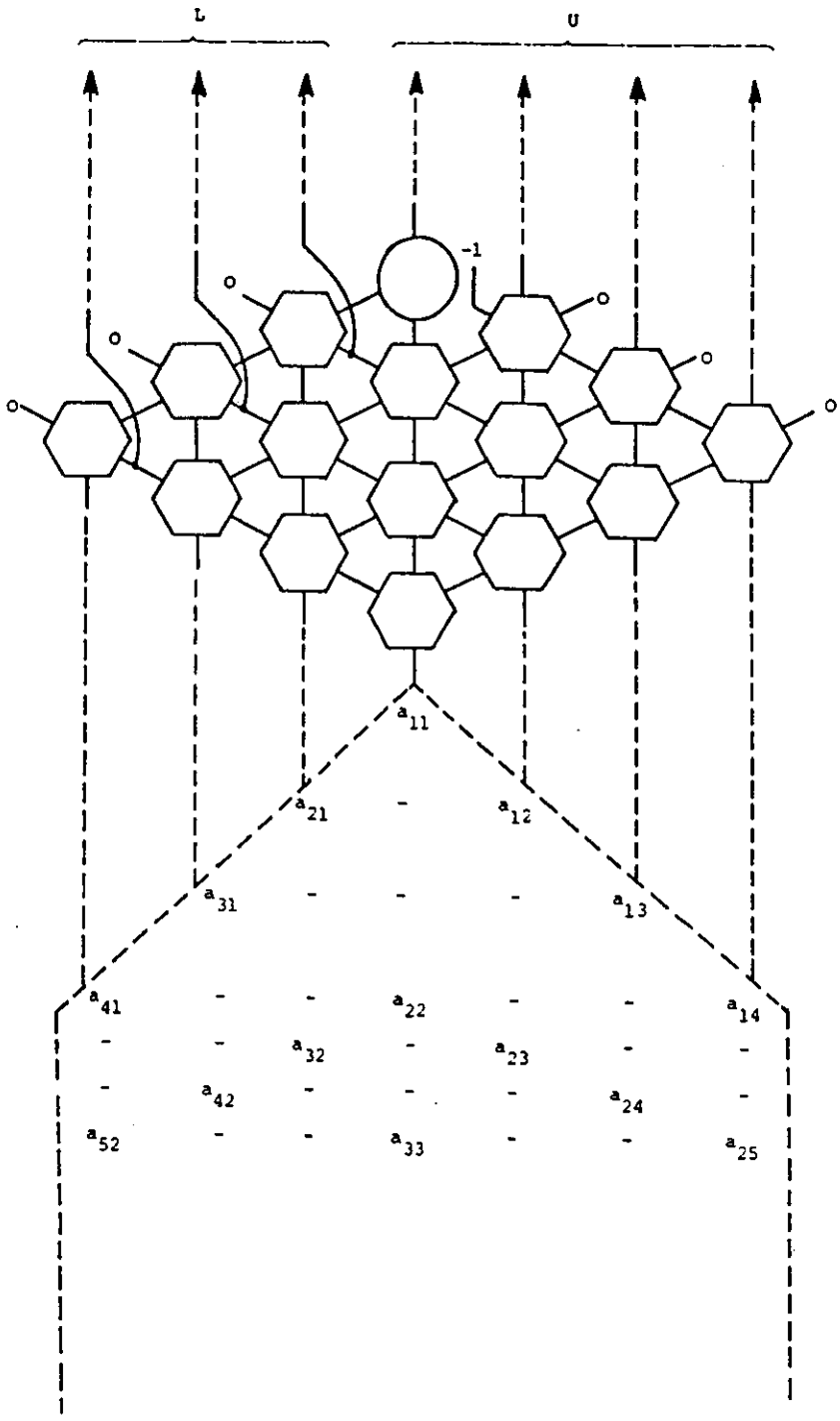


FIGURE 3.2.2.3: Systolic array for matrix factorisation

submatrices. Tracing the wavefronts from the array shows that successive row and column factorisation is overlapped pipelining sub-matrix modifications and producing:

Theorem 3.2.2.3: Let A be an n×n band matrix with w=p+q-1, a systolic array with pq hex IPS equivalents can form the LU factors in T=3n+min(p,q) IPS cycles which includes input output time.

Proof: [From Fig.(3.2.2.3)]

(i) From the diagonal input sequences the total number of inputs is w and this defines the array dimension to be pq.

(ii) Computation starts when a_{11} reaches the circular cell, this takes min(p,q) cycles.

(iii) The length of the a_{ii} sequence (which is the longest) is 3n and this is the maximum number of outputs for a single stream. At one output per cycle the timing follows.

Notice that when A is dense we use only n^2 cells and 4n cycles, an equivalent orthogonal mesh preloaded with A would require similar area and time. In the case of a symmetric matrix only the rows or columns of the sketch need to be explicitly constructed and this can reduce the hex cell count by half. Finally we note that once the triangular forms are found they can be pipelined into back and forward substitution arrays to solve the system. The method for triangularisation is utilised in least squares calculations as illustrated in Gentleman & H.T. Kung [81].

3.2.3 Arrays for Iterative Solution of Linear Systems

Systolic arrays for iterative solution of equations were first reported in Berzins, Buckley & Dew [83] and Dew [84]. The basic principle is to devise a linear systolic array for performing a single

iteration and then cascade a number of them to pipeline successive iterations. This cascaded iterative pipeline or array is illustrated in Fig.(3.2.3.1) the solution vector is successively approximated by cascading it through r linear arrays to output finally the rth iteration. The coefficient matrix A is pipelined through the r arrays and synchronised with $x^{(i)}$ on each level as is the righthand side vector d. We assume that A has a simple splitting so that the E and F matrices in (2.4.3) can be formed 'on-the-fly'.

Now if the iteration matrix M and vector c from (2.4.3.1) were known and M was banded, each linear array would simply be a matrix vector array like Fig(3.2.1.3)except that M would have to flow south out of the cells and the recurrence (3.2.1.4) would be initialized with $y_i^{(1)}=c_i$, i=1(1)n where c_i are the components of c. For a band matrix with bandwidth w=p+q-1 the first component of $x_1^{(i)}$ would emerge 2p cycles after $x_1^{(i-1)}$ entered the array and p cycles after it met m_{11}, the first matrix input to the array. It follows that matrix elements (and hence data sequences) must be delayed by 2(p-1) cycles between corresponding cells in array i and i-1 for successive iterations to synchronise. Hence each IPS cell must be augmented with a delay queue of 2(p-1) delay registers for the south output. The righthand side vector c (in this case) also has to synchronise but this is dependent on the sizes of p and q and can be derived in a similar way, thus the boundary cells at the right in Fig(3.2.3.1) are simply delay queues.

Theorem 3.2.3.1: r iterations of the form $x^{(i)}=Mx^{(i-1)}+c$ where M is an $n\times n$ band matrix with bandwidth w=p+q-1 and c is an $n\times 1$ vector requires rw IPS cells (augmented with delay queues) and time T=2n+r(2p-1)+MAX(p-1,q-1)-p+1.

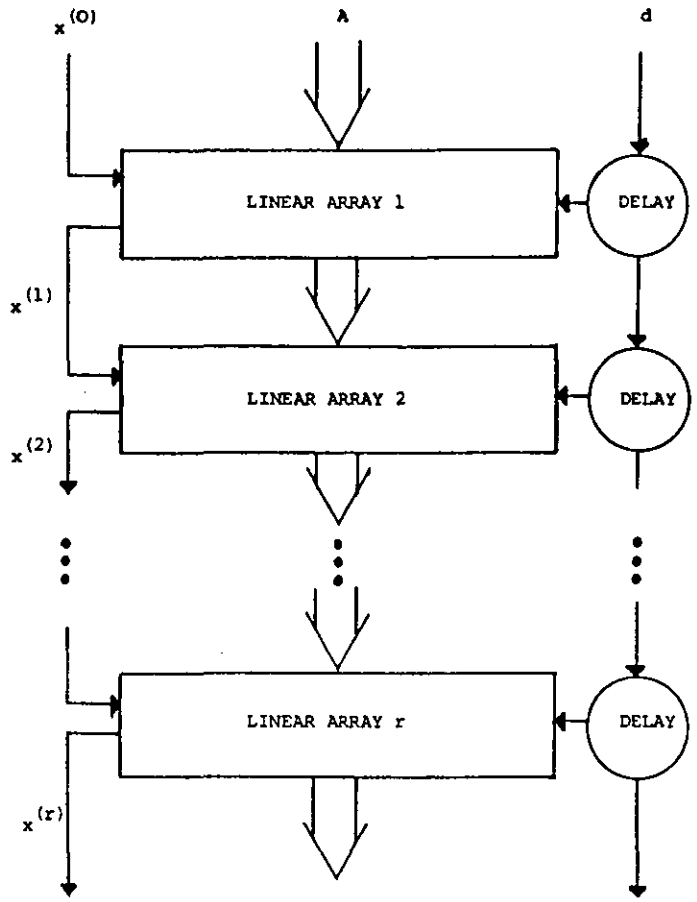


FIGURE 3.2.3.1: Cascaded iterative pipeline

Proof: [using Fig.(3.2.3.1) and Fig.(3.2.1.3)]

(i) Initialisation for first array is max(p-1,q-1) and first component is output in an additional p cycles.

(ii) Each array has latency of output 2p-1 thus the 1st result is ready to emerge from the pipeline after (r-1)(2p-1) cycles.

(iii) There are 2n cycles required to output all components.

If M and c are unavailable alternative linear arrays can be considered for the Jacobi and Gauss-Seidel schemes using modified inputs.

The Jacobi iteration scheme is illustrated in Fig.(3.2.3.2) and consists of a matrix-vector array which forms $z=Bx^{(i-1)}+b$ and a boundary cell which gives $x^{(i)}=D^{-1}z$. Thus computing (2.4.1.1).

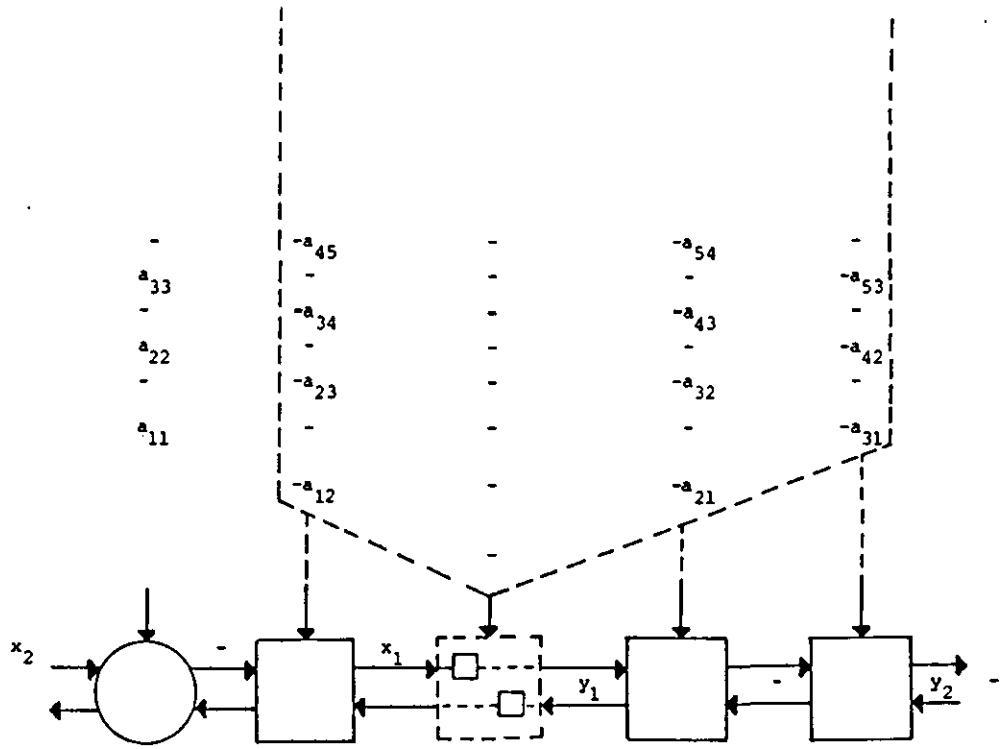


FIGURE 3.2.3.2: Array for single Jacobi iteration (w=4)

The cell in the matrix vector array associated with the zero main diagonal of B can be replaced by a simple delay cell performing no computation, and the boundary cell consists of a divider and accepts the elements of D. The timing for the array can be easily derived from Theorem (3.2.3.1) by substituting $\tilde{p}=p+1$ for p to take account of the extra delay through the boundary cell.

For the Gauss-Seidel method (2.4.2.1) a single iteration is a composite array constructed from an upper triangular matrix vector array for $z=Ux^{(i-1)}+b$ and a backsubstitution array for $(D-L)x^{(i)}=z$.

Theorem 3.2.3.2: r iterations of the Gauss-Seidel iterative method for an $n\times n$ band matrix A with bandwidth $w=p+q-1$ requires rw ips cells

(augmented with delays) and T=2n+r(2p-1).

Proof: [from Fig.(3.2.1.3) and Fig.(3.2.2.1) and Fig.(3.2.3.1)]

(i) The upper triangular matrix vector array has p-1 cells and thus a component input to a linear array for iteration takes 2(p-1) cycles to produce the equivalent z component from $Ux^{(i-1)}+b$.

(ii) The backsubstituter can be arranged so the z vector enters as the righthand side requiring only a single delay to produce the $x^{(i)}$ component.

(iii) The total delay is 2p-1 cycles between input and output of components the 1st result arrives at output of the rth linear array after r(2p-1) cycles.

(iv) There are 2n cycles to output all the result components hence T=2n+r(2p-1).

Improvements to these basic schemes will be introduced later in the text, but for the moment we can observe that as long as w<<n r iterations of an iterative scheme of form (2.4.2) can be computed in O(2n) IPS cycles.

3.3 THEORETICAL CONCEPTS FOR MANIPULATING SYSTOLIC ARRAYS

So far the mapping of a systolic algorithm into a processor geometry has been achieved in an ad-hoc manner. As designs become more complex this task becomes more difficult and error-prone, and a systematic methodology to synthesize systolic designs can save considerable time and effort. For the restricted class of algorithms from the 2-D systolic space and a regular constrained frame, namely algorithms with highly regular structures expressed as recurrence

relations, transformational methods based on data dependencies provide powerful manipulation tools. In this section we concentrate on two representations of the systolic algorithm, firstly, by a computational graph (implicit in the examples of the previous section) and second an algebraic form. The former pictorial representation is useful when combined with a geometry for deriving hardware specifications and the latter for algebraic transformations on a design. These approaches have a number of attractive features as they:

1. Indicate an automatic method of manipulating existing and complex designs by so-called "symbol pushing" techniques used in algebra to prove theorems.
2. Permit reasoning about systolic designs without recourse to detailed sketches and traces.
3. Admit the possibility of formal verification of design correctness.
4. Identify and derive sequences of transformations to produce alternative designs which may be better than existing ones.

The starting point is a formal mathematical model for the data sequence processor geometry combination of the systolic array.

3.3.1 Systolic Array Model [cf. Melhem & Rheinboldt [84]]

The model is based on a strict formalisation of the intuitive and informal representation used above expressing data sequences as wave-fronts the main characteristics are:

a) Data items on communication channels = data sequences
b) Computations of cells are modelled by systems of difference equations (involving operations on data sequences).

c) Input/output descriptions indicate the global effect of the network computations and are deduced by solving systems of difference equations.

d) The resultant output descriptions verify correct network operation.

The model itself can be further subdivided into models for the data and geometry.

Abstract Model of Data:

Notation: $\mathbb{N}$ = set of integers, $\mathbb{R}$ = set of reals,

δ = "don't care elements"

$$\mathbb{R}_\delta = \mathbb{R}_U \{\delta\}$$

A data sequence is an infinite sequence whose elements are members of $\mathbb{R}_\delta$.

Definition 3.3.1.1: Operations on data sequences are logical extensions of operations on elements of $\mathbb{R}$ and subdivided into:

(i) *δ-regular operators:* where δ"op" x=x"op"δ=δ for all $x \in \mathbb{R}_\delta$ and model destructive interference. (3.3.1.1a)

(ii) *non δ-regular operators:* where x"op"y=y"op"x if $x,y \neq \delta$ and x"op"δ=δ"op"x=x (3.3.1.1b)

which models constructive and neutral interference respectively.

Definition 3.3.1.2: Any data sequence η can be defined as a mapping $\mathbb{N} \to \mathbb{R}_\delta$ whereby $\eta(i)$ for $i \in N$ is the ith element in the sequence. The set of all data sequences $\mathbb{R}^*_\delta = \{\eta \mid \eta: \mathbb{N} \to \mathbb{R}_\delta\}$.

Hence an operation on two sequences $\eta_1, \eta_2 \in \mathbb{R}^*_\delta$ produces a third sequence η_3 according to

$$\eta_3(i) = \begin{cases} \eta_1(i)\text{"op"}\ \eta_2(i) & \text{if } \eta_3(i) \text{ is defined} \\ \delta & \text{otherwise,} \end{cases}$$

and *scalar product* for sequences is similar to that of vectors with $\eta \in \mathbb{R}_\delta^*$ and $\omega \in \mathbb{R}$ yielding, $\zeta=\omega\eta \in \mathbb{R}_\delta^*$.

Definition 3.3.1.3: The set of *bounded data sequences* $\overline{\mathbb{R}} \subset \mathbb{R}_\delta^*$ with only a finite number of non δ elements. The end of the sequence is defined by a *Termination function* $T: \overline{\mathbb{R}}_\delta \to \mathbb{N}$ such that for $\eta \in \overline{\mathbb{R}}_\delta$, $T(\eta)$ is the position of the last non-δ element.

Actions on collections of bounded data sequences can then be formalized by an n-ary sequence operator.

Definition 3.3.1.4: An n-ary sequence operator Γ is a transformation $\Gamma: [\overline{R}_\delta]^n \to \overline{R}_\delta$ where $[\overline{R}_\delta]^n = \overline{R}_\delta \times \overline{R}_\delta \ldots \overline{R}_\delta$ the cartesian product of n copies of $\overline{R}_\delta$.

For instance, the basic inner product cell structures a), b) in Fig.(3.2) effect a transformation $\Gamma: [\overline{R}_\delta]^3 \to \overline{R}_\delta$ which could be written

$$\Gamma_{ips}(\xi,\eta,\zeta) = \Omega[\xi+\eta.\zeta] \qquad (3.3.1.2)$$

for $\xi,\eta,\zeta \in \overline{R}_\delta$, where "(" enclose arguments and "[" signifies grouping, and Ω is a delay symbol defined below.

Definition 3.3.1.5: The *shift* (Ω) and *spread* (θ) operations for sequences ξ,η,ζ are given by $\Omega^k\xi=\eta$ and $\zeta=\theta^r\xi$ where

$$\eta(i) = \begin{cases} \delta & i \leq k \\ \xi(i-k) & i>k \end{cases} \qquad (3.3.1.3)$$

$$\zeta(i) = \begin{cases} \xi_{\frac{(i+r)}{(r+1)}} & , \ i=1,r+2,2r+3,\ldots,(n-1)r+n\ldots \\ \delta \text{ otherwise.} \end{cases} \qquad (3.3.1.4)$$

and for bounded data sequences β, β_1 and β_2 satisfy the properties:

(i) $\Omega^r\Omega^k\beta=\Omega^{r+k}\beta$

(ii) $\Omega^{(r+1)k}\theta^r\beta=\theta^r\Omega^k\beta$

(iii) $\omega[\theta^k\beta]=\theta^k[\omega\beta]$, $\omega[\Omega^k\beta]=\Omega^k[\omega\beta]$

(iv) $\Omega^k[\beta_1\text{"op"}\beta_2]=\Omega^k\beta_1\text{"op"}\Omega^k\beta_2$, $\theta^k[\beta_1\text{"op"}\beta_2]=\theta^k\beta_1\text{"op"}\theta^k\beta_2$

Shift and spread operations can be applied to individual sequences and permit the addition of δ-elements to achieve synchronisation, Ω^0 signifies no delay and will be useful later.

The following example illustrates the use of Ω and θ. Let $\xi=a_1,a_2,a_3,a_4,\delta,\delta,\ldots$ be a sequence then $T(\xi)=4$ and $\xi(i)=a_i$ for $i=1(1)T(\xi)$. Now $\Omega^3\xi=\delta,\delta,\delta,a_1,a_2,a_3,a_4,\delta,\delta,\delta,\ldots$ and $\theta^2\xi=a_1,\delta,\delta,a_2,\delta,\delta,a_4,\delta,\delta$.

Finally an n-ary sequence operator is termed a *causal operator* if the ith elements of its input sequences affect the jth element of its result or output sequence for j>i, and *weakly causal* if the ith output element relies on the first i elements of the input sequences.

Abstract Model of Geometries:

The processor geometry is represented as a loopless multigraph

$$G(V,E,f_+,f_-)$$

where V=set of nodes(processors), E=set of arcs (communication paths) and f_+,f_-: E→V are two functions mapping arcs to nodes defining the direction of communication. Any two nodes can be connected by a number of arcs as G is a multigraph and $f_+(e)\neq f_-(e)$ for any $e \in E$ prohibiting direct loops. The set of nodes V can be further partitioned into three disjoint subsets such that $V=V_S \cup V_I \cup V_T$ with the definitions below,

V_S = a set of source nodes (with no arcs directed into the nodes)

V_I = a set of interior nodes defining the geometry

V_T = a set of sink nodes (with no arcs directed out of the nodes)

An arc e is directed into a node V iff $f_+(e)=V$ and directed out if $f_-(e)=V$, consequently sources define points at which data enter a systolic space and sinks points where data leaves and interior nodes potential computation sites. Finally we define a colouring function

$col: E \to C_E$ mapping arcs to a finite set of colours (C_E); such that each arc has a colour, and all the incoming edges of a node have different colours, and likewise different outgoing edges also have different colours. Corresponding incoming and outgoing edges can be allocated the same colours if required.

A systolic array is then simply constructed by assigning a sequence $\xi_e \in \overline{R}_\delta$ to each arc $e \in E$, and for each node v with n incoming arcs and m outgoing arcs a set of m,n-ary sequence operators $\xi^i = \Gamma^i_v(\eta^1,\ldots,\eta^n)$ i=1(1)m defining how input sequences affect output sequences and consequently processor structure. It is clear that nodes from V_I represent geometry processors and sources/sinks act as pumping stations, with each arc a uni-directional communication link.

Definition 3.3.1.6: A subset $\overline{V}_I \subset V_I$ of interior nodes is *homogeneous* if:

(i) All nodes in $\overline{V}_I$ have the same number of incoming and outgoing arcs say n and m respectively.

(ii) The colours of the n incoming edges and m colours of the outgoing arcs are identical for every node.

(iii) The n-ary sequences defining a node are generic in the sense that every node is described by the same set of sequences denoted,

$$\zeta^i_v = \Gamma^i(\eta^1_v,\eta^2_v,\ldots,\eta^n_v) \quad i=1(1)m,$$

and so are independent of any node.

It follows that if $V_I = V_I^{(1)} \cup V_I^{(2)} \cup V_I^{(3)} \ldots V_I^{(k)}$ are k disjoint homogeneous subsets of V_I that the graph can be termed k-partially homogeneous. As examples the matrix factorisation of Fig.(3.2.2.3) is 4-partially homogeneous (due to rotations of hex ips) and backsubstitution

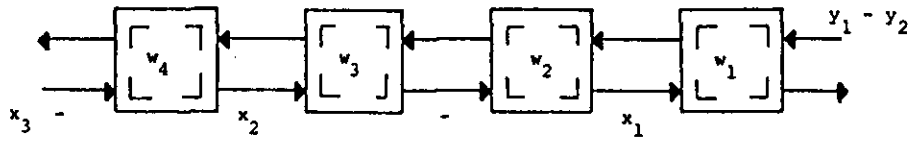
in Fig.(3.2.2.1) is 2-partially homogeneous. Implicitly part (3) of Definition (3.3.1.6)defines a stationary instruction group hence dedicated array.

The above definitions are sufficiently powerful to provide a limited verification capability for arrays which have a repetitive pattern and so fit our regular frame. To illustrate the verification we consider only a single simple example from Melhem & Rheinboldt [84] where more complex examples are also given.

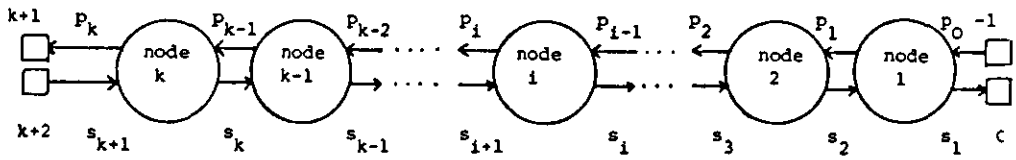
The problem is the 1-D convolution problem derived from a 4-tap Finite Impulse Response (FIR) filter with coefficients w_1, w_2, w_3 and w_4, which produces the matrix vector problem,

$$\begin{bmatrix} w_1 & w_2 & w_3 & w_4 & & \\ & w_1 & w_2 & w_3 & w_4 & \\ & & \ddots & \ddots & \ddots & \ddots \\ & & & w_1 & w_2 & w_3 \\ & & & & w_1 & w_2 \\ & & & & & w_1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \qquad (3.3.1.5)$$

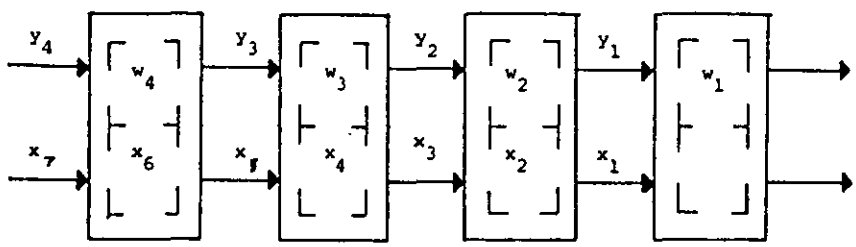
normally we are interested in only the first n-k+1 y components in this case k=4, and the w_i, i=1(1)k are termed weights. A systolic array can be derived easily from Fig.(3.2.1.3) (as mentioned previously) by making the matrix input stationary but requiring the preloading of the w_i values. The network is shown in Fig.(3.3.1.1a) on the start of the first computation cycle, each cell is a simple three-way IPS modified to contain a register holding the w_i value thus removing the vertical connection. Fig.(3.3.1.1b) shows the equivalent multigraph using two colours P and S augmented with a subscript label to identify



a) Two-way convolution



b) Coloured computational graph



c) Uni-directional convolution

FIGURE 3.3.1.1: Representations of 1-D Convolution

different arcs, square boxes represent sources and sinks,circles interior nodes, and the graph is homogenous. Verification of the design is as follows:

(i) The input sequence definitions

$$\sigma_1 = \Omega^{k-1}\theta\eta \quad \text{(for initial values of y associated with P)} \tag{3.3.1.6}$$

and

$$\pi_k = \theta\xi \quad \text{(for x associated with S)} \tag{3.3.1.7}$$

with $T(\xi) = n$, $T(\eta) = n-(k-1)$ and $\xi(t) = x_t$.

(ii) n-ary sequence operators are given by,

$$\pi_{i-1} = \Omega\pi_i \tag{3.3.1.8}$$

$$\sigma_{i+1} = \Omega[\sigma_i + w_i\pi_i] \tag{3.3.1.9}$$

This definition is generic as the graph is homogenous, the cell is a simple IPS.

(iii) Derivation of output sequences:

By repeated substitution $\pi_i = \Omega^{k-i}\pi_k$ (3.3.1.10)

and substituting (3.3.1.10) in (3.3.1.9) yields,

$$\sigma_{i+1} = \Omega\sigma_i + w_i[\Omega^{k-i+1}\pi_k] \tag{3.3.1.11}$$

Now (3.3.1.11) has the form $\sigma_{i+1}=\Omega\sigma_i+\Delta_i$ i=1(1)k+1 its solution is

$$\sigma_r = \Omega^{r-1}\sigma_1 + \sum_{j=1}^{r-1}\Omega^{j-1}\Delta_{r-j}\ , \ r=2(1)k+1 \tag{3.3.1.12}$$

The proof of this fact follows by induction,

for i=1 $\sigma_2 = \Omega\sigma_1+\Delta_1$, which satisfies (3.3.1.12) for r=2

assume (3.3.1.12) is true for r=1(1)k, then,

for r+1
$$\sigma_{r+1} = \Omega\sigma_r + \Delta_r = \Omega[\Omega^{r-1}\sigma_1 + \sum_{j=1}^{r-1}\Omega^{j-1}\Delta_{r-j}] + \Delta_r$$

$$= \Omega^r\sigma_1 + \sum_{j=1}^{r-1}\Omega^j\Delta_{r-j} + \Delta_r$$

$$= \Omega^r\sigma_1 + \sum_{j=1}^{r}\Omega^{j-1}\Delta_{r+1-j}$$

It follows that (3.3.1.11) is solved by applying (3.3.1.12) with r=k+1 on $\Delta_{k+1-j}=w_{k+1-j}[\Omega^{k-(k+1-j)+1}\pi_k]$ producing,

$$\sigma_{k+1} = \Omega^k\sigma_1 + \sum_{j=1}^{k} \Omega^{2j-1}[w_{k-j+1}\pi_k] \qquad (3.3.1.13)$$

substituting for (3.3.1.6) and (3.3.1.7) the original input sequences,

$$\sigma_{k+1} = \Omega^{2k-1}\theta\eta + \sum_{j=1}^{k} \Omega^{2j-1}[w_{k-j+1}.\theta\xi] \qquad (3.3.1.14)$$

and on further manipulation using the properties of Defn.(3.3.1.5) equation (3.3.1.14) becomes,

$$\sigma_{k+1} = \Omega^{2k-1}\theta\eta+\Omega\theta \sum_{j=1}^{k} \Omega^{j-1}[w_{k-j+1}\xi] \qquad (3.3.1.15)$$

From the ordering of the w_i coefficients in the graph it follows that

$$\sigma_{k+1} = \Omega^{2k-1}\theta\eta+\Omega\theta\Omega^{k-1}\beta$$

$$= \Omega^{2k-1}\theta[\eta+\beta] \qquad (3.3.1.16)$$

where $\beta_j(t)=w_{k-j+1}\xi(t)$ and as $\eta(t)=0$ i.e. initial results of accumulation zero

$$\sigma_{k+1} = \Omega^{2k-1}\theta\beta$$

with $T(\beta) = n-k+1$ and,

$$\beta(t) = \sum_{j=1}^{k} \beta_j(t+k-j) = \sum_{j=1}^{k} w_{k-j+1}x_{t+k-j} = \sum_{\substack{q=1 \\ t=1(1)T(\beta)}}^{k} w_q x_{t+q-1}$$

and this is the algebraic form of (3.3.1.5).

Finally Fig.(3.3.1.1c) indicates an alternative systolic array for the same problem where both streams move in the same direction, and y_i's moving twice as fast as the x_i's, and the padding neutral elements have been removed. This is desirable because only n cycles would be required to read the n-components of the solution vector instead of 2n

cycles in the former case. The cell is another variation on the IPS cell and is given later. The above proof is significant because it illustrates that systolic arrays can be verified algebraically.

3.3.2 Transformation Rules

The two main objectives of transformation rules are:

(i) To modify existing and correct systolic designs to produce new designs which have improvements over the old ones.

(ii) To convert non-systolic algorithms into equivalent systolic designs.

For purposes of discussion it is necessary to assume that the design to be modified is correct, in (i) this is achieved by using the verification technique of the previous section while for (ii) an alternative method must be found. In the case of systolic designs an essential feature of transformations is that they preserve the 'systolic property' or pumping action. In its weakest form this means that a single data or instruction item must reach different nodes of the graph at different times, for non-systolic designs a graph must be modified to give it this property before further improvements can be made. Once a transformation is validated it can be used as a legitimate rule in all designs with similar arrangements. In this sense validated means that the rule preserves the systolic property and the correctness of the computation. It follows that sequences of validated transformations applied to a verified design result in a new array which is implicitly verified. Hence verification can take place only on arrays which suit the proof techniques.

We can assess the validity of transformation by considering their

effects on three quantities (Ullman [84]) which are:

D = the delay of data to propagate through a cell

S = the spacing of elements within a sequence (i.e. the effect of the spread operator)

P = the period or cycles between real inputs in a sequence arriving at a cell.

The aim of a transformation is to modify D,S or P, such that DS=P is preserved, which usually indicates that the array continues to operate correctly.

Definition 3.3.2.1: Two given designs are *equivalent* if for initial values given for one design there exist initial values for the other design with the same input function and the two designs compute essentially the same output function.

One final quantity termed latency is also useful and defined as:

Definition 3.3.2.2: The *latency* of a systolic array is the number of cycles between the first input and the first output.

Now, there are two main types of transformations we can consider.

1) Re-timing of data flow around the network by adding or removing delay elements (Ω).

or 2) Re-placement of cells which can detect isomorphisms between data flow in different designs to simplify processor geometries.

The principles of re-timing can be summarised by the following lemmas and theorems which use a timing graph notation called the *Ω-graph representation* (modified from the z-graph representation of H.T. Kung & Lin [83]) which is simply a geometry graph whose arcs indicate delays with a label Ω^k for k delay cycles.

Lemma 3.3.2.1 (k-slowing)

If all the delays on arcs of a timing graph are multiplied by k>1 then the cells can be redesigned so the array network will work correctly but a 1/kth speed (i.e. k-slowed).

Proof:

(i) The network will perform the same function if the period of all streams at all nodes are multiplied by a constant, and is simulated by all processors taking k cycles instead of one, with k-1 idle cycles.

(ii) The equation DS=P is modified to (kD)S=kP

(iii) Two data streams a,b which travel between nodes u and v and have delay Ω^{ℓ} and Ω^{m} arrive at v by times that differ by $|\ell-m|$. Multiply by k and they arrive at times differing by $k|\ell-m|$ so if the process is k slowed operation is preserved.

Note: If k delays are added to each arc timing is no longer preserved.

Lemma 3.3.2.2 (timing shift)

Suppose k>0, and that v is a node whose outgoing arcs all have delay greater than k, we can add k delays to each incoming arc and subtract k from all outgoing arcs. Provided processors are modified correctly the network will perform correctly. The reverse procedure applies if all incoming edges have delays greater than k, and k is added to outgoing edges and subtracted from incoming edges.

Proof:

(i) Intuitively delays before and after a cell provide a certain amount of slack time and k represents the largest shift in time that the cell can make and still produce its result before or at the time required by other cells dependent on its output.

(ii) If u is a processor node and some arc leaving u has delay D, and u needs D cycles to compute its result before the theorem can be applied the cycle time must be changed so that computation completes in at most D-1 cycles.

(iii) To preserve DS=P, the change in D must be compensated by S or P. It is not easy to change P locally or globally, and so changing delays from D to D-1 make S change to DS/(D-1).

Using these two Lemmas and the assumption that there are no loops of zero delays (Ω^{O}) the retiming theorem for general graphs follows:-

Theorem 3.3.2.1 (retiming): Any finite network with cycles consisting only of zero-delay arcs can be transformed into a network performing the same function whose delays are at least 1.

Proof:

Let N be the original network with no zero-delay cycles, define lag(v) of every node to be the longest path consisting of zero-delay arcs ending at v.

(i) lag(v) is finite and the amount v must lag behind nodes of lag zero.

(ii) There is at least one node with lag zero otherwise tracing back arcs would locate a cycle of zero-delay because N is finite and we eventually repeat nodes.

(iii) An arc from u to v with delay O in N will be given a delay lag(v)-lag(u) as the largest path to u must be shorter than the largest path of v making the delay positive.

(iv) A likely problem is that v may be the source of an arc with delay d>O to a node w with lag much less than v and we would have to subtract lag(v)-lag(w) from arc with delay d. In this

case apply Lemma(3.3.2.1) to select k such that,

$$dk > lag(v) - lag(w)$$

where v and w are taken over all nodes so d>0 for v to w.

The network is now reconstructed in a two stage process. First, select k and apply Lemma(3.3.2.1). Second, delay each node of lagℓ by ℓ cycles and for all arcs from nodes u to v with delay d=0 replace with delay lag(v)-lag(u), for arcs with d>0 replace with dk+lag(v)-lag(u).

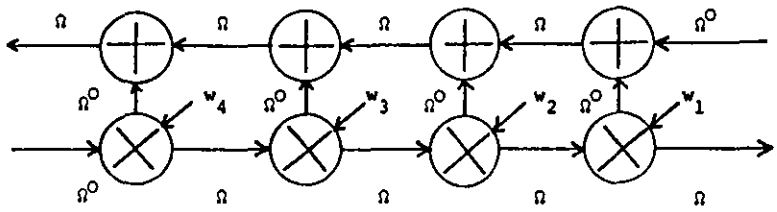
These three mechanisms constitute the famous re-timing Lemmas of Leiserson and Saxe [83], the problem with these techniques is that for large designs they can become quite complex. In H.T. Kung & Lin [83] and H.T. Kung and Lam [84] alternatives have been suggested but the latter is the simplest and can be summarized as:

Theorem 3.3.2.2: [Cut Theorem]

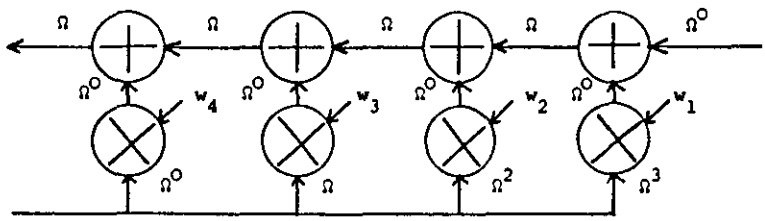
For any design, adding the same delay to all the edges in a cut and to those pointing from sources to sinks set of the cut will result in an equivalent design.

Recently, Dew, Manning, & McEvoy [86] presented a more general notion of a cut which allowed delays on all arcs of the cut set, a cut partitions the nodes of a graph into two sets with undirectional dataflow between them.

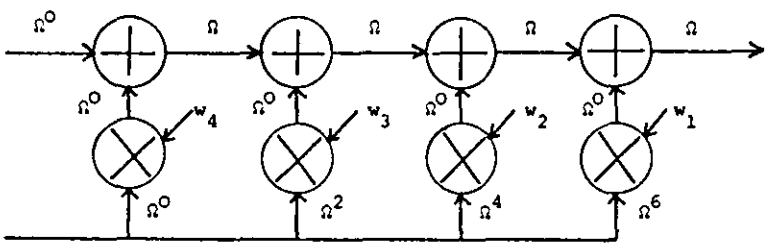
Retiming methods can also be used to derive new arrays as shown in Fig.(3.3.1.2), part a) is the timing graph of Fig.(3.3.1.1a) and part d) is the equivalent timing graph for Fig.(3.3.1.1c). In part b) the point to point connections of the x input are turned into broadcasting wires by using additional delays preserving computation. In part c) the direction of the y stream is reversed and by collecting terms in reverse order the unidirectional form is produced by observing the delays necessary on x to synchronise with y as it filters through. The broadcast



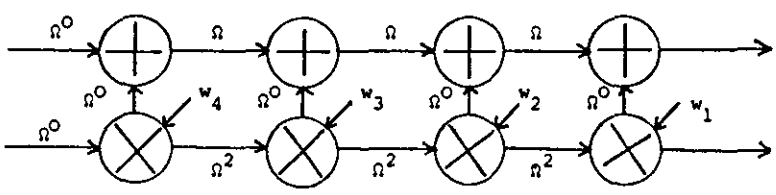
a) Λ -graph representation of two-way convolution



b) Semi-systolic convolution



c) Uni-directional semi-systolic convolution



d) Uni-directional convolution

FIGURE 3.3.1.2: Retiming of convolution scheme

line is then removed by sharing delays producing point to point connections.

An analogous method to retiming aimed at processor geometries is replacement. The general technique is to modify the design by allowing the processor geometry to move identifying isomorphisms between data-flows of different designs and determining a path or locus of computation sites which a smaller network could visit to achieve full computation. In practical terms this often means folding the data streams to fit a stationary but reduced network. We shall only consider a simple example comparing the two matrix multiplication arrays of Fig.(3.2.1.5) and Fig.(3.2.1.6). The orthogonal array has two-directions of dataflow and is stationary allowing data to be stored in cells between steps. The hexagonal scheme has three-way dataflow and is non-stationary with no data residing in the cells between steps. In the latter case the dataflow splits into three disjoint sets as illustrated by Fig.(3.3.1.3)

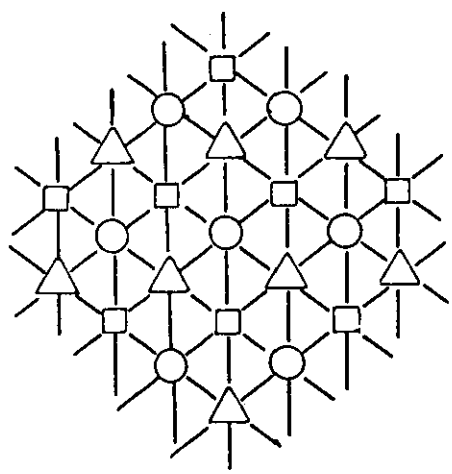
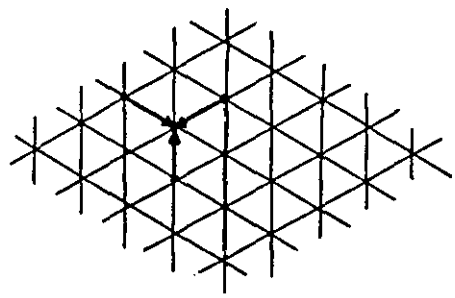


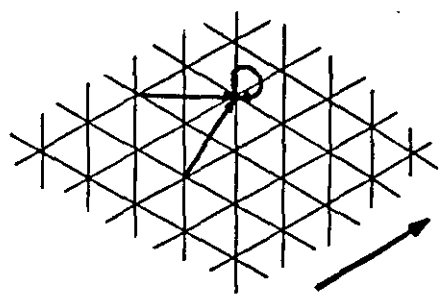
FIGURE 3.3.1.3: Rote transformation

with circles, triangles and squares. A data element participates in constructive computation by passing from circle to triangle to square then back to circle. This fact and the Huygens principle ensure that the three data sets never meet and indicates that these problems could

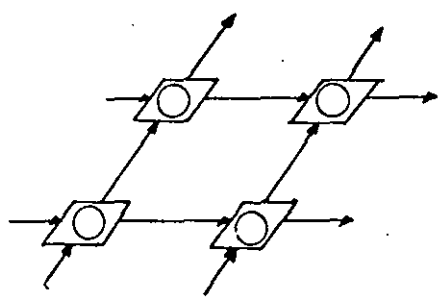
be interleaved on the same array. The argument in reverse implies that each matrix multiplication on a hexagonal array is really three problem instances interleaved. Now embedding the hexagonal grid into an infinite grid representing the systolic space as shown by Fig.(3.3.1.4a) with normal data flow. If we move the grid relative to the data in the direction shown by Fig.(3.3.1.4b) the dataflow appears to have changed with one data path becoming stationary. Drawing the relevant part of



a) Hex dataflow



b) Shifted dataflow



c) Rectangular form

FIGURE 3.3.1.4: Relationship between hex and orthogonal geometries

the grid in Fig.(3.3.1.4c) indicates that connections between only the circular (or star or triangles) are possible producing a rectangular form and hence uses only a third of the original cells. A simple reorganisation of the dataflow and rotation of the rectangular grid gives the form in Fig.(3.2.1.5).

For general replacement strategies three steps are necessary once a structure on the infinite grid has been found.

1. The bounds of the array are located:-

   As the grid moves the union of all the visited locations is the new array, and it may occur that data movement operations in the old algorithm fall outside the new design saving time.

2. The place of input and output are located:-

   This is trivial if arrays have an initial position for input and satisfy simple and regular dataflows.

3. Processor actions:-

   By moving the processor grid the instruction groups become non-stationary and so processors may no longer be stationary and will require some type of control mechanism.

Note: If we start with a rectangular array we can also develop a hexagonal scheme.

Using retiming and replacement strategies a better hexagonal matrix multiplier which uses $T=n+w$ time can be derived by reversing the direction of the result connections, a version of the array appears in Huang & Abraham [84].

Finally retiming and replacement can be viewed as operations on elements (designs) of a systolic frame. A *sub-frame* can then be defined as the set of all designs for the same underlying algorithm, and an

Anchored sub-frame as a sub-frame with at least one formally verified design. Furthermore, a *closed subframe* is a subframe, such that for any design from it, application of retiming and replacement methods always produces an alternative design in the subframe. Clearly the 1-D convolution problem is an anchored subframe and is probably the closest to being a closed subframe in existing literature. From the designers point of view an anchored sub-frame is useful because there is no need to verify designs formally just apply the timing and placement rules. When a sub-frame is shown to be closed there is no point investigating its systolic algorithms further in a mathematical sense, only implementation methods can improve their operation.

3.4 PRACTICAL CONSIDERATIONS AND VLSI

In this section we examine and justify the heuristics used to derive a constrained systolic frame, which are applicable to chip implementation of arrays. The first step in the argument is to restrict the systolic space model to a more formal set of constraints which back up experience in circuit production, and allow reliable reasoning about practical implementation problems.

3.4.1 The Grid Model

For VLSI implementation purposes the systolic model is constrained into a 2-D space and gives rise to a variety of circuit models attributed to Thompson and Brent & Kung among others, and discussed in Savage [81]. The grid model we use has many common features with these models and is a modified version of the one given in Ullman [84] and has the following components:

1. The systolic space is constrained to a 2-D rectangular grid which is further stratified into a number of layers fixed before we start a design.
2. Wires run horizontally or vertically, and on the same grid line there can be at most one wire in each layer.
3. Circuit elements occur only at grid points, wires entering grid points are inputs, those leaving outputs.
4. If the number of inputs to an element are >4 a shape can be placed over a number of grid points to provide the correct number.
5. There is limited fan-in, that is a small finite number of inputs to a circuit element.

These definitions cover heuristics H[1] and H[3] with H[2] following almost immediately. The grid model itself is representative of VLSI processing techniques because the manufacturing process requires three layers for wires in different materials (polysilicon, diffusion and metal) see Mead & Conway [79] and while 2. appears restrictive it is a method often used in real manufacturing techniques and is supported by the following theorem.

Theorem 3.4.1.1: If C is a circuit in the grid model that uses k layers in which to run wires. A second circuit $\overline{C}$ can perform the same function as C in the same time using 2 layers but k^2 times the area of C with wires in one layer running only horizontally or vertically.

Proof: [Ullman [84] pg.37]

Notice that diagonal connections can be implemented by a staircase arrangement of horizontal and vertical wires passing through a number of grid points which contain no circuit elements. Thus it becomes

possible to embed a timing graph or processor geometry for arrays given in the previous sections on to the grid. The geometries are planar and so require only a single layer of the grid to accommodate its wires. It should be clear that if Δx and Δy are the grid lines spacing in the two available directions the area of the circuit is bounded by the smallest rectangle which contains all the grid points. If we further assume that these points form a convex set the area is only over-estimated by a factor of two at the most - this is termed the *convexity assumption* and the extra area is asymptotically negligible. Once the geometry is embedded we can reduce Δx and Δy to provide a finer grid creating a number of grid points inside each cell. In the case of the hexagonal array of Fig.(3.2.1.6) the circuit in Fig.(3.2a) would be added to the smaller grid covering points bounded by the hex's. The refinement continues until finally basic fabricable logic elements are allocated to grid points. Now the internal arrangement of the hex IPS involves wires that cross and is avoided on the grid by using the additional layers. Consequently, if the systolic array geometry is planar (H[2]) the complexity of wiring using different layers is controlled and often implies regular geometries.

To justify the remaining heuristics the notion of timing must be examined in more detail. So far the notion of time has been given by a cycle which implicitly assumes that:

1. A computation is a discrete computational step marked by a series of beats or clock pulses.
2. There are a limited number of logic elements through which signals propagate in one time step, and the transmission of signals on wires is instantaneous.

Thus H[3] is justified because if no limit was placed on inputs to a

grid point the size of the basic cell could be unlimited, hence propagation delay through its circuit elements would be variable and length of a time step difficult to derive. Furthermore we have defined a cycle to be the time to complete an inner product step of one multiplication and one addition, the circuits to perform these operations are made up of logic elements which themselves must be switched on and off to compute results. So the basic cell cycle consists of a number of smaller cycles or clock ticks defined by the basic switching time of logic components. The time between ticks (or pulses) is dictated by the properties of the materials used to implement the components and control the speed at which electrical signals can propagate through logic elements and down wires. Even for the fastest devices propagation delays must be given serious consideration. H[4] and H[5] are now easily justified because as a wire gets longer the more time a signal takes to travel its length; eventually wires will be so long that data arrives at the wrong clock tick and computation is invalid. This phenomenon is called data skew, a solution is to increase the period between clock ticks slowing down the computation, or to limit the length of wires to allow time for the signal to arrive. The situation is further exacerbated by the fact that the clock tick itself must be broadcast to components in all the cells of the geometry and so tend to be long wires, an added constraint on the clock period. It follows that broadcast of data should also be avoided.

3.4.2 Area/Time Tradeoffs

In the examples given so far the theorems have graded designs according to cell count and computation time. These seem quite natural

measures but also have practical significance.

During the factorisation process of a cricuit many opportunities arise for a flaw to appear in a chip making it useless. In fact the largest feasible circuits have a very low probability of producing and unflawed chip of about 0.1 (or 10%). The total cost of manufacture must be spread over all the 'good' chips making the cost inversely proportional to yield. If we multiply the area of a chip by a constant C>1 the probability of a good chip is $(0.1)^C$ and costs increase by 10^C. Consequently designs with small area are of paramount importance.

Minimising computation time is also important, because existing problems can be solved faster, and bigger problems previously impractical may become solvable. Furthermore, in real-time applications like signal and image processing the ability to provide results for time dependent processes is crucial.

The selection of a particular algorithm for a problem is a complex tradeoff between area (A) and computation time (T) which is inherently problem dependent. An idea of the best possible tradeoff can be derived theoretically from the grid model using lower bound arguments on area and time, when a bound is achieved it indicates that our design is in a sense optimal. Three basic bounds which have had some success use an area-time solid which can be formed by taking all the snapshots of an array and stacking them on top of one another. First we can argue about the volume of the solid giving us a lower bound on AT, second we can argue that a certain amount of information must flow temporally across the boundary between two snapshot levels. And, thirdly consider the amount of information flowing across a line (partitioning the grid by cutting its sides of smallest length) during execution of the algorithm

and which gives us a lower bound according to AT^2. The strongest or these three bounds is the AT^2 result, because it represents a bound which matches the best circuits that can be constructed, results for hexagonal matrix multiplication and related problems using AT^2 can be found in Savage [81] and extensions in Lin & Wu [85]. In our case, the importance of lower bounds is that they indicate room for manoeuvre between existing designs and the optimal bound betraying the possible existance of a new design. These new designs can then be created by modifying A and T to produce an overall reduction in AT^2.

There are basically two levels at which we can attempt to modify A and T, first at the array level by re-timing and re-placing cells or using fundamentally different algorithms for the same problem, and second modifying the techniques of implementation. The former technique is adopted throughout the thesis and is not discussed here, while the latter idea has been considered in detail by Fisher [84]. For area reduction we can consider re-arranging the cell changing it from a bit parallel implementation to a byte or bit serialized version. Serialization attempts to reduce the area by replacing parallel logic by a smaller piece of logic performing the same task but requiring a larger number of steps. The problem here is that additional circuitry in the form of latches for data and logic for controlling the serialized calculation must be added and offset the initial reduction. Serialization also affects the cycle time of a cell due to the increased number of steps, but this is also offset by a reduction in the circuit clock period due to smaller propagation delays through the cell logic. Related to serializing a parallel computation is pipelining which breaks up the computation into a number of not necessarily identical stages with the

same delay. All the original logic is used with the addition of delay registers between each stage to allow successive problems to be overlapped. This technique gives rise to the term *two-level pipelining* in systolic arrays, where the first level is the global pipelining between array cells, and the second level the pipelining of computations within a cell. An example is shown in Fig.(3.4.2.1) for the unidirectional convolution algorithm.

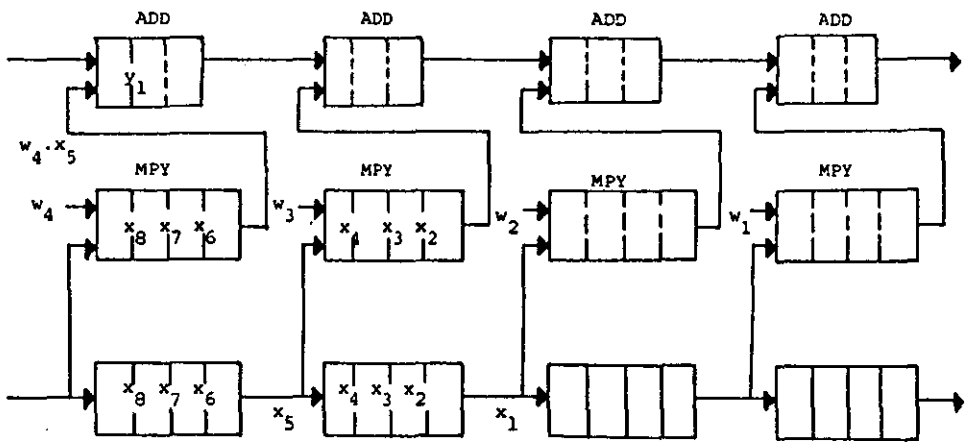


FIGURE 3.4.2.1: Two level pipelining of convolution

Two levels are useful because only the propagation delay of individual stages is important allowing a reduction in clock period, but limited because the individual problems pipelined cannot depend on the results of closely preceding problems as results will still be in the pipe. Thus, the technique encounters problems with arrays with feedback like backsubstitution (see H.T. Kung & Lam [84]).

Another significant problem in implementing a systolic array as a chip is communication with the outside world. A bit parallel approach is expensive on pins (off chip connections) as each input or output requires enough pins for accommodating the word length. Serialization has the advantage of producing hardware independent of word length and

thus reducing pin count, an attractive feature as the chip perimeter is fixed by its area. Finally, when area considerations are not so important we can combine serialization and pipelining to increase the throughput of the system, as the convolution example indicates. However the parallelism in the algorithm must be sufficient to keep the pipelines full, otherwise the possibility of interleaving problem instances must be investigated to maintain efficiency of cells.

3.4.3 Fault Tolerance

Serialization and global redesign of the algorithm are not the only ways to cope with manufacturing problems. Another intuitive approach is to design a complex circuit and then increase the resolution of the implementation technique to scale down the feature size (which defines basic dimensions of wires and logic elements), and 'stuff' the circuit into a smaller area. The problem with this is that scaling down cannot continue for ever and eventually a circuit will be produced for which there is no alternative but to increase area. Furthermore as resolution approaches its limits the electrical properties of wires and components limit performance. For instance, the propagation delay as basic elements shrink is reduced but the corresponding delay on interconnections remains the same, so eventually transmission of signals dominate the speed of operation creating a lower bound on the clock period.

A more devious approach to solving the area/yield problem is to increase yield significantly so that area is no longer a good measure for cost. For instance, if yield (i.e. probability of an unflawed chip) was 0.9 (90%) then increasing area by a constant C>1 would produce

a yield of $(0.9)^C$, hence yield decreases slowly which in turn raises cost slowly, making it less sensitive to chip area. This approach is adopted by fault tolerence techniques aimed at producing more reliable and robust chips; and essentially designs a circuit so that it can be restructured after fabrication. Restructuring avoids the flaws in a chip by routing around faulty elements allowing a defective chip to be used. Circuits employing fault tolerance can be envisaged as a four part design (Sami & Steffanalli [86]):

1. An original array of cells/processors.
2. Spare cells/processor (preferably arranged in simple patterns)
3. An interconnection network (usually a grid) consisting of data paths, switches and control paths for reconfiguration.
4. A control algorithm performing a fault tolerance algorithm.

Using this model fault tolerant circuits and strategies can be designed to cope with two types of failure:

1. *Production defects:-* which are high with current fabrication technology.
2. *Operational defects:-* which occur while the device is operating inside a real system.

The corresponding fault tolerant strategies are termed static and dynamic respectively. Static schemes test and fix defects before the chip is packaged and contribute only to production costs and are performed just once. Dynamic schemes cater for operational defects and must be applied many times in a real computing environment consequently static schemes generally use less hardware than dynamic schemes which require programmable switching elements and additional control networks so restructuring can be performed automatically. Where the area of a basic

cell is not significantly more than the extra connections and switches to implement fault tolerence dynamic schemes are questionable.

The reconfiguration or restructuring whether static or dynamic is not usually performed as a direct one-to-one substitution of faulty cells for spare ones, but depends upon the components of the model present. For instance, if 1. and 3. are available but 4. is omitted there are no spare cells and 3. is reduced to a fixed set of paths which must be reconfigured manually, for instance by laser programmable links (e.g. VLSI RAMS) making the scheme static. The lack of spare cells means that a smaller network is produced and consequently can only solve smaller problems. When all the models components are present the spare cells can be used by re-configuring the network around faulty cells. In these cases the control algorithm and network are simplified if the whole row or column containing faulty cells is switched out of the network, resulting in a number of perfectly good cells being wasted. These wasted cells can be recovered by a more complex control algorithm which incorporates a global re-naming procedure mapping the structure of the array onto the available working cells. Renaming introduces additional problems over and above the complexity of the control algorithms as it requires a highly flexible communication network and may introduce long wire connections for large number of cell failures. Many variations are possible on these dynamic schemes such as fault stealing structures where rows and columns can steal good cells from adjacent rows and columns, but all trade hardware and complexity for increased reliability and flexibility.

As well as error detecting and correcting tolerance schemes there are also error masking schemes. For example, in Triple Module

Redundancy (TMR) only 1., 2. and 4. of the model are retained, but each original cell has two spares with it. The control algorithm is built into these cells which can check for faults and switch in a spare cell if necessary to mask the error. An alternative is to redesign the original network to incorporate 4. with only a few extra cells to compute where errors occur but fix the results rather than modifying the cell network.

An implicit assumption in all these fault tolerent systems is that the extra hardware incorporated cannot be faulty, or at least has a significantly less chance of failure than the original cells. As fault tolerence techniques become more complex they may contradict this basic premise.

Fault tolerence techniques can be tailored to systolic arrays by using the additional information about the regularity of data flow, and produce effective low-overhead schemes. Briefly the salient features of the systolic fault tolerance schemes are as follows. The above model is restricted to 1. and 3. with the communication network further confined to nearest neighbour communication. Attention is then restricted to unidirectional arrays and basic cells are augmented with simple bypass registers and switches, as illustrated by Fig.(3.4.3.1) when a faulty cell is detected it is switched out using these extra connections and registers to maintain throughput but reducing the range of problems that can be solved. For instance, in the convolution example of Fig.(3.3.1.1c) a fault in a single cell would mean only three coefficients could be used instead of four. Notice that a series of faulty cells will not produce a long wire as the data will be fed from bypass register to bypass register of successive faulty cells maintaining

nearest neighbour dataflow. The amount of extra hardware is trivial and we can reasonably assume that the extra wires and registers will not be faulty. For 2-D arrays and arrays with feedback the problem is more complex and throughput is reduced rapidly as faults occur. A solution to the feedback problem is to convert feedback arrays into unidirectional ones, this is done by re-timing the array to create a systolic ring architecture, which mixes stationary and non-stationary dataflow. A systolic ring for backsubstitution is shown in Fig.(3.4.3.2) described below. The q-1 term recurrence of Fig.(3.2.2.1) can be implemented using only q/2 cells in the systolic ring, and is operated as follows. The q/2 most recently computed results are stored in the cells (one in each), while the partial results of the next q/2 results cycle around the ring meeting the stored values. Every two cell cycles

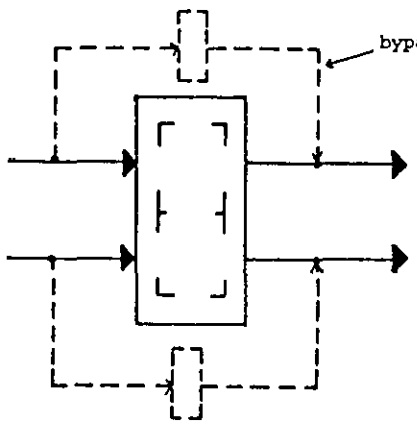


FIGURE 3.4.3.1: Fault tolerant convolution cell

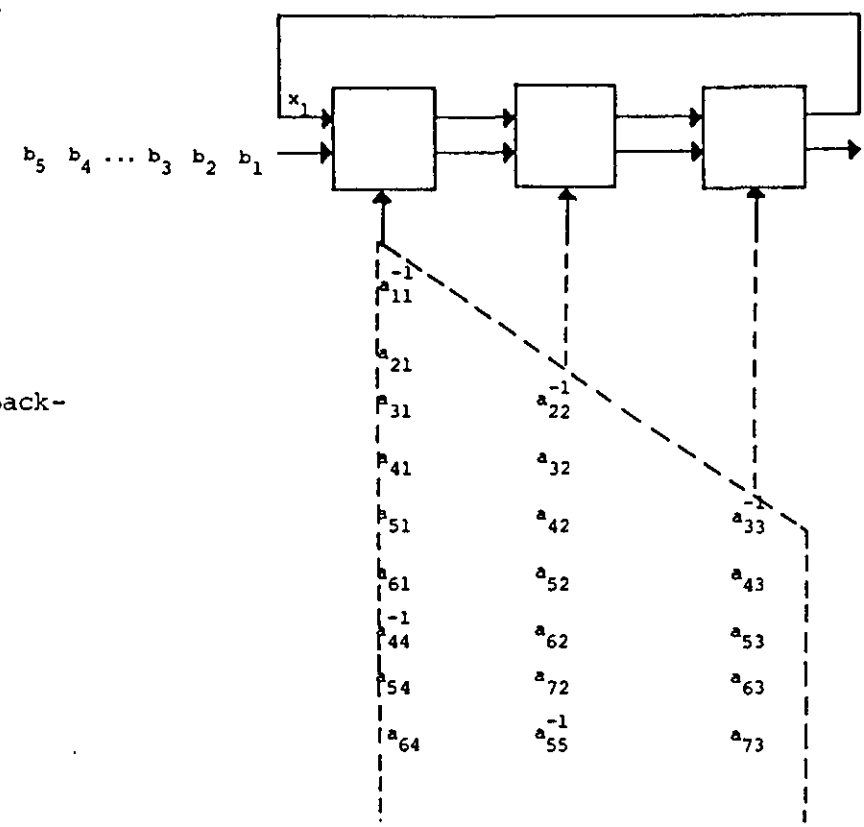


FIGURE 3.4.3.2:
Systolic ring for back-substitution (q=7)

a result is completed and gets loaded into a cell. The output period is the same as the feedback array but now cells work all the time improving efficiency and require only half as many, loading and unloading of values complicates the cells and requires an extra line for output on some problems but this is compensated by reduced cell count and increased fault tolerance. The division required by the former two-way array is now performed outside the array to avoid placing a divider in every cell. In addition to the backsubstitution ring 2-D ring arrays for triangularising and factorising a matrix requiring only a half and a third of the original cells and the same computation time exist. (The methods can be found in H.T. Kung & Lam [84]). In conclusion, the basic characteristics of the above schemes are that throughput is unaffected, interconnection length is not increased so clock period is unaffected and the arrays degrade gracefully as more faults are encountered.

In contrast to the principle of routing around bad cells is algorithmic fault tolerance (Huang & Abraham [84]). This technique is used to detect and correct errors resulting from permanent and transient errors, and is not generally applicable but for specific cases can be achieved with a very low overhead. Rather than tailoring the correction technique to architectures, algorithm-based tolerance is tailored to specific algorithms and consists of three parts:

(i) The input data is encoded

(ii) The algorithm is redesigned to operate on encoded data

(iii) The new algorithm is distributed onto the architecture.

For a low overhead the unencoded result or information part must be easy to recover, and enough redundancy must be available for the detection

and correction of errors.

One particular encoding is the checksum method which is applicable for many common matrix operations.

Definition 3.4.3.1: The row, column, and full checksum matrices for an $n \times m$ matrix $A=(a_{ij})$ are defined as follows:

(i) The *column checksum* matrix A_c of A is an $(n+1)\times m$ matrix consisting of A and the extra row $a_{n+1,j} = \sum_{i=1}^{n} a_{ij}$, $j=1(1)m$.

(ii) The *row checksum* matrix A_r of A is an $n\times(m+1)$ matrix consisting of A and the extra column $a_{i,m+1} = \sum_{j=1}^{m} a_{ij}$, $i=1(1)n$.

(iii) The full *checksum matrix* A_f of A is the $(n+1)\times(m+1)$ matrix is the column checksum matrix of the row checksum matrix of A.

Each column of A_f is an encoded vector, and the following results hold for $n\times n$ matrices,

$$A_c B_r = C_f \tag{3.4.3.1}$$

$$C_f = L_c U_r \tag{3.4.3.2}$$

$$A_f + B_f = C_f \tag{3.4.3.3}$$

Once a computation has been performed the fault tolerance scheme is as follows:

1. Detect error:
   a) Compute sum of information element of each row + column
   b) Compare results with corresponding checksums
   c) Errors are indicated by inconsistent values.
2. Locate error:
   Errors lie on the intersections of inconsistent rows and columns.
3. Correct error:
   (i) Add difference between computed row or columns sum and its corresponding checksum.

or (ii) Replace checksum by computed value.

Note: A small tolerance for round-off error should be allowed to avoid detecting a faulty cell incorrectly.

Additional cells are required to compute the checksum parts of the problems, but there is no need to route around faulty cells because error correction can be performed outside the array. Thus, the same type of problem can be solved even in the presence of errors. Unfortunately the scheme has only limited correction capability, and for certain configurations of errors the technique can be fooled. A more powerful method is the *weighted checksum scheme* reported by Jou & Abraham [86] and extended by Luk [86].

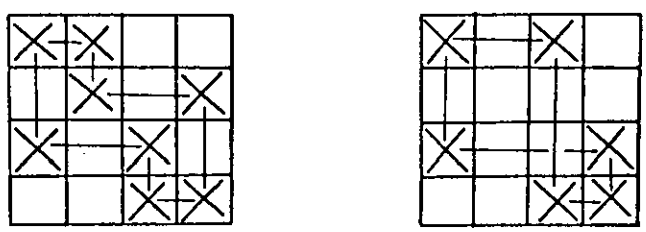


FIGURE 3.4.3.3: Uncorrectable faults in checksum scheme (for cf. n=4)

The normal procedure in chip manufacture is to fabricate a number of identical circuits simultaneously on a wafer of silicon. Once the chips are completed the wafer is diced and the individual chips tested, without fault tolerance the faulty chips are discarded at this stage. With fault tolerance virtually all the chips will be used and it becomes feasible to use the whole wafer as a single circuit. This process is termed Wafer Scale Integration (WSI) and is an active area of research. The larger area makes it possible to connect larger and separate subcircuits which ordinarily would reside on separate chips on the same wafer and removes the communication bottleneck caused by pin restrictions. Fault

tolerance is of course essential because the probability of producing an unflawed wafer are very remote. WSI also adds to our problems of signal propagation because the potential for longwires and large number of components through which signals must be driven increases dramatically. By using the grid model and consequently designs from constrained and regular systolic frames dataflow and cell complexity can be controlled. The global clocking mechanism however is another matter, clock skew is a key factor even for single chips, for WSI it could present a significant problem. Fortunately work by Fisher [84] indicates clocking these large systems is practical.

3.4.4 Synchronous Versus Asynchronous Array Operation

The essential feature of the systolic array is the mapping of data-flow into a simple and regular pattern implying communication between cells. In the examples so far it has been convenient to envisage cells computing in lock step or synchronisation. This view arises from the definition and structure of snapshot tracing making computations easy to follow, as computation is chopped up into discrete steps, and produces a global clocking mechanism. If we assume that the circuit has bounded communication delay and an ability to pipeline signals the clock pulse can be broadcast to cells in two ways:

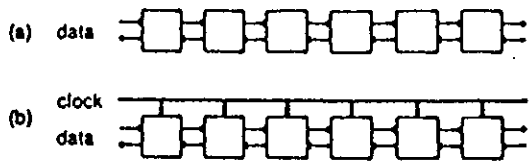
1. For 1-D arrays the clock period is made independent of array size by repeatedly folding the cell structure to bound clock skew.
2. In 2-D arrays a H-tree layout is used making all cells equi-distant from the root (clock source) of the tree again controlling clock skew.

As systolic arrays satisfy these assumptions global clocking even for VLSI is controllable.

An alternative method which avoids global clock distribution is self-timing or asynchronous communication. In a synchronous scheme, the clock signal tells cells when data is available and transfer between all cells is essentially simultaneous. In contrast, asynchronous schemes communicate by a predetermined handshaking protocol that allows cells to start computation as soon as its inputs are ready. The absence of a global clock avoids clockskew and allows variations in component speed to take advantage of data dependencies. The cost of this extra flexibility appears in additional hardware for handshaking and increased difficulty in testing of the synchronisation logic. This overhead is further compounded by the regular structure of arrays using uniform cell structures with a high dependency on input and output relationships which force the asynchronous scheme to run at its worst case speed. Consequently the tendency is to choose synchronised schemes.

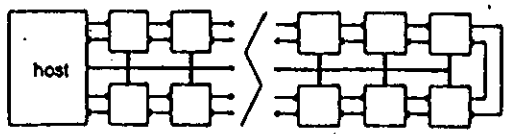
Where the size of a circuit is unbounded - for instance in the case of extendable arrays a trade-off between asynchronous and synchronous can produce a hybrid clocking scheme. Here, the circuit is partitioned into chunks of bounded size and each chunk given a clock node. Synchronisation between chunks is performed asynchronously and the clock nodes are then used to broadcast a clock pulse to all the cells in the chunk.

We conclude that the use of constrained and regular systolic frames to derive systolic arrays will produce designs suitable for possible VLSI implementation provided cell count and global input and output is limited.

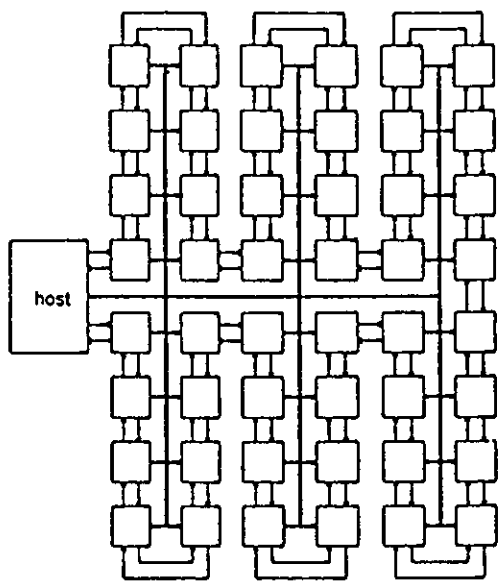


a) Ideally synchronised one-dimensional array

b) Corresponding clocked array

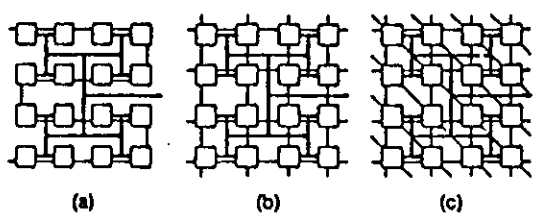


c) Array folded to bound clock skew

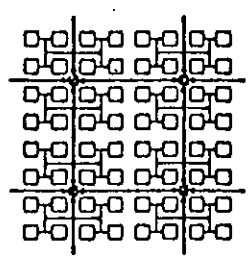


d) Comb layout of one-dimensional array

FIGURE 3.4.4.1: Bounding of clock skew using folding



H-tree layouts for clocking using a) linear arrays b) square arrays c) hexagonal connected arrays



d) Hybrid synchronisation scheme

FIGURE 3.4.4.2: H-tree clocking strategies

3.5 GENERIC ARCHITECTURES

For large applications it may not be feasible to design a single chip implementation of an array, especially when balance between flexibility, efficiency, performance and implementation costs is essential. An alternative implementation strategy is to implement basic cells at the board level using a collection of "off-the-shelf" components which are widely available as chip sets from various manufacturers.

Systolic arrays achieve high performance and efficiency by considering only restricted problem classes, at the expense of flexibility and some times implementation cost. A more economic solution results if arrays can be constructed which incorporate features for a number of systolic algorithms. These more flexible systolic forms are called generic systolic arrays, and in this section we shall briefly review the main contenders which have received attention to date.

3.5.1 The Warp Architecture

The Warp Architecture developed at Carnegie Mellon University (CMU) by H.T. Kung and his associates is the most advanced generic design for purely systolic algorithms. Its design began with a study to identify architectures of general purpose micro-processors which could implement a variety of systolic algorithms efficiently. The study resulted in the Programmable Systolic Chip (PSC) discussed in Fisher [84] and Fisher, H.T. Kung & Sorocky [84] and prompted research into cell structures for high performance systolic arrays in a particular area (in this case signal processing).

The Warp architecture is a 1-D linear systolic array with data and control flowing in one-direction with input at one end of the array and output at the other. From the preceding discussions we observe that the design allows easy implementation allowing synchronization by a simple global clock mechanism, minimum input/output requirements and the use of efficient fault tolerance techniques for faults. The basic Warp cell is constructed from a collection of chips as is illustrated in Fig.(3.5.1.1), its main characteristics being the pipelining of data and control. Weitek 32-bit floating point multiplier (MPY) and ALU perform operations and can be used in pipeline mode to improve throughput by two-level pipelining. The MPY and ALU register files use Weitek register file chips and can compute approximate functions like inverse square root using look-up facilities. The x,y and addr-files are also register files but this time used to implement delays for synchronising data paths, and can be used as extra registers for book-keeping operations, while the data memory is used to reduce the input/output band-width by implementing tables of data and storing intermediate results, it can also be used to implement multiple cells on the same processor and hence 2-D arrays. The crossbar and input multiplexors (muxes) provide communication between the individual elements and can be re-configured by control signals. The muxes permit two-directional data flow and ring setups (using wrap around). A 10-cell prototype has been built at CMU and tested on a number of example arrays discussed in H.T. Kung [84a].

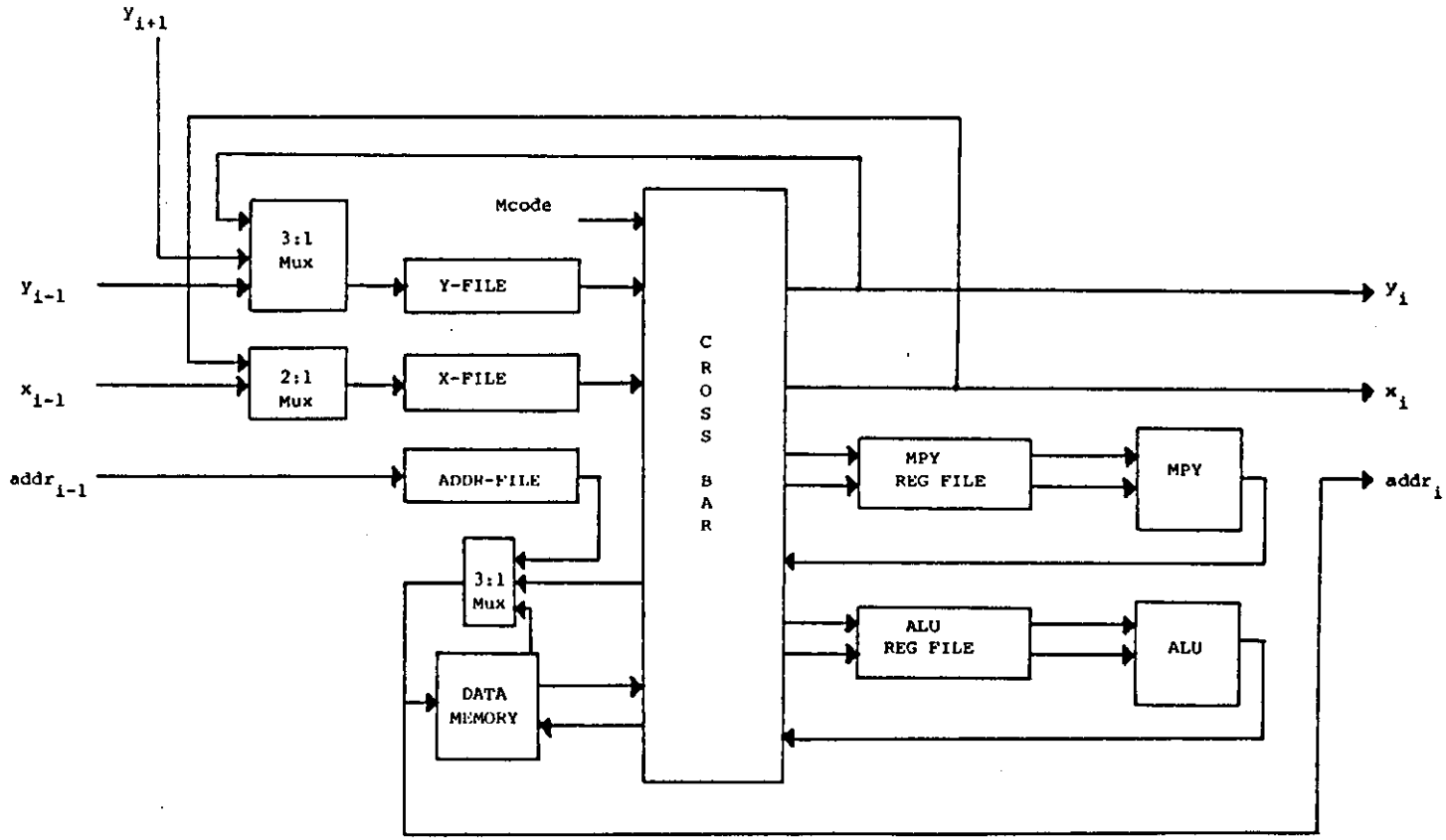


FIGURE 3.5.1.1: Data paths for the Warp cell

3.5.2 The Wavefront Array Processor (WAP)

The WAP was introduced by S.Y. Kung and consists of an $N^{\times}N$ processing element with regular connection structure, a program store and memory buffering modules as illustrated in Fig.(3.5.2.1). The processor grid acts as a wave propagating medium and timing is asynchronous, using handshaking protocols, thus no global clock is needed.

Each processor can perform a limited number of operations and is controlled by a program loaded from the program store. Data for problems is stored in memory modules around the boundary and extra time must be allowed to set up a computation. An algorithm is executed by a series of wavefronts moving across the grid with processors computing whenever its data and instructions are available. Processors are assumed to support pipelining of waves and the spacing of waves (T) is determined by the availability of data and the execution of the basic operation. The speed of the wavefront Δ is equivalent to the data transfer time.

Different algorithms are computed by changing the control program which is written in a special Matrix Data Flow Language (MDFL) (see S.Y. Kung, Arun, Bhasher, Rao & Hu [81]) which exploits the principles of waves acting according to Huygens principle. Non-wavefront algorithms can be converted (or systolized) to wavefront version using similar re-timing and cut theorems based on a signal flow graph, to those in Section 3.3.

To compare the WARP and WAP we can consider pipelining ability, architectural extendibility, programming simplicity and fault tolerance. The Warp is essentially a pipelined architecture and can use two level

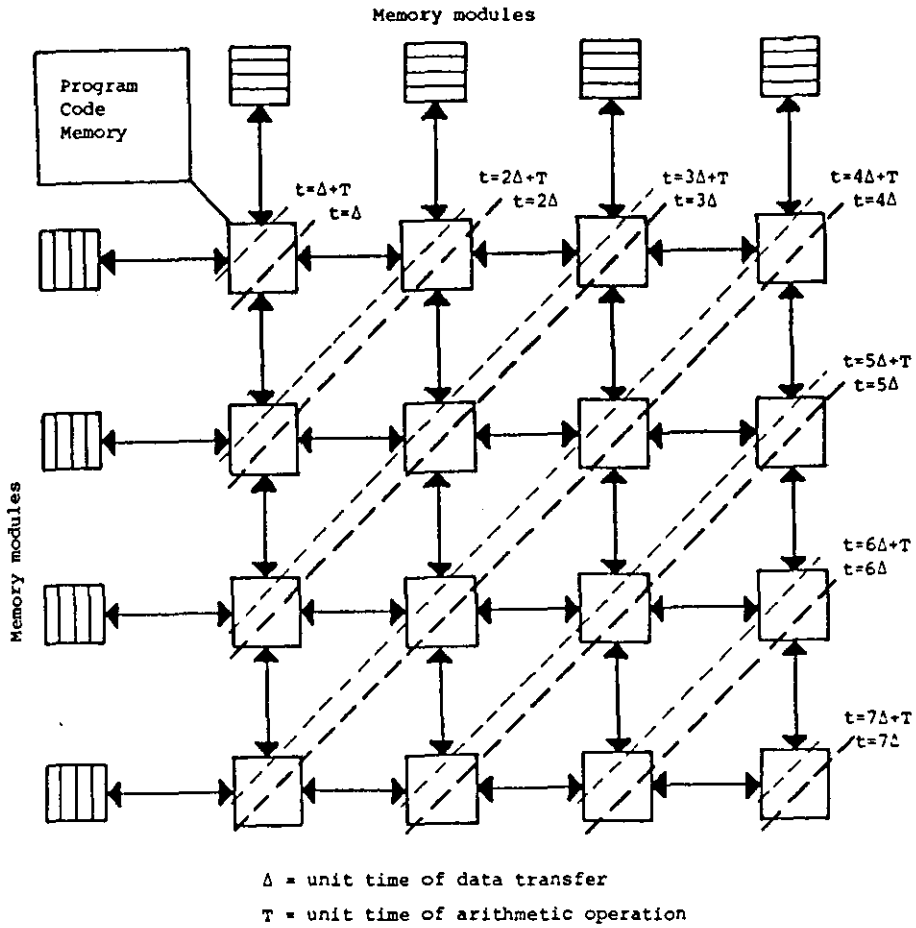


FIGURE 3.5.2.1: The Wavefront Array Processor

pipelining to improve throughput, as well as having the ability to perform stationary and non-stationary algorithms with special control. On the other hand the WAP being asynchronous can take advantage of data dependencies in problems which may result in significant speedups.

In terms of extendability the WAP is quite flexible, its lack of global clock allows the array size to be extended to very large sizes, whereas the synchronous Warp must use folding arguments to bound clock skew requiring reconfiguration on each upgrade of cells. The WAP is simpler to program too, having an architecture structure already well studied in SIMD and MIMD machines and supporting the MDFL language. The WARP on the other hand requires the pipeline of control and the

choice of a suitable language for performing while and repeat loops is still problematical. But in fault tolerance the WARP processor has an advantage as it can cope with faults quite easily whereas hardware recovery is quite difficult on the WAP without additional processors. A neat solution would be to modify wavefront algorithms, use the checksum procedures by simply adding an extra row and column of processors, but this would require extra computation stages to set up the checksum and fix the error.

3.5.3 INMOS Transputers and OCCAM

A third possibility is the INMOS transputer, a single chip microprocessor containing a memory, processor and communication links for connection to other transputers, which provides direct hardware support for the parallel language OCCAM. The structure of a transputer is given in Fig.(3.5.3.1).

The transputer and OCCAM were designed in conjunction and all transputers include special instructions and hardware which provide optimal implementations of the OCCAM model of concurrency and communication. Different types of transputers can have different instruction sets depending on the required balance between cost, performance, internal concurrency and hardware, without altering the users view of OCCAM. Hence the transputer is a Reduced Instruction Set Computer (RISC)

The processor contains a scheduler which enables any number of processes to run on a single transputer sharing processor time, while each link provides two uni-directional channels for point to point communication synchronised by a handshaking protocol. Communication

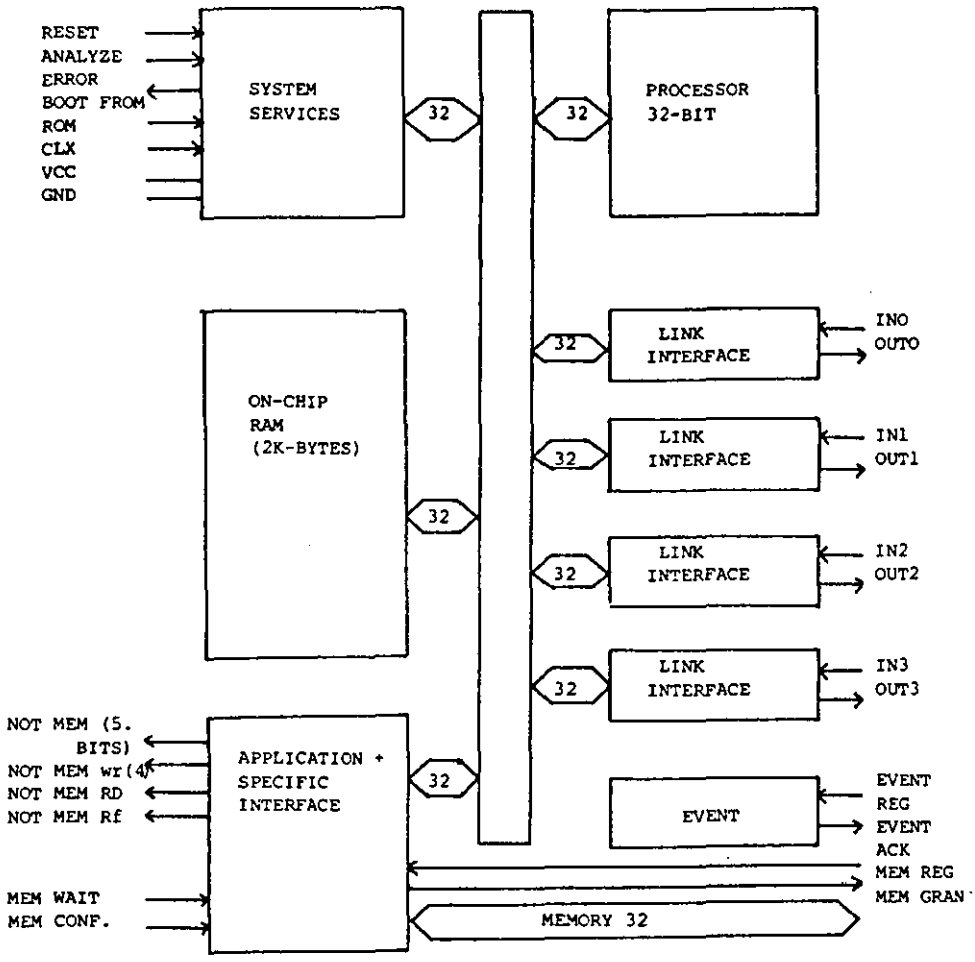


FIGURE 3.5.3.1: Transputer architecture

on any link can occur concurrently with communication on other links and with program execution.

OCCAM itself is based on communicating sequential processors (Hoare [78]) where parallel activities are viewed as black boxes with internal states called processes and which communicate with each other using a one-way channel. Communication is achieved by sending a message down a channel between two processes, one process sends a message and the other reads it from the channel.

As every transputer implements OCCAM, an OCCAM program can be executed on a single transputer or a network of transputers. In the

former case, parallel processes share the processor time and channel communication is simulated by moving data in memory. For a transputer network processes are distributed among transputers and channels allocated to links. Thus, the OCCAM program can be implemented in a variety of ways. One transputer network can be used to minimise cost, another to optimise performance, or a third to balance the two. It follows that an approximation to both the WARP and WAP machines can be made using transputers. We say approximation because the general purpose nature of the transputer must pay some penalty in performance over a dedicated network. In the case of a WAP, OCCAM is particularly useful for implementing wavefronts.

The definition of the transputer architecture divides neatly into logical aspects defining network interconnections and programs, and physical aspects defining how transputers are connected. For systolic algorithms we can liken this to the separation of dataflow and the processor geometry.

3.5.4 Simulation of Systolic Arrays

We use the fact that OCCAM programs can be divorced from, transputer configurations by using the language as a simulation tool throughout the thesis, for testing designs. A brief summary of the OCCAM language is given in the Appendices, together with selected simulation programs. Fig.(3.5.4.1) indicates the general structure of the programs, where branching indicates parallel execution. The construction of programs follows ideas developed by the author in Megson [84] but is modified to take advantage of the formal model developed in Section 3.2.1. Consequently OCCAM programs simulate the

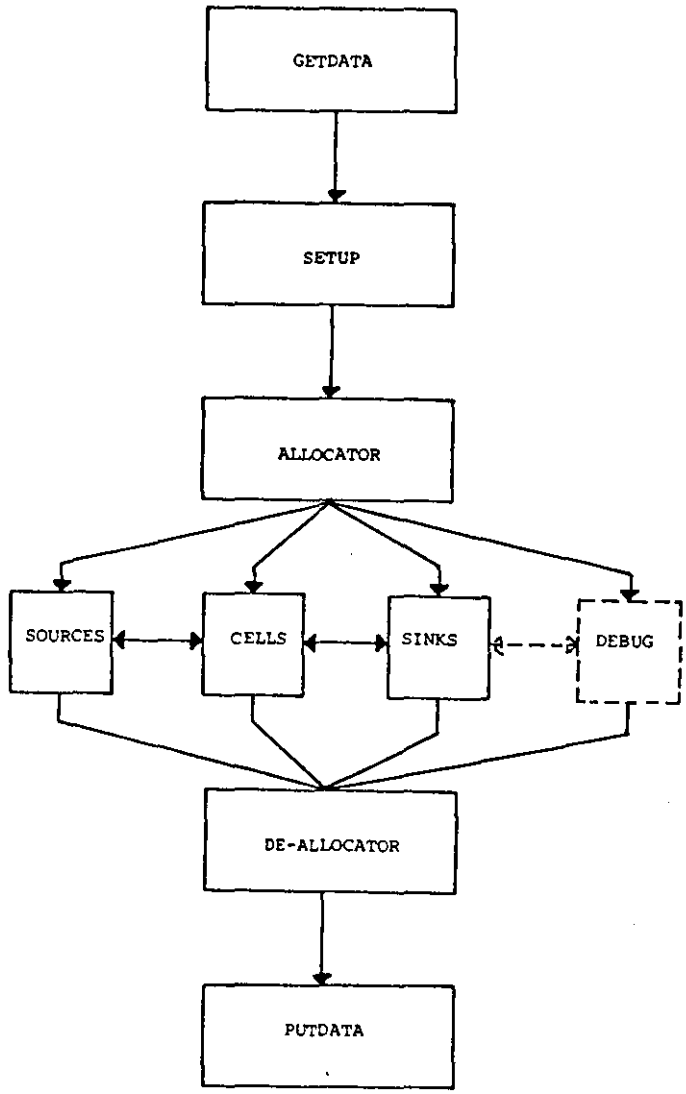


FIGURE 3.5.4.1: Structure of OCCAM program for simulating systolic arrays

the formal proofs by replacing input output descriptions by actual results. Although the simulation does not guarantee correctness it is nevertheless a less time consuming approach which does not result in unsolvable equations. Furthermore, a working OCCAM program retains the possibility of actual transputer implementation and so solves two problems in one attempt.

The getdata and putdata sections of Fig.(3.5.4.1) represent the host machine interface and are responsible for receiving and sending

data and control to and from the program. Each routine contains enough memory to store the initial array input data and the final output data corresponding to the global input and output sequences of the model. The amount of storage is easily calculated by summing the termination functions of the bounded data sequences of each input output sequence. In principle the two routines can be run in parallel with each other and the array, and for some of the examples this is the case, but generally they are sequential, in order to emphasize the parallel operation of the array. The actual host can be predefined input and output files or simply the terminal, the former method is useful for buffering and throughput testing, while the latter helps with debugging and interactive array performance. The routines can be augmented with user friendly features directing the program use, the collection of data necessary for the array construction and formatting of results.

The setup routine is a key section of the algorithm which computes array dependent quantities. For instance, in the matrix vector array, Fig.(3.2.1.3) depending on the size of p and q the x,y and A data streams must be delayed in order for them to synchronize. The setup performs these calculations and in addition computes certain values useful in defining the structure of the array. These latter structural values are more important as the array becomes more complex.

Sources, sinks and cells are OCCAM procedures corresponding to nodes in V_S, V_T and V_I of the network model. A source is loaded initially with a vector from getdata representing its associated bounded data sequence, together with additional values from the setup routine. The values from setup are used to create the shift and spread of a sequence before inputting it to the V_I procedures. Sinks are analogous

to sources except they work in reverse, using shift and spread data to cut out neutral elements and placing real values into data vectors which are then passed to putdata for output. The cell procedures implement the n-ary sequence operators associated with nodes in V_I. Generally there is one procedure for each type of cell, and the programming task is simplified for homogeneous networks. The input and output sequences are represented by OCCAM channels appearing as actual parameters in the procedure headings. Where cell definitions are only marginally different extra switches and flags can be added to a procedure heading so it can set up the correct cell type. This collapses a number of definitions onto a single generic one. Extra parameters can also be used for preloading array values, such as convolution weights. A cell definition is divided into three sections, initialization, communication and computation. Initialization is performed only once and allows cells to be cleared before use or pre-determined values to be set up. In particular, initialization defines neutral element quantities which can be used in communication before real data reaches the cell, and is essential to maintain dataflow in OCCAM programs. The communication and computation sections of the cell are performed many times and are enclosed in a loop for iteration, and are performed sequentially one after the other. All communication is performed in parallel and computation is mainly sequential. The loop control depends on the type of algorithm, if the execution time can be computed in advance by a simple setup calculation a for-loop can be used where the operation is uncertain a while-loop can be used. The while-loop is more flexible because it allows successive problem instances to be pipelined, whereas the for-loop scheme runs an algorithm

a fixed number of times. The two loop schemes also have differences in the way they terminate but to follow this the allocation de-allocation procedures must be discussed.

The Allocator routine is called after setup and is supplied with parameters about the array dimensions, synchronisation details (shift, spread, etc.) and the total number of cycles in the algorithm if a loop scheme is used, and data sequence sizes. The allocator is simply a set of parallel loops which specify and startup the computational graph by connecting V_S,V_T and V_I procedures using OCCAM channels as arcs and allocating channels according to the colour set C_E. To achieve setup the graph is mapped onto a grid of points whose points and hence arcs can be recovered from a simple address type calculation. The simpler the array the easier are the mapping functions, and the result is an allocation similar to the VLSI grid model. Once started the sources and sinks control computation, and the allocator only terminates when all the graph cell procedures have terminated. Termination of procedures is assumed to be globally synchronised if a for-loop is used in cells and asynchronous if while-loops are incorporated. As OCCAM is an asynchronous communication language for-loops tend to be messy requiring some additional computation after the loop to clear all the channels - hence avoiding deadlock. While-loops are better suited to the model of concurrency and when augmented with systolic control sequences can be used to selectively closedown a cells input and output channels. Consequently array cells can be switched off or de-allocated by a wave-front progression or pipelined approach from sources to sinks.

An additional procedure for debugging purposes can be added which runs in parallel with graph network, and is mainly a screen/file mixer

routine. The allocator sets up the procedure and network cells are augmented with an additional channel each which the debug routine uses to analyse cells. Debug channels are allocated from a pool of channels all of the same colour and require an ordering of network cells for correct indexing. When the indexing function is simple debug can be used to output snapshots of array operation so data flow can be easily verified. Snapshots are output in a sequential cell-ordering and the additional debug channel communication must be placed carefully in cell definitions. It should be clear that a globally synchronised scheme can always be simulated by an asynchronous network and placing the debug channel operation as the first statement in a loop acts as a synchronising and ordering mechanism for cell computation rather than computations. Although we cannot force OCCAM procedures into lock step execution a similar idea to the debug scheme can be used to distribute cell termination controls for while-loop schemes when controls are themselves complex. Such a scheme has merit because it lends an intuitive synchronous outline to the algorithm construction even though it is executed asynchronously.

Finally, the techniques described above have been used successfully throughout this thesis to implement designs in OCCAM but can in principle be extended to any parallel language provided channels and cells can be modelled. In fact Brent & H.T. Kung & Luk [83] used an extended version of PASCAL, ADA also seems a likely candidate as ADA rendezvous is very similar to channel communication both being based on CSP. We adopt OCCAM because it offers more direct hardware support for special purpose designs as well as common architectures.

3.6 THE SOFT-SYSTOLIC PARADIGM

To conclude this chapter we shall attempt to unify the systems of the previous sections into a unified framework which we term the soft-systolic paradigm. This new paradigm incorporates the old version allowing the systolic principle to be exploited to the full. In particular, we define three types of systolic algorithm, hard, soft and hybrid which lead to new types of systolic design frames, and which can adapt to changing technology conditions over time to evolve new constrained frames.

Hard systolic algorithms (S_H) are the so-called traditional algorithms discussed in Section 3.2 which form regular frames and also obey additional heuristics of constrained frames. In essence, these algorithms should be designs which can be implemented using current technological practices without programming.

In contrast soft-systolic algorithms (S_S) are the most general algorithms which are not constrained and may even be irregular. Intuitively such algorithms are not suitable for direct hardware implementation and can only be realised in a simulated environment using a programming language executed on some parallel architecture (which may be is a general purpose systolic architecture) see Shapiro [84]. Soft-systolic algorithms map the systolic space onto the memory structure of the machine and array architectures and processor geometries form data structures for controlling the parallelism within the machine. Such algorithms are clearly possible and supported by the OCCAM language and architectures such as the Meiko computing surface.

Hybrid systolic algorithms fill the gap between hard and soft algorithms allowing the marriage of limited programming and hardware to

achieve satisfactory cost/performance relationships for well defined areas of computation. A current contender for hybrid classification is the WARP architecture which is aimed at generic problem solving but will be incorporated into a more general purpose machine.

This framework neatly partitions designs meant for direct implementation and those which are programmed, allowing algorithm designers the choice to develop designs based on a different trend in technology. Consequently systolic algorithms can be considered in a state of flux or migration through soft, hybrid and finally to hard designs as technology develops. At the soft end new systolic frames can be devised together with manipulation rules to enumerate all the possible systolic designs for a particular problem. Closed frames will result in a finite set of designs which can then be considered to produce a few hybrid and hard architectures. Such techniques will be essential for examining connections between systolic and existing algorithms as well as generalising the systolic concept. In particular, the goals of fifth generation computing projects requiring massive parallelism and high performance would greatly benefit from an approach which explicitly incorporates a technology sensitive migration of algorithms.

To illustrate these ideas we can point to two main trends in manufacturing technology which could provide alternative definitions for systolic frames in the near future.

3.6.1 3-D VLSI

Rosenberg [83] presents a case study in 3-D VLSI which is relevant to our purposes. The main concept is to extend the grid model of Section

(3.4.1) so it forms a rectangular solid of grid points and allows vertical wire connects as well as in the plane of each horizontal level of grid points. Each level or laminar also consists of enough layers to run wires as in the original model but only a single wire can occupy a vertical connection. The immediate consequences of such an arrangement are listed below.

1. Wire routing should become easier due to the extra dimension for avoiding crossovers.
2. An overall volume reduction should occur due to reduction in area used to route around processors.
3. Shorter wires will reduce clockskew bounds and cycle time.
4. An increased surface area for input and output pins.

Fig.(3.6.1.1) illustrates a 3-D chip arrangement.

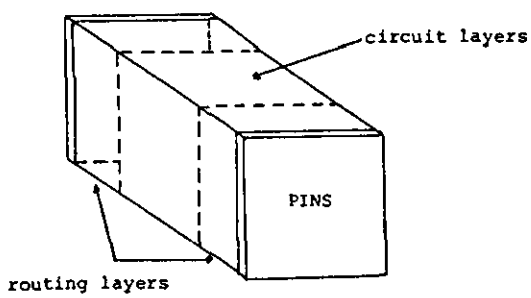


FIGURE 3.6.1.1

To keep production costs low only a finite number of extra laminars should be adopted and schemes based on this model are easily derived. The simplest scheme would be a two-layer design, with one-layer for circuit elements and the other for wire routing. A more complex arrangement would be 3 laminars forming a sandwich with circuit elements on the middle layer with data and control separated onto the remaining

two layers. Advanced techniques would permit more layers with a mixture of circuit element layers and routing planes. Unfortunately such ideas are non-trivial and will require a great deal of time and effort. Current technology is just beginning to introduce two layer schemes in which the second layer is used for metalisation for power and ground lines. Significant problems in future development are:

(i) The alignment of successive layers - this must be precise if all inter-layer connections are to be formed. An additional problem is the alignment and bonding of pins on each layer.

(ii) The sinking of shafts for wires between layers - difficulties arise due to diffraction of x-rays and the non-uniform exposure to solvents used in etching the shaft, and increase with depth.

(iii) Cooling of densely packed chips is already a significant problem circuits layers sandwiched between other layers only have the perimeter of the layer to lose heat.

The two layer design with wires on one layer reduce these problems particularly when we assume that wires give off less heat than circuit elements.

3.6.2 Optical Computing

3-D VLSI avoids clockskew and area problems by stacking circuits on top of one another in the same way sky-scrapers solve space problems in urban areas. Optical computing on the other hand is aimed at replacing the limitations imposed on computing circuits due to electrical properties of materials. In optical computing, electrical signals are

replaced by streams of photons and electrical logic elements replaced by optical counterparts. Introductory material to optical computing can be found in Goodman, Leonberger, S.Y. Kung, Athale [84], Athale & Lee [84] and Huang [84]. Some immediate benefits of this approach are apparent:

(i) Clockskew: is avoided due to communication at the speed of light.

(ii) Crosstalk: the interference of electrical signals by mutual inductance in closely placed wires is reduced due to relative difficulty in making streams of photons interact.

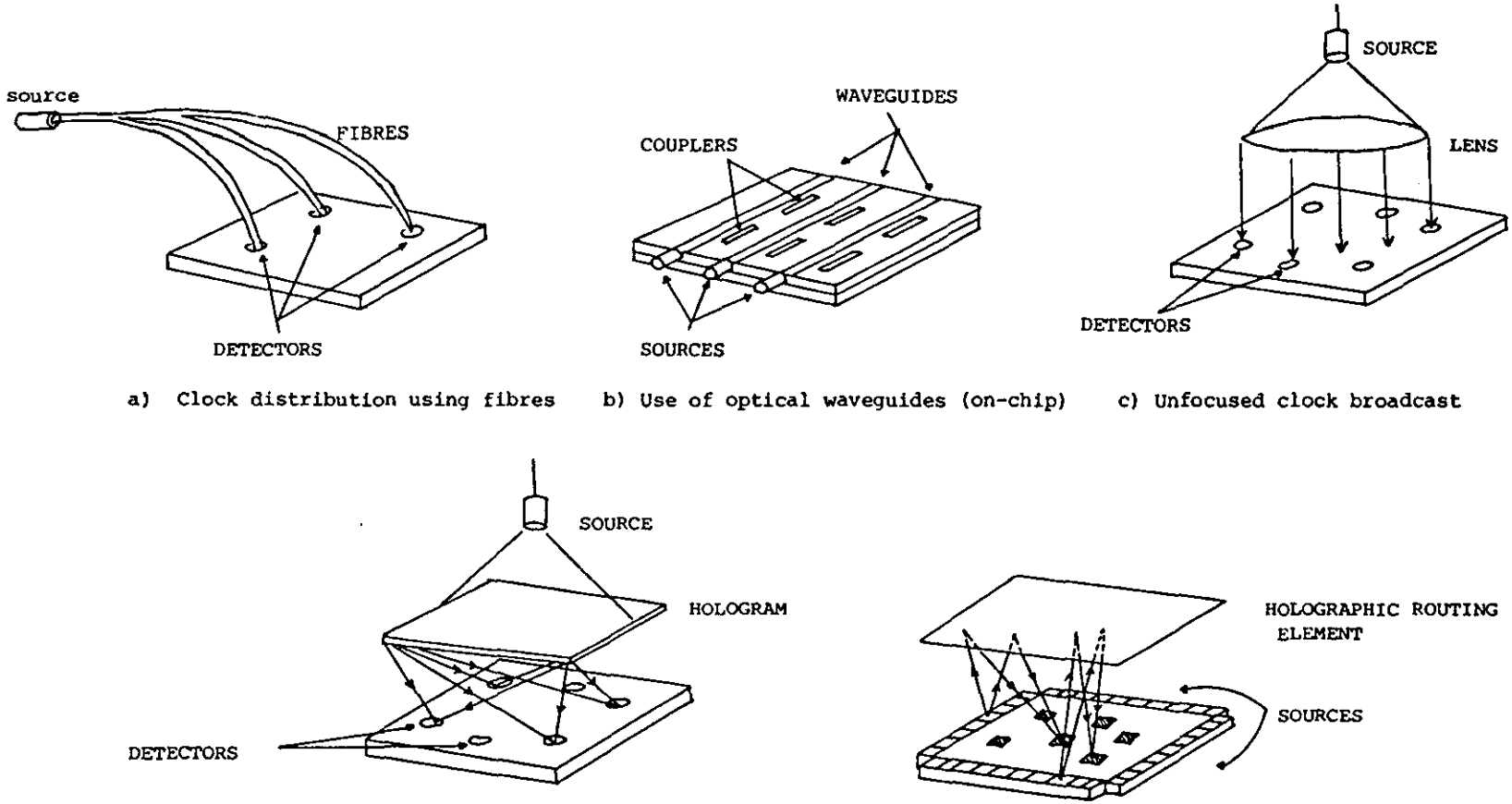
(iii) Radiation hard: Transient errors in computation due to radiation (e.g. cosmic rays) incident on the chip surface is reduced.

(iv) Non-planarity: Circuits can be non-planar as photon streams crossing at an angle of >10% suffer no crosstalk occurance and separate signal lines can intersect, with no ill-effects.

There are three kinds of optical computing which can be defined. Acousto-optical is essentially analogue and has been used in optical signal processing to produce moving data streams for multiplications. The same principle is easily adapted for systolic arrays and is discussed in Caulfield, Rhodes, Foster and Horvitz [81]. However acousto-optics is limited by low accuracy inherent in its analogue representation and flexibility in terms of the types of operations to which it can be applied. Digital optical computing on the other hand is involved with the development of optical logic elements to complement optical signal transmission. In principle, this method will be as

accurate as current digital machines, but problems arise because components so far accept inputs in one form such as intensity and output in another like phase and so cannot be cascaded to form circuits. There are also doubts about the best representation of the binary units '0' and '1'. Some favour pulses while others prefer methods of polarization (see Brenner & Lohmann [86]). It follows that full digital optical computing requires a great deal of time, money and effort to get it working. A third alternative and perhaps the best in the short term is electro-optical computation which combines electronics and optics. Optics take the role of signal transmission, electronics providing well established cascadable logic elements. Optical signals arriving at logic is converted to electrics processed and then converted back to optics for further transmission. On a straight comparison purely electronic components would appear to be faster due to the extra overhead of signal conversion, but as feature sizes shrink and resistance and capacitance of wires conspire to make propagation the key factor in speed the conversion is justified, with optics used mainly for data and control transmission a number of schemes can be envisaged and illustrated in Fig.(3.6.2.1).

1. Waveguided: A waveguided signal can take two forms, first, as optical fibre transmission for connection chips and second a guide (such as glass) integrated onto the chip, for internal chip communication waveguides are restricted by the constraint that they must be kept as straight as possible otherwise radiation and signal loss occur. Consequently, a network of orthogonal guides must be used on chip, and we could use a variant of Fishers' hybrid clocking scheme where light is used for global synchronisation of clocks distributed electrically for subcircuits.



a) Clock distribution using fibres b) Use of optical waveguides (on-chip) c) Unfocused clock broadcast

d) Focused distribution of clock signal e) Data communication in free space

FIGURE 3.6.2.1: Optical signal transmission

2. Free-Space Transmission: In free-space transmission light is not guided but controlled by the laws of propagation of light in free space, and is further subdivided into focussed (or imaging) and unfocussed. In unfocussed schemes optical signals are broadcast to the whole chip (i.e. collineated light illuminates the chip at normal incidence) and is detected by sensors distributed around the chip. The problem with this is its inefficiency as only a small amount of light falls on the required area, and the need for a blocking layer to avoid transient errors in other parts of the circuit. Focussed light avoids these problems but uses a hologram to focus the signal to all the sights required but as with 3-D VLSI it requires a high accuracy of alignment. Finally for data signals inputs to pins can distribute signals into the middle of the chip by reflecting them from a routing holograph.

Alot of development is required for all these techniques but research is progressing into waveguides on chip with some success and fibre optics is already well developed. Furthermore electro-optics by marrying the existing abilities of chip technology with optical developments will provide a transient technology until true digital optics becomes achievable.

3-D VLSI and optics represent two of the most realistic improvements in implementation technology for the near future, their basic philosophy is to remove some of the limitations listed as heuristics for a constrained systolic frame. We can also modify the basic rules of a systolic frame to accommodate the soft-systolic approach. A revised list of rules defines a soft-systolic frame as follows:-

R[1-3] unchanged, these are the essential features of systolic algorithms.

R[4] flow of data should be regular, and complex data movements must be constructed from simple ones in a controlled manner.

R[5] The majority of connections should be nearest neighbour or to local cells but long connections are permitted.

R[6] Unchanged.

The heuristics are modified as follows:-

H[1] The systolic space can be multi-dimensional

H[2] The processor geometry is non-planar in a limited sense, that it can be stratified into a series of planar layers.

H[3] Unchanged.

H[4] Local broadcasting is permitted to cells on the same layer or between adjacent layers.

H[5] Long wires are permitted but should be omitted if possible.

In the following chapters we will examine new algorithms which make use of these properties as well as the original restrictions, emphasis is placed on a limited and controlled expansion of rules of systolic design based on current research trends. As the technology is not yet available our designs are limited to OCCAM testing and so are essentially soft-systolic.

PART II

IMPROVEMENTS TO SYSTOLIC ARRAYS FOR LINEAR ALGEBRA

CHAPTER 4

SOFT-SYSTOLIC PIPELINED MATRIX ALGORITHMS

"... or an Opus of tunes based on canons and fugues"

Definition: *Cǎ'non (Music)*
piece with different parts taking up the same subject sucessively in strict imitation.

Definition: *Fùgue (Music)*
Polyphonic composition in which short melodic theme ('subject') is introduced by one part and sucessively taken up by others and developed by interleaving the parts.

An allegory of systolic computation.

In Hwang & Cheng [82] the idea of partitioning matrices to solve arbitrarily large linear systems using iteration was considered, while Heller [85] considers partitioning large matrices for small systolic arrays. The motivation behind these techniques is the technological constraints imposed by the physical realization of devices. In this chapter we have suggested some new arrays which use the relaxed constraints of a soft-systolic frame and which improve efficiency and possibly computation time. Two main themes are the combination of multi-layer and multi-pass arrays and the use of block partitioning to improve efficiency.

4.1 ADDITIVE SPLITTING AND DOUBLE PIPES

Additive splittings take the form,

$$A = A_1 + A_2 + \ldots A_i + \ldots A_m, \quad i=1(1)m \ , \tag{4.1.1}$$

for an n×n band matrix A with bandwidth w and have been applied successfully to matrix multiplication problems in order to reduce the effective bandwidth of A used in computation. Now, the size of many systolic arrays (for matrix problems) are independent of n but dependent on w; thus reducing w to $\bar{w}$ implies that a smaller array can be employed. For a splitting this reduction is achieved by choosing the A_i to have bandwidths $\bar{w}_i \leq \bar{w}$, i=1(1)m. Hence, the original problem can be solved by solution of the A_i subproblems by repeated use of an array of size $\bar{w}$. As an example, the matrix vector problem,

$$Ax = y \ , \tag{4.1.2}$$

is solved for $\bar{w} \geq 1$ using the splitting form of (4.1.1) by the procedure

$$\begin{aligned} &y = 0 \\ &\text{for } i=1 \text{ to } m \text{ do } y=y+A_i x \ , \end{aligned} \tag{4.1.3}$$

where the inner loop is computed by a matrix vector array of the form Fig.(3.2.1.3) with $\bar{w}$ cells, when $\bar{w}$=1, m=w and the solution of (4.1.2) is achieved by a sequence of single elemental diagonal type matrices, and the array consists of a one cell non-stationary arrangement similar to Fig.(3.2.1.1). Alternatively, when $\bar{w}$=n and m=2 dense matrix product can be computed by applications of lower and upper triangular matrix vector problems. When $\bar{w}$=w and m=1 we have the normal systolic array which is applied only once. Computations using (4.1.3) for m=1 are called single-pass methods and when m>1 multi-pass schemes; in the former case data is passed through the array only once, whereas in the latter case data passes through m times. Consequently multipass computations have the disadvantages of repeatedly pumping x and y data from the host and increased computation time for extra passes. But this is offset by the advantage of fixed sized hardware which can be used to solve problems of arbitrary bandwidth by using a variable number of passes.

Splitting techniques can also be used to avoid redundant computations (e.g. symmetry properties), increase effective band density and improve efficiency. For instance, in the case $\bar{w}$=n, m=2 above with the array of Fig.(3.2.1.3) with $\bar{w}$ cells, upper and lower triangular computations can be interleaved by filling neutral elements occurring in the original input data sequences. Hence, two passes are reduced to only one and array efficiency is increased from e=1/2 to e=1.

The use of additive splittings for multipass computations is optimised when a balance between the number of passes and the time for a single pass is achieved. The number of passes are reduced only by increasing $\bar{w}$, while the duration of a single pass is only affected by

array design. In this section, we propose a splitting which addresses both problems simultaneously. This is achieved not by reducing the bandwidth of the A_i, but producing forms which allow the array to be split into disjoint groups of cells which can be mapped onto a multi-layer layout. As a result more cells can be placed in the same effective area by stacking layers, so increasing $\overline{w}$ and producing decrease in m, a side-effect is improved array efficiency and reduced computation time for a pass.

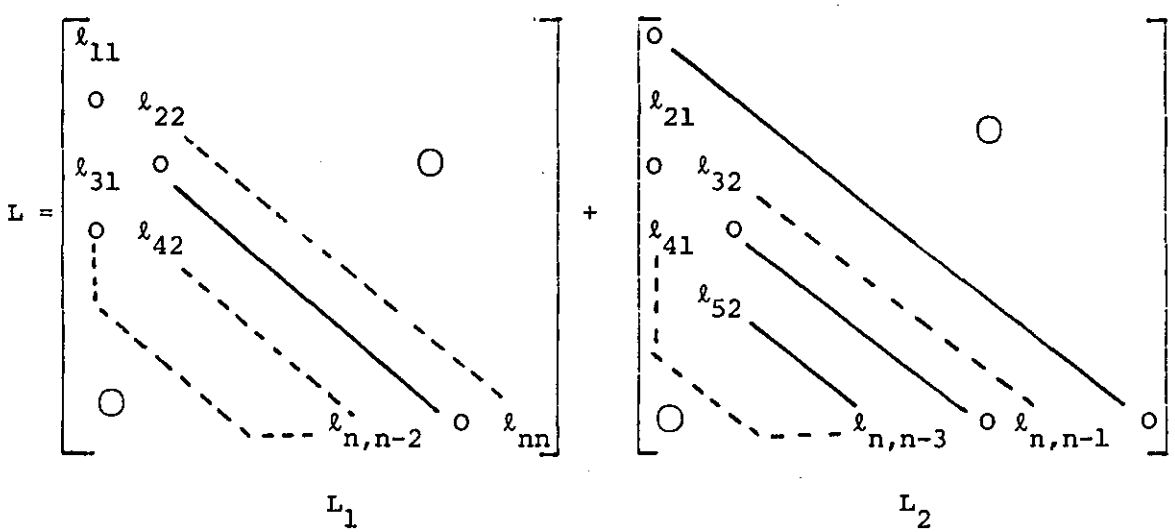
To introduce the splitting, consider the simple problem of multiplying an n×n lower triangular matrix L with bandwidth q with a n×1 vector x, i.e.,

$$Lx = d \ , \tag{4.1.4}$$

which from Theorem (3.2.1.4) requires q ips cells and time T=2n+q. We define an additive splitting with m=2 of the form,

$$L = L_1 + L_2 \ , \tag{4.1.5}$$

such that,



L_1 contains $\lceil q/2 \rceil$ subdiagonals and L_2, $q-\lceil q/2 \rceil$ subdiagonals. Hence when q is even L_1 and L_2 contain equal numbers of subdiagonals and when q is odd, L_1 has one more subdiagonal than L_2. Substituting for L in (4.1.4)

produces two subproblems of bandwidth q containing some sparse diagonals, written as follows with y_1 and y_2 auxiliary result vectors,

$$\text{a) } L_1x = y_1 \qquad \text{b) } L_2x = y_2 \qquad \text{c) } d = y_1+y_2 \tag{4.1.6}$$

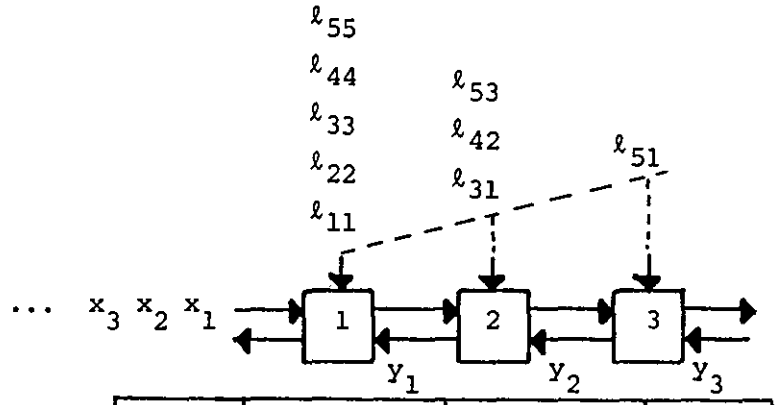
This problem can be solved by two passes using (4.1.3) or 1 pass using interleaving which produces the original array timing. A superior array is derived by noticing that L_1 requires only the odd subdiagonals and L_2 the even ones, so that each solution of y_1 and y_2 needs to collect only $\lceil q/2 \rceil$ terms at the most. Consequently the data flow can be re-timed to remove all the neutral synchronising elements and (4.1.6a) and (4.1.6b) can be solved on arrays with $\lceil q/2 \rceil$ and $q-\lceil q/2 \rceil$ cells respectively with times bounded by $T=n+\lceil q/2 \rceil$ as illustrated in Fig.(4.1.1) for q=5. The time for a single pass is then reduced by solving (4.1.4) using two arrays computing in parallel arranged in a double pipe format shown in Fig. (4.1.2a). As the timing of the whole array is bounded by the cost of (4.1.6a) we have proved the following theorem.

Theorem 4.1.1: An n×n lower triangular matrix vector multiplication problem of bandwidth q can be solved using a double pipe in time $T=n+\lceil q/2 \rceil+1$ using q ips cells, an adder and one delay cell.

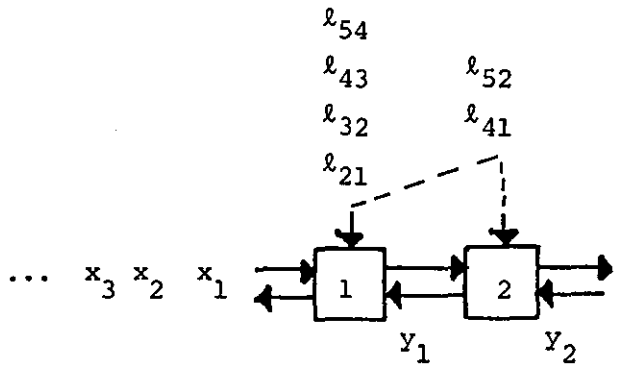
A simple corollary follows immediately and is stated for completeness:-

Corollary 4.1.1: A n×n upper triangular matrix vector problem with bandwidth p is solved by a double pipe in $T=n+\lceil p/2 \rceil+1$ using p ips an adder + delay cells.

The extension to banded matrix vector multiplication follows naturally and is illustrated in Fig.(4.1.2b) with the corresponding theorem.



t	1	2	3
1	$\ell_{11}x_1$	δ	δ
2	$\ell_{22}x_2$	$\ell_{31}x_1$	δ
3	$\ell_{33}x_3+\ell_{31}x_1$	$\ell_{42}x_2$	$\ell_{51}x_1$
4	$\ell_{42}x_2+\ell_{44}x_4$	$\ell_{53}x_3+\ell_{51}x_1$	-
5	$\ell_{53}x_3+\ell_{51}x_1$ $+\ell_{55}x_5$	-	-



t	1	2
1	$\ell_{21}x_1$	δ
2	$\ell_{32}x_2$	$\ell_{41}x_1$
3	$\ell_{43}x_3+\ell_{41}x_1$	$\ell_{52}x_2$
4	$\ell_{54}x_4+\ell_{52}x_2$	$\ell_{63}x_3$
5	$\ell_{65}x_5+\ell_{63}x_3$	-

FIGURE 4.1.1: Double pipe dataflow

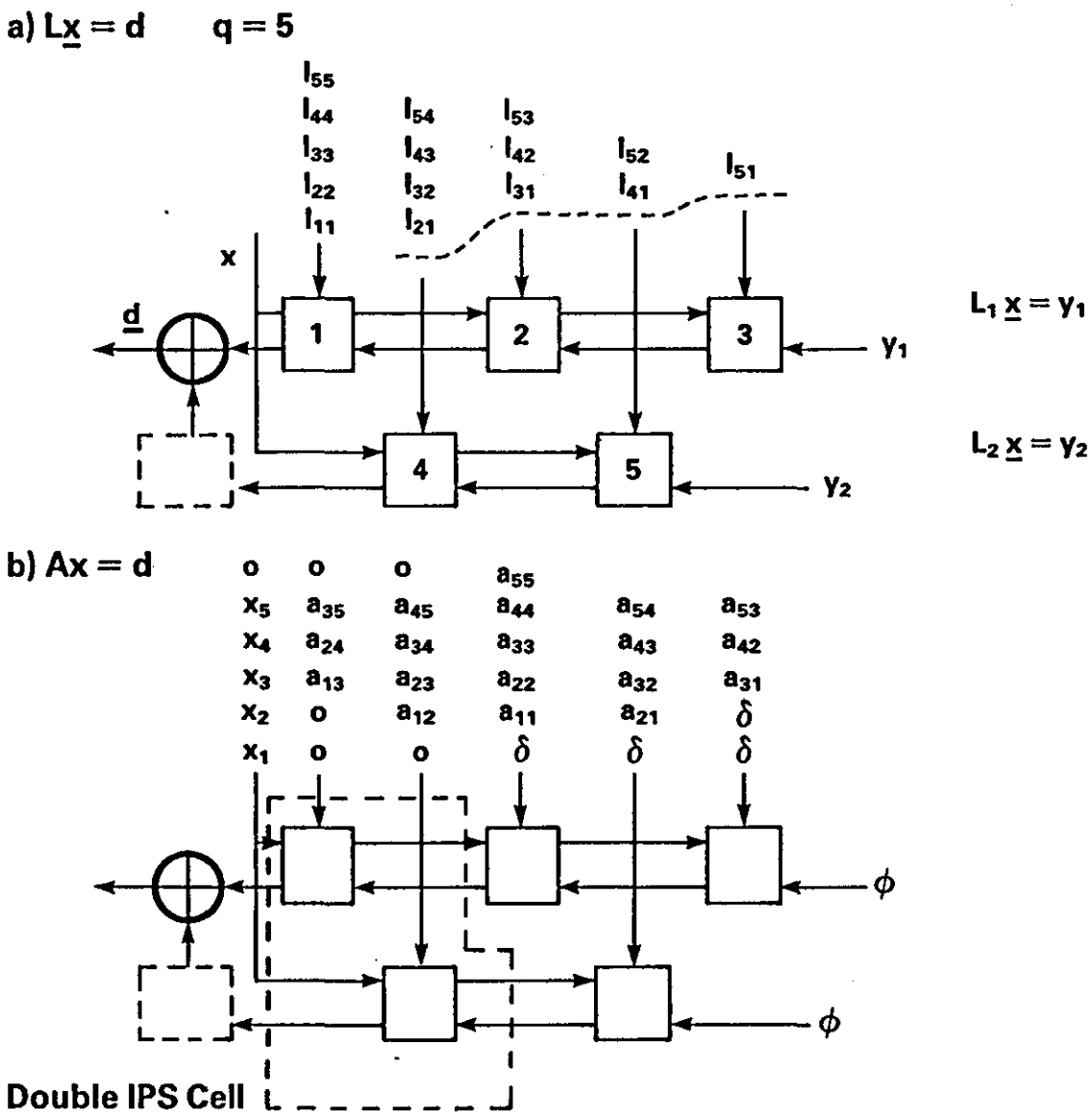


FIGURE 4.1.2: Double pipe arrays

Theorem 4.1.2: The multiplication of an n×n band matrix A with bandwidth w=p+q-1 can be solved on a double pipe with $T=n+\lceil w/2 \rceil+1$ using w ips cells, one adder and a delay cell.

Proof: [by construction of the array].

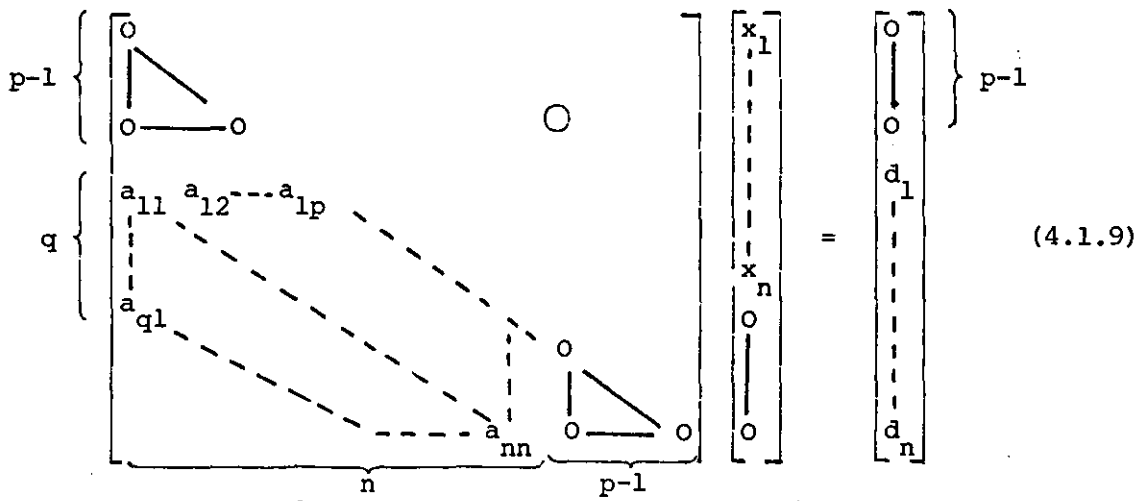
The band matrix problem,

$$Ax = d \ , \tag{4.1.7}$$

can be rewritten as a lower triangular problem,

$$L\bar{x} = \bar{d} \ , \tag{4.1.8}$$

where L is an (n+p-1)*(n+p-1) lower triangular matrix of bandwidth w and $\bar{x},\bar{d}$ are n+p-1 component vectors. Applying Theorem (4.1.1) gives $T=(n+p-1)+\lceil w/2 \rceil+1$ and requires w ips cells an adder and a delay cell. Now (4.1.8) has the form,



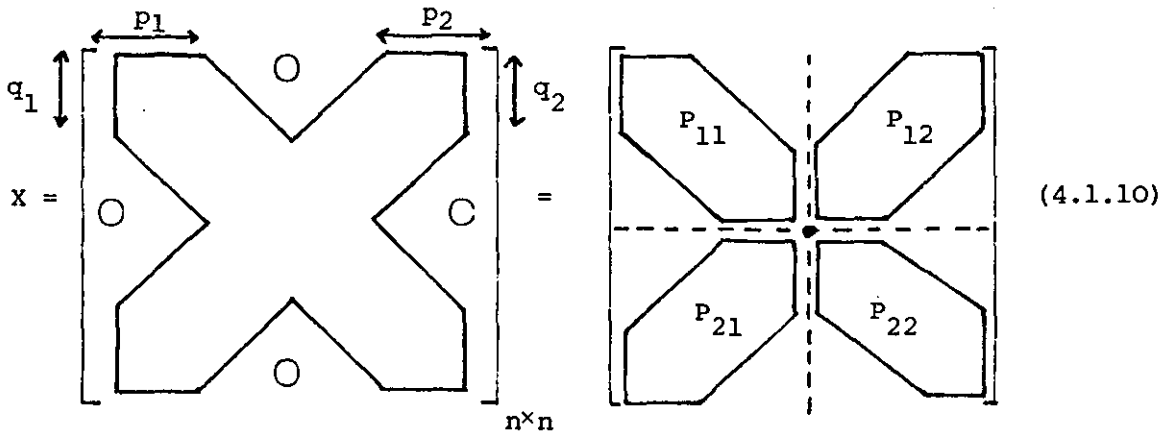
(4.1.9)

L has its first p-1 rows zero and the splitting (4.1.5) produces two similar triangles of size (p-1)/2 as the leading inputs to the component arrays of the double pipe which produce only zero results. By virtue of the fact that $\lceil w/2 \rceil > (p-1)/2$ these preliminary values can be overlapped with the initialisation of the array absorbing the additional (p-1) cycles and producing the exact theorem timing.

Comparing the double pipe arrays with their corresponding

traditional arrays indicates that double pipes are twice as fast and twice as efficient with a negligible hardware overhead of one adder and delay register. The double pipe can be represented by a two-layer design with one pipe in each layer and x values broadcast between layers. All the connections are in this sense regular and each layer contains a planar layout, consequently the double pipe is a design from a soft-systolic frame (an OCCAM program is given in the Appendix). Each layer contains at most half the cells of the traditional 1 layer design, so we have produced a design with half the effective area or alternatively we can double $\overline{w}$ to use the same area. The number of matrices in an additive splitting like (4.1.1) is then reduced and pass time is smaller because we apply the larger double pipe as the inner loop of (4.1.3). An additional property is that a 1-layer design as depicted by Fig.(4.1.2b) could be made using waveguides if electro-optical implementation was considered. No cross talk would occur because all intersecting lines are orthogonal.

Let us now consider a more difficult matrix-vector problem consisting of a matrix banded about its diagonal and anti-diagonal. For reference purposes, the matrix will be termed the X-band matrix which reflects its structure, indicated below,



(4.1.10)

where $w_1=p_1+q_1-1$ and $w_2=p_2+q_2-1$ are the forward and backward bandwidths respectively. Many problems give a matrix this structure if we are willing to allow some sparseness to be included in the bands. For instance constant quindiagonal Toeplitz linear systems derived from non-linear partial differential equations under Dirichlet or periodic boundary conditions (Evans,[80a]) and symmetric linear systems in initial and boundary value problems containing fourth order parabolic and elliptic partial differential equations again under periodic and specified conditions (Evans [83a]). To solve the X-band matrix-vector problem a single pass systolic array must have w=2n-1 and T=4n-1. Furthermore, the selection of a suitable splitting for a multi-pass scheme is more difficult as some diagonals will possess a proportion of zero and hence redundant elements. The task is simplified if some of the sub(super) diagonals crossing p_{21} and p_{12} in (4.1.10) contain only zeroes (as is the case in the examples mentioned above). In the single pass case, the zero diagonals can be used to replace true cells by delay cells as in Fig.(3.2.3.2), but do not affect computation time. For multi-pass schemes zero diagonals reduce the number of matrices (in a splitting) reducing the number of passes. However by applying the double pipe splitting in conjunction with a simple partitioning for which an efficient solution to the X-band is easily derived.

Without loss of generality assume n is divisible by 2 then X can be partitioned into four $m\times m$ submatrices as indicated by (4.1.10) with m=n/2. The matrix vector problem,

$$Xu = d \ , \qquad (4.1.11)$$

is then given by,

$$\left.\begin{array}{ll} \text{a)} & (p_{11}+p_{12})u = d^{(1)} \\ \text{b)} & (p_{21}+p_{22})u = d^{(2)} \end{array}\right\} \qquad (4.1.12)$$

with $d^t=(d^{(1)},d^{(2)})$ and $d^{(1)}=(d_1,\ldots,d_{n/2})$, $d^{(2)}=(d_{\frac{n}{2}+1},\ldots,d_n)$ or,

$$\left.\begin{array}{ll} \text{a)} & p_{11}u^{(1)} = c^{(1)} \\ \text{b)} & p_{12}u^{(2)} = c^{(2)} \\ \text{c)} & p_{21}u^{(1)} = c^{(3)} \\ \text{d)} & p_{22}u^{(2)} = c^{(4)} \\ \text{and} & \\ \text{e)} & d^{(1)} = c^{(1)} + c^{(2)} \\ \text{f)} & d^{(2)} = c^{(3)} + c^{(4)} \quad , \end{array}\right\} \qquad (4.1.13)$$

with $u^{(1)}=(u_1,\ldots,u_{n/2})^t$, $u^{(2)}=(u_{\frac{n}{2}+1},\ldots,u_n)^t$.

Each subproblem (4.1.13) a-d corresponds to a simple matrix vector problem of bandwidth w_1 or w_2, and the simple permutation of elements,

$$\left.\begin{array}{l} \bar{p}_{11}(i,j) = p_{11}(i,j) \\ \bar{p}_{12}(i,j) = p_{12}(i,n-j+1) \\ \bar{p}_{21}(i,j) = p_{21}(n-i+1,j) \\ \bar{p}_{22}(i,j) = p_{22}(n-i+1,n-j+1) \end{array}\right\} \text{for } i=1(1)m,\ j=1(1)m \qquad (4.1.14)$$

$$\text{with} \quad \left.\begin{array}{l} u^{(1)}(i) = u(i), \quad \tilde{u}^{(2)}(i) = u(n-i+1) \\ d^{(1)}(i) = d(i), \quad \tilde{d}^{(2)}(i) = d(n-i+1) \end{array}\right\} \quad i=1(1)m \qquad (4.1.15)$$

where $\tilde{a}$ denotes a vector reversed in order. This allows the X-band problem to be solved using four passes on a traditional or double pipe array with $\bar{w}=\max(w_1,w_2)$ ips cells. More importantly, the usual splitting techniques can be applied to each permuted submatrix problem to introduce more passes and reduce hardware. As a result, choosing $\bar{w}=\min(w_1,w_2)$ requires the splitting of only $\bar{p}_{11}$ and $\bar{p}_{22}$ or $\bar{p}_{21}$ and $\bar{p}_{12}$ depending on which have the largest bandwidth, thus

minimising the number of additional passes introduced. For instance if $\bar{w}=w_2$ we split $\bar{\bar{p}}_{11}$ and $\bar{\bar{p}}_{22}$ to give,

$$\bar{\bar{p}}_{11} = \bar{\bar{p}}_{11}^{(1)} + \ldots + \bar{\bar{p}}_{11}^{(m)}$$

and $$\bar{\bar{p}}_{22} = \bar{\bar{p}}_{22}^{(1)} + \ldots + \bar{\bar{p}}_{22}^{(m)} \quad , \text{ with bandwidths } <w_2.$$

Then solve according to the multi-pass procedure,

$$\left.\begin{array}{l} y = 0 \\ \text{FOR } i=1 \text{ TO } m \text{ DO } \{y=y+\bar{\bar{p}}_{11}^{(i)} u^{(1)}\} \\ d_1^{(1)} = y+\bar{\bar{p}}_{12}\tilde{u}^{(2)} \\ \tilde{y} = 0 \\ \text{FOR } i=1 \text{ TO } m \text{ DO } \{\tilde{y}=\tilde{y}+\bar{\bar{p}}_{22}^{(i)}\tilde{u}^{(2)}\} \\ \tilde{d}^{(2)} = \tilde{y}+\bar{\bar{p}}_{21} u^{(1)} \quad . \end{array}\right\} \qquad (4.1.16)$$

An alternative approach which utilises a multi-layer approach to the full is shown in Fig.(4.1.3). This design is a double-double or D^2-pipe and can be separated out onto two or four layers depending on implementation details. If we assume $w=w_1=w_2$ then a straight forward 3-D approach produces a four layer design with a maximum of $2\lceil w/2 \rceil$ ips cells on each layer and u broadcast to all the layers. If waveguides and electro-optics are adopted to make a double pipe planar, the design reduces to two layers with 2w ips cells on each and the local broadcasting of u.

Theorem 4.1.3: The matrix vector multiplication problem for an n×n X-band matrix with forward and reverse bandwidth w_1 and w_2 respectively can be computed in a time $T=\frac{n}{2}+\frac{w}{2}+2$ on a D^2-pipe requiring at most 4w ips cells six adders and four delay cells (where $w=\max(w_1,w_2)$).

Proof: Trace the array operation.

Assuming all pipes have w cells simplifies synchronisation, but if $w_1 \ll w_2$ or vice versa a large number of cells could be wasted. These

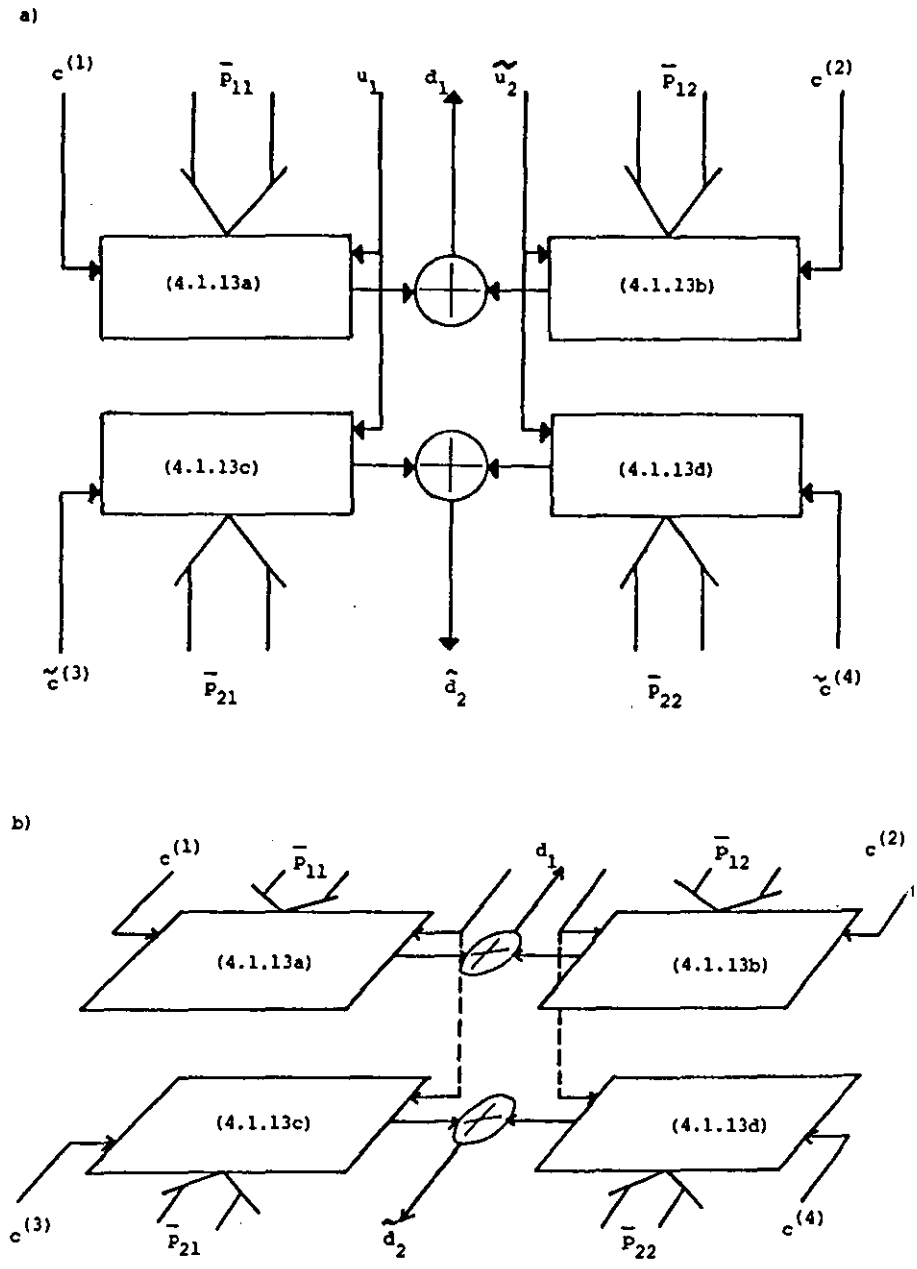


FIGURE 4.1.3: D^2-pipe arrangement a) planar b) multi-layer layout.

redundant cells can be exchanged for delay cells to retain synchronisation if necessary but a typical periodic matrix will often satisfy $w_1=w_2+2$ and the main concern is not the redundant cells but utilisation of cells for the sparse anti-diagonal band.

The D^2-pipe design can also be used for multi-pass computation, allowing the number of cells to be fixed at $4\bar{w}$ for $\bar{w}<w$. In this scheme each of the subproblems of (4.1.13) are themselves split and in permuted form for array input produce,

$$\bar{P}_{11} = \bar{P}_{11}^{(1)} + \ldots + \bar{P}_{11}^{(m_1)} \quad , \quad \bar{P}_{12} = \bar{P}_{12}^{(1)} + \ldots + \bar{P}_{12}^{(m_2)}$$
$$\bar{P}_{22} = P_{22}^{(1)} + \ldots + P_{22}^{(m_3)} \quad , \quad \bar{P}_{21} = \bar{P}_{21}^{(1)} + \ldots + P_{21}^{(m_4)}$$

and if $m=\max(m_1,m_2,m_3,m_4)$ the splitting with $m_i<m$ are padded with extra null matrices to produce the multi-pass procedure,

$$\left.\begin{array}{l} d^{(1)} = 0 \\ \tilde{d}^{(2)} = 0 \\ \text{FOR } i=1 \text{ TO } m \text{ DO} \\ \quad \{ \\ \quad\quad d^{(1)} = d^{(1)} + \bar{p}_{11}^{(i)} u^{(1)} + \bar{p}_{12}^{(i)} \tilde{u}^{(2)} \\ \quad\quad \tilde{d}^{(2)} = \tilde{d}^{(2)} + \bar{p}_{21}^{(i)} u^{(1)} + \bar{p}_{22}^{(i)} \tilde{u}^{(2)} \\ \quad \} \end{array}\right\} \qquad (4.1.17)$$

with Fig.(4.1.3) corresponding to the inner loop allowing simultaneous processing of the four splittings, and the total number of passes required to solve a general X-band problem of (4.1.11) is reduced.

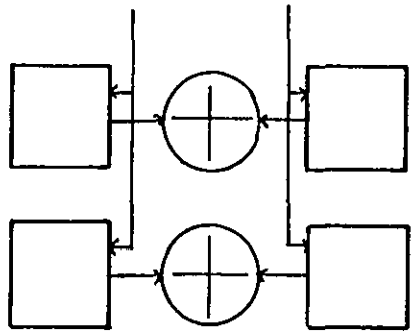
A natural question which arises from these experiments is, "how far can we decompose a linear systolic array into parallel pipes?" The answer is a complex one depending upon the number of layers and additional cells we are willing to admit. The use of multi-layers is an argument based on the principle that a large design can be folded

(or stuffed) into the same effective area. Consequently, a birds-eye view of the design reveals only the area of the top layer with additional cells hidden on subsequent levels. From an implementation viewpoint additional layers can be added so that the effective area of each layer is less than the original design, this improves yield for a single layer. But now we have to be able to make and stack a series of these layers, the effect on yield can only be assessed with any accuracy from practical experience which is not yet attainable. Intuitively a small number of layers would limit the opportunities for further flaws. From a theoretical standpoint the basic principle of partitioning in (4.1.10) can be applied recursively to each submatrix to develop arbitrary depths of parallel pipes. Different types of pipe can be related using the D^n-pipe expression, with D=2, D^0,D^1 and D^2-pipes being the original, double and double-double pipes respectively. The next level of recursion after D^2-pipes produces the partitions below creating $\frac{n}{4}\times\frac{n}{4}$ submatrices,

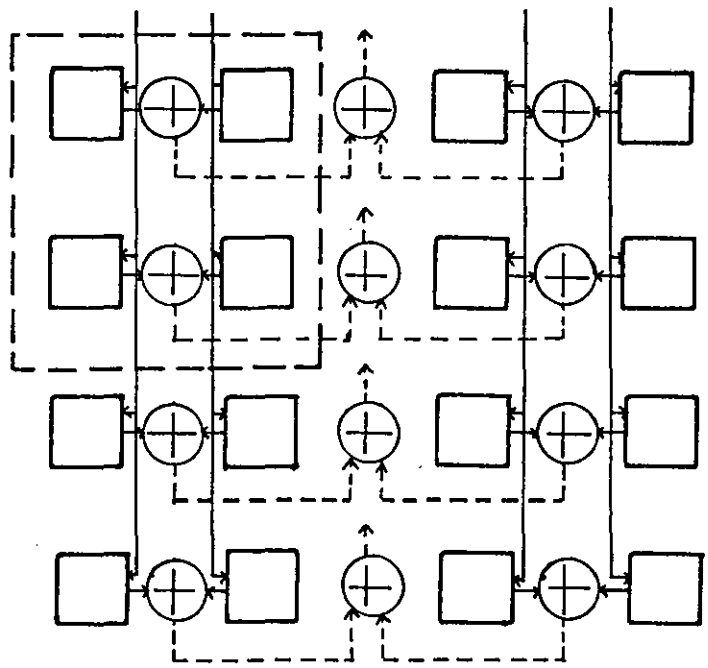
$$\left[\begin{array}{c|c} P_{11} & P_{12} \\ \hline P_{21} & P_{22} \end{array}\right]_{n\times n} = \left[\begin{array}{c|c|c|c} P_{11}^{(1)} & P_{12}^{(1)} & P_{11}^{(2)} & P_{12}^{(2)} \\ \hline P_{21}^{(1)} & P_{22}^{(1)} & P_{21}^{(2)} & P_{22}^{(2)} \\ \hline P_{11}^{(3)} & P_{12}^{(3)} & P_{11}^{(4)} & P_{12}^{(4)} \\ \hline P_{21}^{(3)} & P_{22}^{(3)} & P_{21}^{(4)} & P_{22}^{(4)} \end{array}\right]_{n\times n} \qquad (4.1.18)$$

Recursion stops when 1×1 submatrices occur and there can be at most $\lceil \log_2 n \rceil$ recursion levels. For a dense matrix the starting bandwidth is $w_0=2n-1$, and the bandwidth of subsequent submatrices are $w_1=\lfloor w_0/2 \rfloor$, $w_2=\lfloor w_1/2 \rfloor$ and generally $w_{i+1}=\lfloor w_i/2 \rfloor$. The array for recursion level i+1 is constructed from the array of level i by replacing each D^1-pipe in the D^2-pipes corresponding to each submatrix of level i, by D^2-pipes

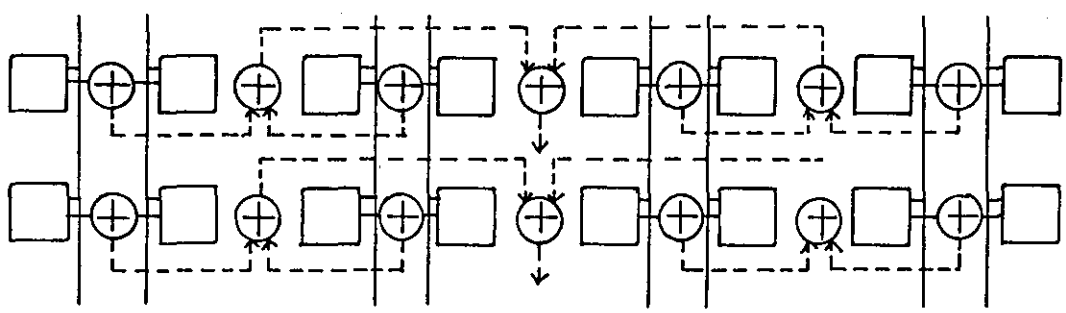
for the submatrices of level i+1, and inserting extra adders to filter the results out of the network. Each D^1-pipe of the substituted D^2-pipe contains w_{i+1} ips cells, 1 adder and a delay. Consequently when 2×2 blocks are reached there are $(n/2)^2$ submatrices and corresponding D^2-pipes consisting of D^1-pipes with 3 ips cells, 1 adder and delay, because 4×4 blocks have bandwidth w=7. Thus, 1×1 submatrices have D^1-pipes with 2 ips cells implying D^0-pipes of 1 cell allocated to separate levels. The technique is illustrated by Fig.(4.1.4) from which it is clear that at the end of recursion an n-layer design results such that one component of the right hand side in (4.1.2) with A, a general matrix, output on each layer. There are n ips cells on each layer which form the leaves of a binary fanin adder tree of height $h=\lceil \log_2 n \rceil$ and containing 2^h-1 adders. An area efficient design is produced by using H-tree layouts for each layer and ips cells can be arranged so that the same x_i, i=1(1)n component is broadcast to leaf cells aligned in the same column through all the layers. The leaf cell of layer i also require the elements of row i of A which present a significant communication problem. To construct a systolic solution a 3-D structure like Fig.(3.6.1.1) (without routing layers), is required. Each H-tree level is rotated to form a column and data communication is achieved by pumping the vector x horizontally in one direction and the rows of A in the other, such that a complete row or the whole of x enter all the tree leaves of a column every cycle. Wherever a row meets x the column tree outputs their vector product $\lceil \log_2 n \rceil$ cycles later, and for all the rows to meet x they must be input every alternate cycle. The array operation is indicated in Fig.(4.1.5)and the timing is clearly



a) D^2-pipe

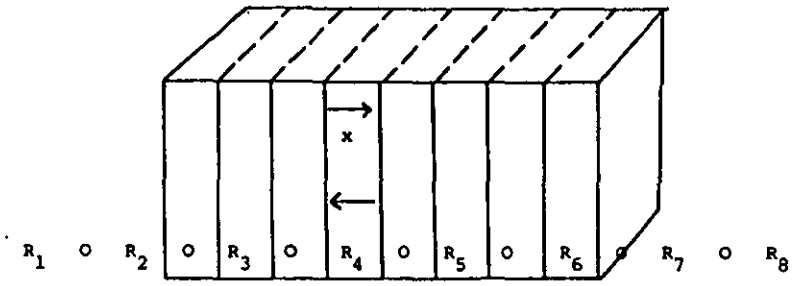


b) Expansion and additional adders

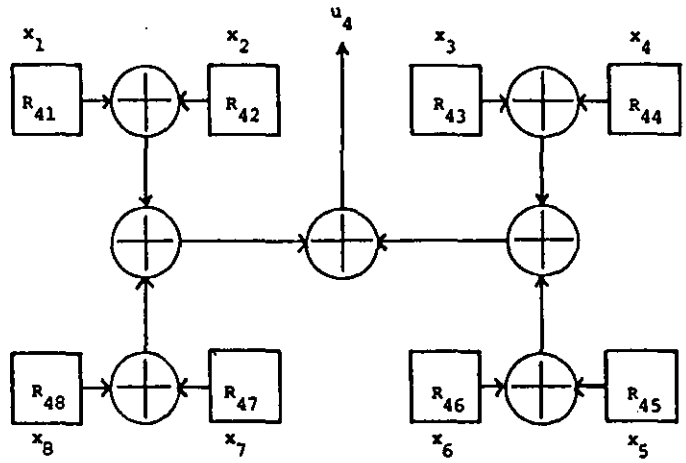


c) Second level expansion of top row of b)

FIGURE 4.1.4: D^2-pipe expansion in recursive pipe application



a) Column layout and dataflow of (Ax=d) for n=8



b) Structure of hex-tree column with input (for column 4 above)

c) Active processors on fanin of result

FIGURE 4.1.5: Full tree organisation of recursive method

$T=2n+\lceil \log_2 \bar{n} \rceil$, with 2n cycles for all the rows of A to be input and $\lceil \log_2 \bar{n} \rceil$ for the last result to be output. Notice, that each columns' leaf cells are active on only one cycle, and that results on the fanin tree can be pipelined. Thus, the n-column design can be reduced to a single H-tree requiring n ips cells and n-1 adders to form dense matrix vector product in $T=n+\lceil \log_2 \bar{n} \rceil$ cycles.

REMARK: notice that the fanin tree must be a complete binary one to accumulate results correctly, consequently when n is odd, A must be padded with a zero row and column.

This latter systolic tree technique for computing linear recurrences is well known, and indicates that retiming and re-placement procedures can be used to derive arbitrary sequentialized and D^i-pipe schemes for matrix vector problems. Thus, the original D^0-pipe systolic array is shown to be a special case of the general tree method and adopting multipass schemes for D^1 and D^2-pipes with $\bar{w}=1$ produces trees of height h=1 and h=2 respectively.

The double pipe splitting extends in a straightforward manner to matrix product computations. In this case the general additive splitting is written as,

$$C = AB = \sum_{i=1}^{m} A_i B_j \ , \quad j=1(1)m \ , \tag{4.1.19}$$

for two n×n band matrices A and B of bandwidths w_1 and w_2 respectively. The double pipe splitting has m=2 and,

$$AB = (A_1+A_2)(B_1+B_2) \ , \tag{4.1.20}$$

where A_i and B_j for n=6 have the forms,

$$A_1 = \begin{bmatrix} a_{11} & 0 & a_{13} & 0 & a_{15} & 0 \\ 0 & a_{22} & 0 & a_{24} & 0 & a_{26} \\ a_{31} & 0 & a_{33} & 0 & a_{35} & 0 \\ 0 & a_{42} & 0 & a_{44} & 0 & a_{46} \\ a_{51} & 0 & a_{53} & 0 & a_{55} & 0 \\ 0 & a_{62} & 0 & a_{64} & 0 & a_{66} \end{bmatrix} \quad A_2 = \begin{bmatrix} 0 & a_{12} & 0 & a_{14} & 0 & a_{16} \\ a_{21} & 0 & a_{23} & 0 & a_{25} & 0 \\ 0 & a_{32} & 0 & a_{34} & 0 & a_{36} \\ a_{41} & 0 & a_{43} & 0 & a_{45} & 0 \\ 0 & a_{52} & 0 & a_{54} & 0 & a_{56} \\ a_{61} & 0 & a_{63} & 0 & a_{65} & 0 \end{bmatrix} \qquad (4.1.21)$$

and

$$B_1 = \begin{bmatrix} b_{11} & 0 & b_{13} & 0 & b_{15} & 0 \\ 0 & b_{22} & 0 & b_{24} & 0 & b_{26} \\ b_{31} & 0 & b_{33} & 0 & b_{35} & 0 \\ 0 & b_{42} & 0 & b_{44} & 0 & b_{46} \\ b_{51} & 0 & b_{53} & 0 & b_{55} & 0 \\ 0 & b_{62} & 0 & b_{64} & 0 & b_{66} \end{bmatrix} \quad B_2 = \begin{bmatrix} 0 & b_{12} & 0 & b_{14} & 0 & b_{16} \\ b_{21} & 0 & b_{23} & 0 & b_{25} & 0 \\ 0 & b_{32} & 0 & b_{34} & 0 & b_{36} \\ b_{41} & 0 & b_{43} & 0 & b_{45} & 0 \\ 0 & b_{52} & 0 & b_{54} & 0 & b_{56} \\ b_{61} & 0 & b_{63} & 0 & b_{65} & 0 \end{bmatrix} \qquad (4.1.22)$$

and

$$C = C_1 + C_2 + C_3 + C_4 \quad ,$$

where,

$$\left.\begin{array}{ll} \text{a)} \quad C_1 = A_1 B_1 & \text{b)} \quad C_2 = A_1 B_2 \\ \text{c)} \quad C_3 = A_2 B_1 & \text{d)} \quad C_4 = A_2 B_2 \; . \end{array}\right\} \qquad (4.1.23)$$

Now consider that the result of product (4.1.23a) has the form,

$$C_1 = \begin{bmatrix} c_{11} & 0 & c_{13} & 0 & c_{15} & 0 \\ 0 & c_{22} & 0 & c_{24} & 0 & c_{26} \\ c_{31} & 0 & c_{33} & 0 & c_{35} & 0 \\ 0 & c_{42} & 0 & c_{44} & 0 & c_{46} \\ c_{51} & 0 & c_{53} & 0 & c_{55} & 0 \\ 0 & c_{62} & 0 & c_{64} & 0 & c_{66} \end{bmatrix} \quad , \qquad (4.1.24)$$

and taking advantage of the zero patterns of A_1, B_1 and C_1 produces a special hex array with re-timed dataflow as illustrated in Fig. (4.1.6),

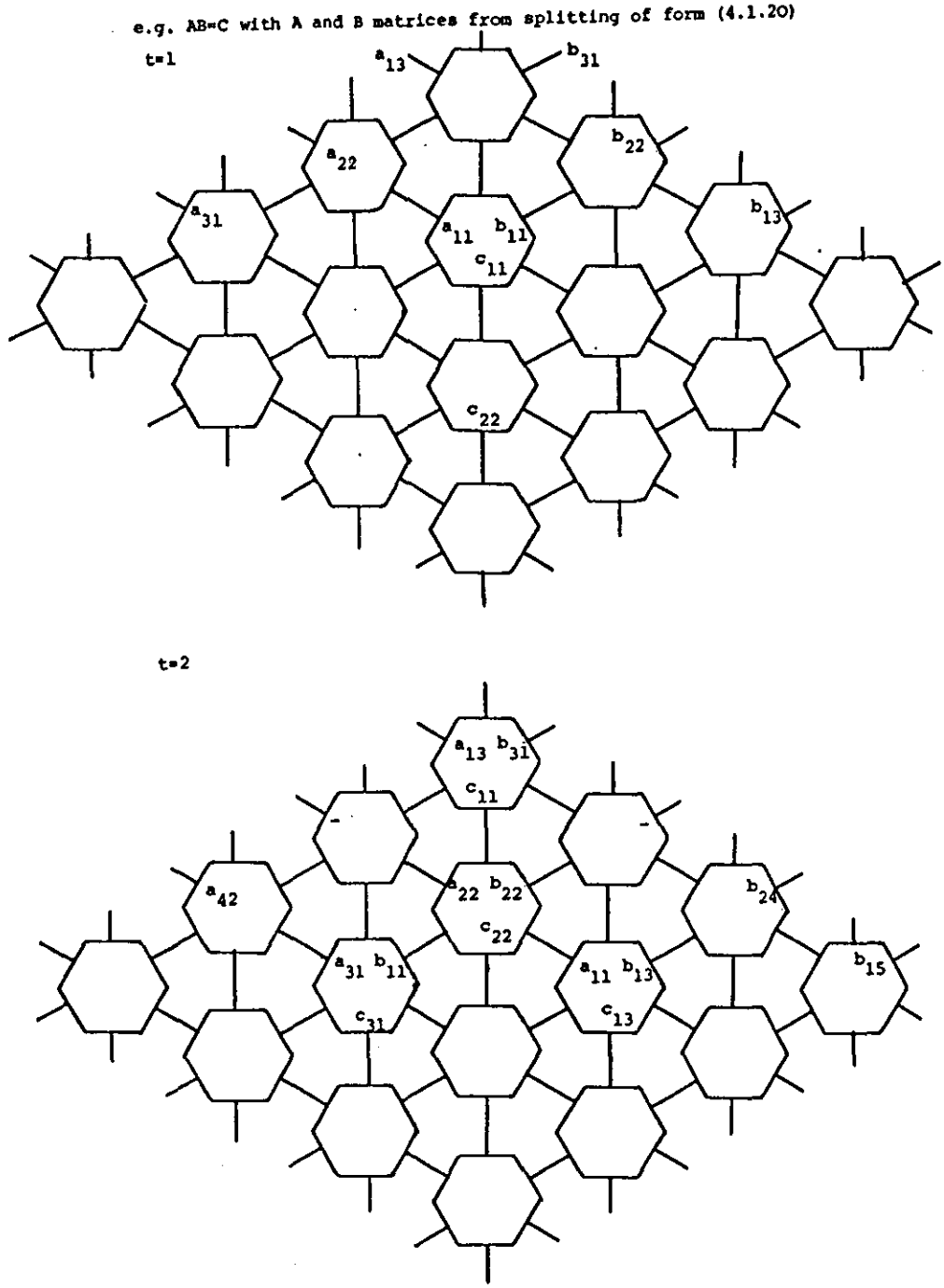


FIGURE 4.1.6: Double pipe hex single layer, standard form matrix product

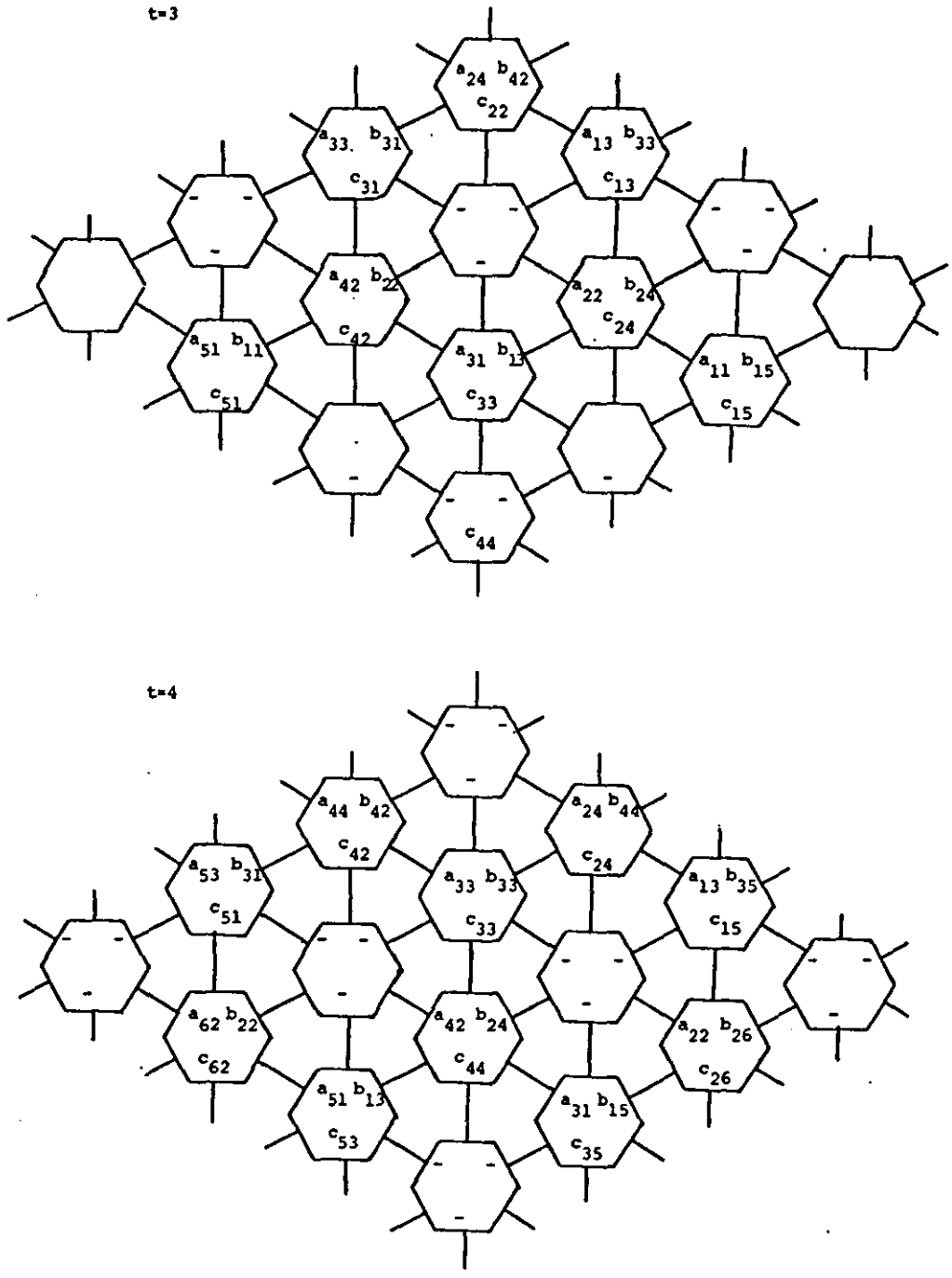


FIGURE 4.1.6: Cont.

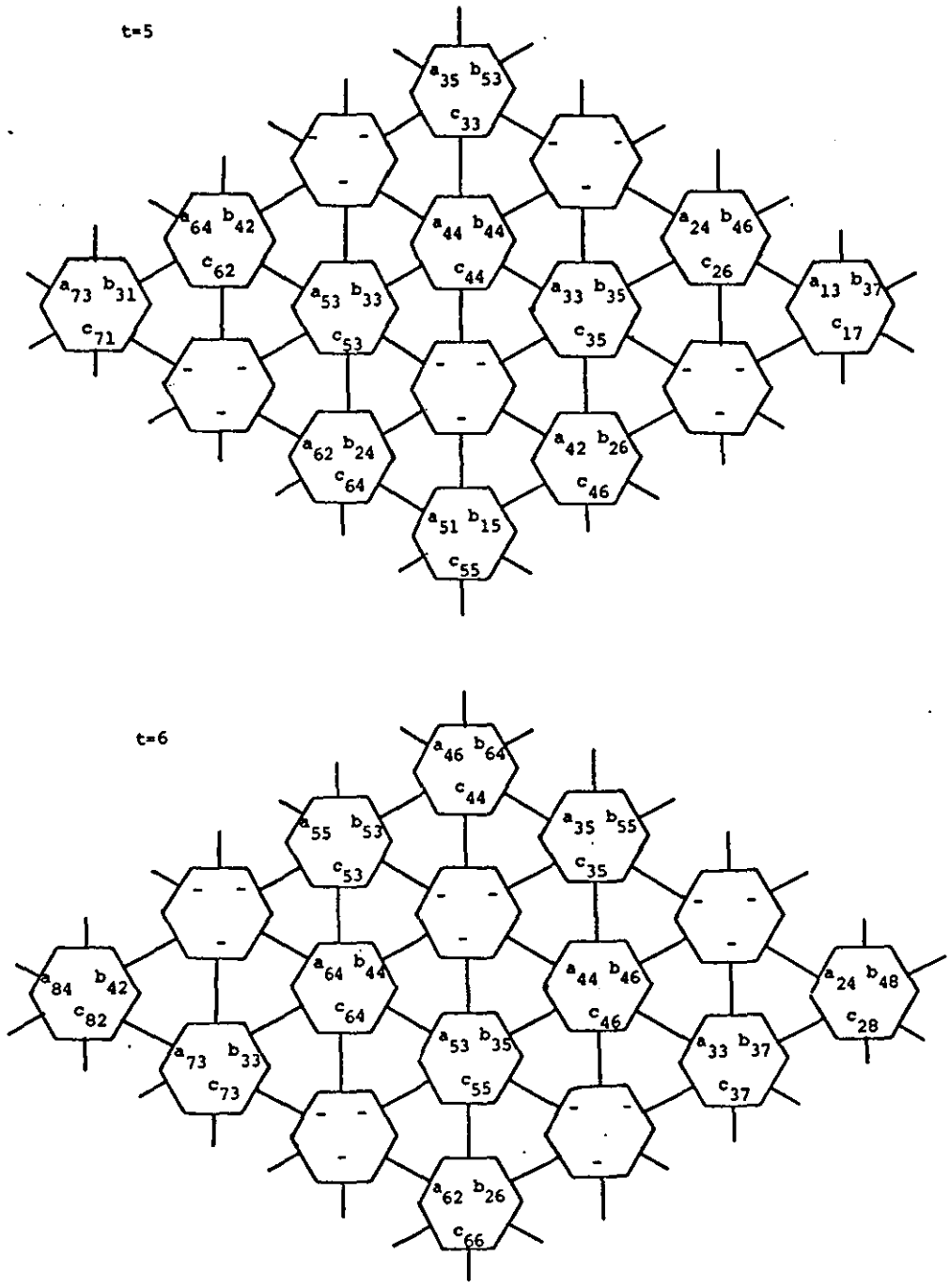


FIGURE 4.1.6: Cont.

which requires at most $T=\frac{3n}{2}+\min(\lceil\frac{w_1}{2}\rceil,\lceil\frac{w_2}{2}\rceil)$ ips cycles. This timing follows from the facts that only a single neutral element is associated with every two genuine input data elements, and the effective bandwidths of A_1 and B_1 are bounded by $\lceil\frac{w_1}{2}\rceil$ and $\lceil\frac{w_2}{2}\rceil$ respectively. As a result the hex array for producing C_1 requires only approximately $w_1w_2/4$, cells quarter that of the normal hex for a matrix product and improves efficiency from e=1/2 to e=2/3 and computes twice as fast. These advantages are retained for the partial products (4.1.23b-d) by converting them to a normal form with the same matrix structure as (4.1.23a) using simple computation preserving row and column interchanges.

For (4.1.23b) C_2 and B_2, interchanges are performed swapping columns i and i+1 for i=1(2)n-1 producing matrices $\bar{C}_2$ and $\bar{B}_2$. In (4.1.23c) C_3 and A_2 produce $\bar{C}_3$ and $\bar{A}_2$ by interchanging row i and i+1 for i=1(2)n-1, while for (4.1.23d), A_2,B_2 and C_4 are permuted to produce $\hat{A}_2$,$\hat{B}_2$ and $\hat{C}_4$ by a two step process, first interchanging rows i and i+1 of A_2 and C_4, and second swapping columns i and i+1 of B_2 and C_4 for i=1(2)n-1. The resulting normal form of (4.1.23),

$$\left.\begin{array}{ll} \text{a)}\ C_1 = A_1B_1 & \text{b)}\ \bar{C}_2 = A_1\bar{B}_2 \\ \text{c)}\ \bar{C}_3 = \bar{A}_2B_1 & \text{d)}\ \hat{C}_4 = \hat{A}_2\hat{B}_2 \end{array}\right\} \qquad (4.1.25)$$

can be solved on the same array used for (4.1.23a) using four passes and a time $T=4(\frac{3n}{2})+4\min(\lceil\frac{w_1}{2}\rceil,\lceil\frac{w_2}{2}\rceil)\approx 2(3n+\min(w_1,w_2))$. That is, using twice the time of the original hex scheme, but a quarter of the cells, and has the interesting quality of preserving the bound AT^2, as $(\frac{A}{4})(2T)^2 = AT^2$, where A and T are the area and time requirements given in Theorem (3.2.1.6).

A D^1-pipe hex design using two layers is constructed by placing

these smaller hexes on separate layers and noticing that C_1 and C_4 have the same structure, as do C_2 and C_3. Hence two passes form the full matrix product by computing C_1 and C_4 on separate layers on the first pass and C_2,C_3 on different layers on the second pass.

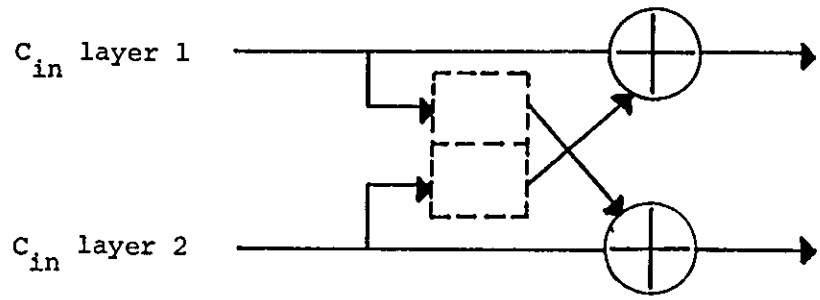
Like the double pipe for matrix vector the final results C_1+C_4 and C_3+C_2 can be overlapped with hex operations by introducing an upper boundary of $\left\lceil \frac{w_1}{2} \right\rceil + \left\lceil \frac{w_2}{2} \right\rceil - 1$ adders to each hexagonal array. Because the arrays actually compute using (4.1.25) the order of the matrix elements must be restored before the addition takes place. Fortunately the localised permutations used to derive (4.1.25) keep the recovery simple, as $\bar{C}_2$,$\bar{C}_3$ and $\hat{C}_4$ have the forms,

$$\bar{C}_2 = \left[\begin{array}{cc|cc|cc} \bar{c}_{12} & 0 & c_{14} & 0 & c_{16} & 0 \\ 0 & c_{21} & 0 & c_{23} & 0 & c_{25} \\ \hline c_{32} & 0 & c_{34} & 0 & c_{36} & 0 \\ 0 & c_{41} & 0 & c_{43} & 0 & c_{45} \\ \hline c_{52} & 0 & c_{54} & 0 & c_{56} & 0 \\ 0 & c_{61} & 0 & c_{63} & 0 & c_{65} \end{array}\right] \quad \bar{C}_3 = \left[\begin{array}{cc|cc|cc} \bar{c}_{21} & 0 & c_{23} & 0 & c_{25} & 0 \\ 0 & c_{12} & 0 & c_{14} & 0 & c_{16} \\ \hline c_{41} & 0 & c_{43} & 0 & c_{45} & 0 \\ 0 & c_{32} & 0 & c_{34} & 0 & c_{36} \\ \hline c_{61} & 0 & c_{63} & 0 & c_{65} & 0 \\ 0 & c_{52} & 0 & c_{54} & 0 & c_{56} \end{array}\right]$$

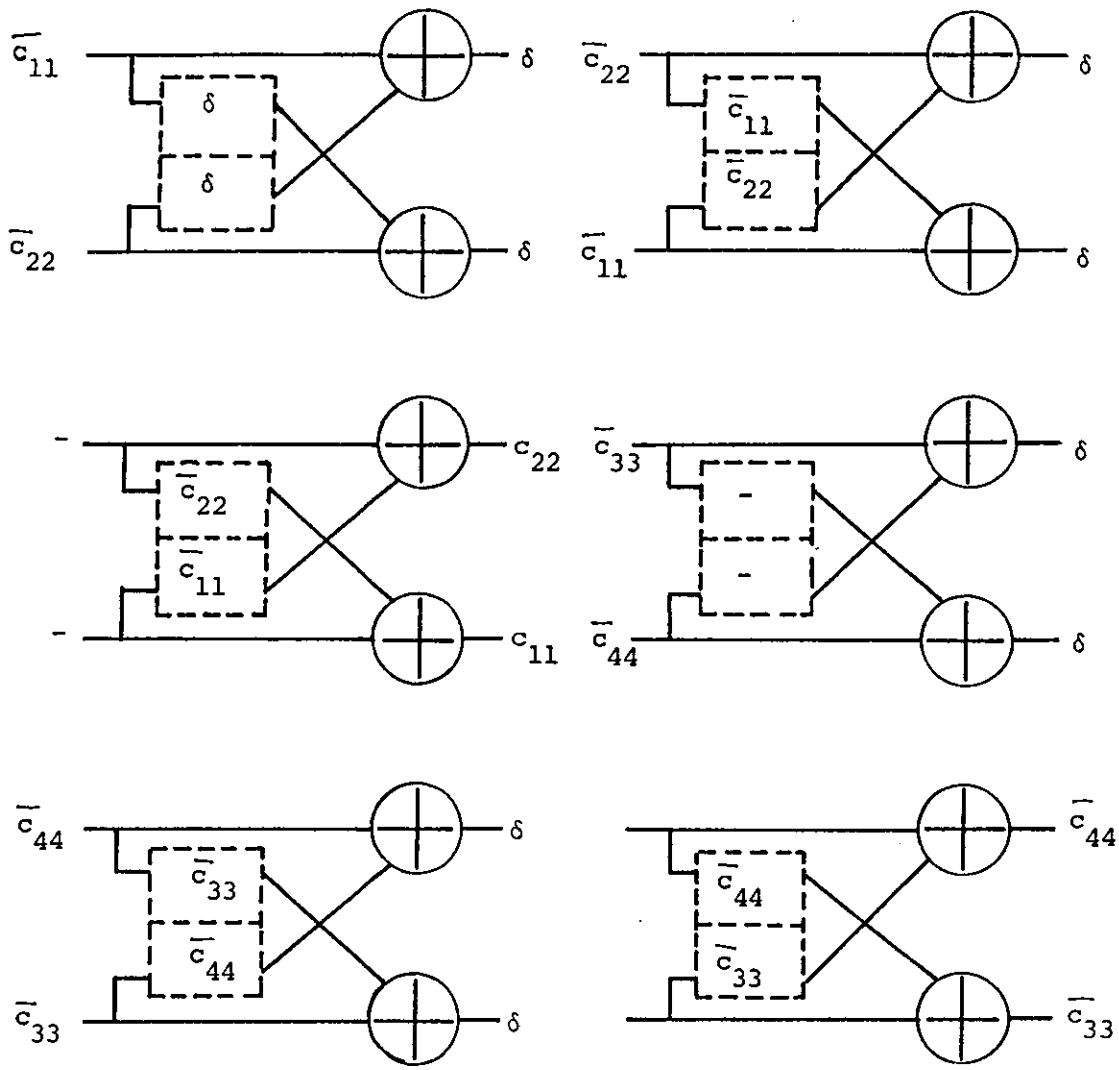
and

$$\hat{C}_4 = \left[\begin{array}{cc|cc|cc} \hat{c}_{22} & 0 & c_{24} & 0 & c_{26} & 0 \\ 0 & c_{11} & 0 & c_{13} & 0 & c_{15} \\ \hline c_{42} & 0 & c_{44} & 0 & c_{46} & 0 \\ 0 & c_{31} & 0 & c_{33} & 0 & c_{35} \\ \hline c_{62} & 0 & c_{64} & 0 & c_{66} & 0 \\ 0 & c_{51} & 0 & c_{53} & 0 & c_{55} \end{array}\right]$$

Simply swapping the diagonal elements of each 2×2 block in $\bar{C}_3$ and $\hat{C}_4$ and adding to C_1 and $\bar{C}_2$ (or vice versa) produces the correct result, and can be achieved using only a single delay with each adder (see Fig. 4.1.7).



a) Adder delay arrangement



b) Production of full result for main diagonal column

FIGURE 4.1.7: Snapshots of double pipe hex adder/delay arrangement

Theorem 4.1.4: The matrix product of two $n \times n$ bandmatrices A and B of bandwidth w_1 and w_2 can be computed on a D^1-pipe hex in at most $T=3n+\min(w_1,w_2)+2$ cycles using $2(\lceil \frac{w_1}{2} \rceil \lceil \frac{w_2}{2} \rceil)$ ips and $2(\lceil \frac{w_1}{2} \rceil + \lceil \frac{w_2}{2} \rceil -1)$ adders and delay cells.

Proof:

Two passes on the two layer design with $\lceil \frac{w_1}{2} \rceil \lceil \frac{w_2}{2} \rceil$ ips cells and $\lceil \frac{w_1}{2} \rceil + \lceil \frac{w_2}{2} \rceil -1$ adders on each layer give $T=2(\frac{3n}{2} + \min(\lceil \frac{w_1}{2} \rceil, \lceil \frac{w_2}{2} \rceil))$.

An extra two cycles are added for delays through the adders.

It also follows that a D^2-pipe hex can be formulated using four layers with a quarter size hex array on each layer and using a single pass to give a time $T=\frac{3n}{2} + \min(\lceil \frac{w_1}{2} \rceil, \lceil \frac{w_2}{2} \rceil)$ cycles. The main result however is the above theorem indicating that a D^1-pipe can compute just as fast as the ordinary scheme with only half the hardware, and improved efficiency.

4.2 BLOCK SCHEMES FOR SYSTOLIC ARRAYS

The double pipe splitting relies on the fact that matrix multiplication problems can be partitioned into independent subtasks which fall neatly into multi-layer arrangements. Arrays involving feedback such as the LU factorisation hex of Fig.(3.2.2.3) and the substitution array in Fig.(3.2.2.1) do not partition or split directly, and create difficulties in allocating cells to layers and retiming dataflow, and alternative methods must be sought to improve efficiency.

For the substitution array double pipes can be applied indirectly, by solving a modified form of the problem discussed in Robert & Tchuente [84], and summarized below. If L is an $n \times n$ lower triangular

matrix of band width q, and D is the diagonal matrix formed from L, then the substitution process,

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

is modified by putting,

$$x = D^{-1}y \ , \tag{4.2.2}$$

and,

$$By = LD^{-1}y = b \ , \tag{4.2.3}$$

essentially dividing columns by their corresponding diagonal values to normalise L, and then,

$$y = Pz \ , \tag{4.2.4}$$

$$Cz = BPz = b \ , \tag{4.2.5}$$

with,

$$P = \begin{bmatrix} 1 & & & \\ -b_{21} & 1 & & \\ & -b_{32} & \ddots & \\ & & \ddots & 1 \\ & & -b_{n,n-1} & 1 \end{bmatrix} \tag{4.2.6}$$

Thus, C has a unit diagonal and null first subdiagonal and is solved using the double pipe shown in Fig.(4.2.1). Fortunately the above procedure can be overlapped with array computation by the use of additional preprocessing cells (see Robert & Tcheunte [84]).

In the case of LU factorisation Robert [85] proposed an explicit 2×2 block form of computation and a modified hexagonal array yielding:

Theorem 4.2.1 [2×2 block LU]: If A is an n×n band matrix with bandwidth w=p+q-1, an array with no more than pq processors can compute the 2×2 block LU factorisation in T=2n+min(p,q) cycles including input/output time with efficiency e=1/2.

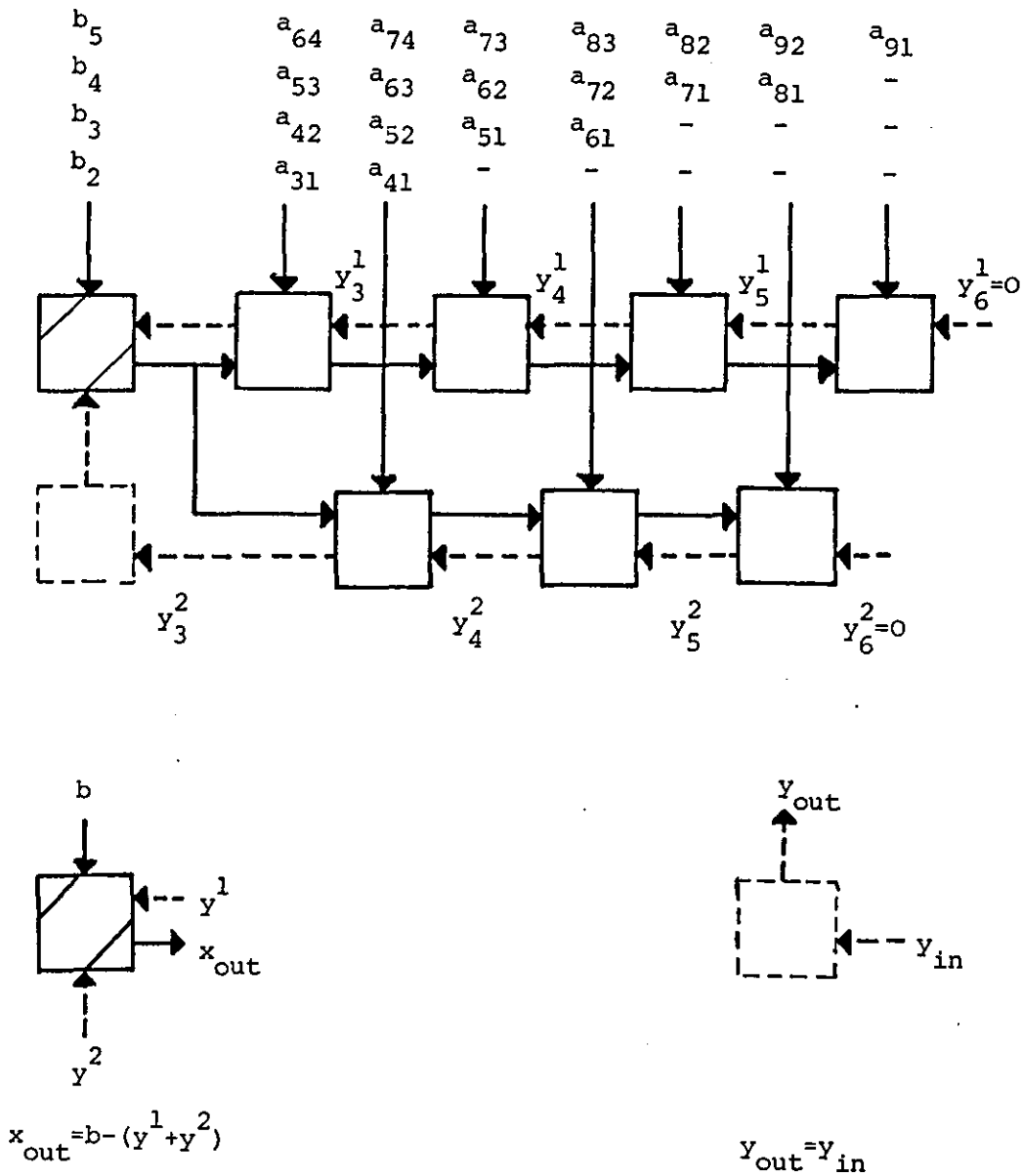


FIGURE 4.2.1: Solution of Ly=b by a double pipe (from Robert [85])

In this section we extend ideas of explicit block computation to examine higher block strategies and multi-layer arrays. This is accomplished by introducing the block hex cell which implements a block inner product using k×k point inner products for a k×k block, with the hardware equivalent to k×k inner product cells. The double pipe schemes improved efficiency by removing the neutral elements in input sequences, here the underlying architecture is adjusted to give better hardware performance and decreased computation time. To illustrate the method consider matrix multiplication again.

4.2.1 Block Matrix Multiplication (BMM)

Fig(3.2.1.6) and Theorem(3.2.1.6) give the structure, timing and cell requirements of banded matrix multiplication for the 1×1 block or point case. The same architecture is easily generalised to the BMM methods creating a generic block array form by interpreting each point hex cell as k×k block-hex containing k×k ips cells. Each data element of the input is replaced by a k×k block of elements read in unit time (an ips cycle); and for k≥3 blocks are separated by k-1 synchronising neutral blocks. A corresponding generic timing theorem is then given by:

Theorem 4.2.2 [k×k blocks BMM]

Let A and B be two n×n matrices, partitioned into k×k blocks with block bandwidths $\bar{w}_1$ and $\bar{w}_2$ respectively. Then using a network of $\bar{w}_1 * \bar{w}_2$ hexagonally connected block-cells the product A*B can be found in,

$$T_k = \begin{cases} k(n/k) + k\min(\bar{w}_1, \bar{w}_2) + k-1 & k>3 \\ 3(n/k) + k\min(\bar{w}_1, \bar{w}_2) + k-1 & k\leq 3. \end{cases} \qquad (4.2.1.1)$$

Proof: (By construction)

(i) With k=1, $T=3n+\min(w_1,w_2)$ and requires $\bar{w}_1 * \bar{w}_2$ cells the result of Theorem (3.2.1.6).

(ii) When k=2 (2×2 blocks). The block ips computation has the form,

$$\begin{bmatrix} c_1 & c_2 \\ c_3 & c_4 \end{bmatrix} = \begin{bmatrix} c_1 & c_2 \\ c_3 & c_4 \end{bmatrix} + \begin{bmatrix} a_1 & a_2 \\ a_3 & a_4 \end{bmatrix} * \begin{bmatrix} b_1 & b_2 \\ b_3 & b_4 \end{bmatrix} \qquad (4.2.1.2)$$

c_1 and c_3 are generated using a pipelined tree arrangement equivalent to two point ips cells and three point ips cycles.

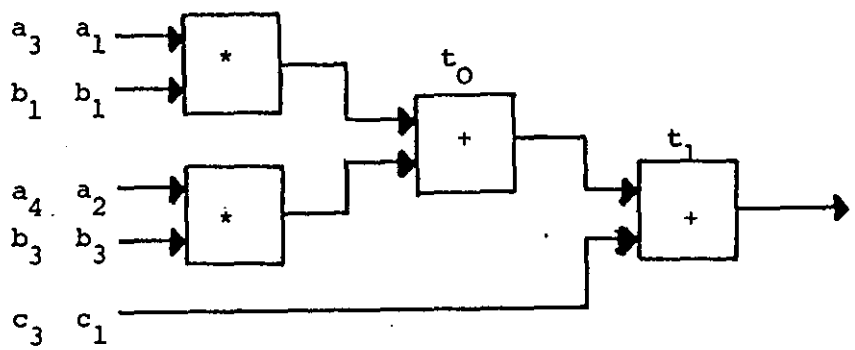


FIGURE 4.2.1.1: 2×2 block ips segment

i.e. cycle 1 : $t_0 = a_1b_1+a_2b_3$ $t_1=\delta$

cycle 2 : $t_1 = t_0+c_1$, $t_0 = a_3b_1+a_4b_3$, result c_1

cycle 3 : $t_1 = t_0+c_3$ result c_3.

The latency of the cell is two point ips cycles and a second tree segment is operated in parallel to form c_2 and c_4 giving a final cell hardware requirement of 2×2=4 point ips cells. Now consider a column of block hex cells running vertically through the array and numbered from bottom to top. The computation of cell i can be overlapped with the start-up of cell i+1 and the result sequence re-synchronised by delaying c_{ij} blocks by one cycle. It follows that the delay through the array is $2\min(w_1,w_2)$ ips cycles and a total of $3\lceil n/2\rceil+1$ cycles are required to output the $\lceil n/2\rceil$ c_{ij} blocks. Hence,

$$T_2 = 3\lceil n/2\rceil + 2\min(\bar{w}_1,\bar{w}_2)+1 \ . \qquad (4.2.1.3)$$

(iii) For k=3 (3×3 blocks), a 3×3 block-hex cell is formed from 9 point ips cells using three pipelined tree segments each equivalent in hardware to 3 point ips cells. The block hex computes,

$$\begin{bmatrix} c_1 & c_2 & c_3 \\ c_4 & c_5 & c_6 \\ c_7 & c_8 & c_9 \end{bmatrix} = \begin{bmatrix} c_1 & c_2 & c_3 \\ c_4 & c_5 & c_6 \\ c_7 & c_8 & c_9 \end{bmatrix} + \begin{bmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ a_7 & a_8 & a_9 \end{bmatrix} * \begin{bmatrix} b_1 & b_2 & b_3 \\ b_4 & b_5 & b_6 \\ b_7 & b_8 & b_9 \end{bmatrix}$$

and c_1,c_4 and c_7 are computed on a single segment structure.

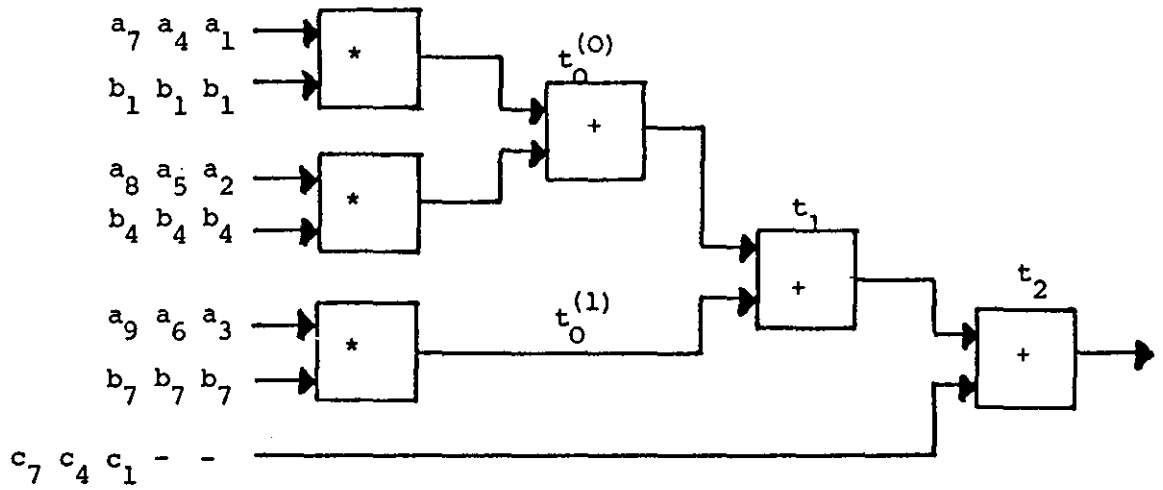


FIGURE 4.2.1.2: 3×3 block ips segment

with c_2,c_5,c_8 and c_3,c_6,c_9 evaluated on the remaining segments. Each segment requires a total of five point ips cycles, viz.,

Step 1 : $t_0^{(0)} = a_1*b_1 + a_2*b_4$, $t_0^{(1)} = a_3*b_7$

Step 2 : $t_1 = t_0^{(0)}+t_0^{(1)}$, $t_0^{(0)} = a_4*b_1+a_5*b_4$, $t_0^{(1)} = a_6*b_7$

Step 3 : $t_2 = t_1+c_1$, $t_1 = t_0^{(0)}+t_0^{(1)}$, $t_0^{(0)} = a_7*b_1+a_8*b_4$, $t_0^{(1)} = a_9*b_7$

Step 4 : $t_2 = t_1+c_4$, $t_1 = t_0^{(1)}+t_0^{(1)}$,

Step 5 : $t_2 = t_1+c_4$

From which we conclude that every k=3 cycles, the hex is ready to receive its next block input. For a column of cells numbered bottom to top when cell i completes Step 3, cell i+1 can be finishing Step 2, and requires c_1 on the next step. Hence operation of cell i and i+1 can be overlapped, with cell i+1 starting only a cycle after cell i. As cells have latency of three cycles and c_{ij} blocks must be delayed by two cycles initially,

$$T_3 = 3\lceil n/3 \rceil + 3\min(\bar{w}_1, \bar{w}_2) + 2 \ , \qquad (4.2.1.4)$$

(iv) k×k blocks (for k>3). Again, a block hex with equivalent hardware of k×k point ips cells is constructed from k pipelined tree segments using k point ips cells. The latency of the block hex is at most k cycles. Results are output on successive cycles giving a total computation time of 2k-1 point ips cycles and the multipliers of the tree are free after k cycles. C_{ij} blocks must therefore arrive every k cycles with k-1 neutral blocks between inputs. Cell i+1 can still start only a single cycle after cell i starts, and the c_{ij} must be delayed initially by k-1 cycles. Hence the timing

$$T_k = k\lceil n/k \rceil + k\min(\bar{w}_1, \bar{w}_2) + k-1 \ , \qquad (4.2.1.5)$$

[End of Proof].

Using this result and additional properties of the generic BMM array the optimal block size for a matrix product is determined as follows:

Block Efficiency: the point scheme has e=1/3 and a point hex computes once in every three cycles. By counting cycles requiring ips

computations and overlapping successive block computations in a single block-hex, 2×2 blocks have e=2/3 and 3×3 blocks e=3/3=1. For k×k blocks when k>3 only the number of cycles required to start up all the array cells is reduced, as the block bandwidths $\bar{w}_1$ and $\bar{w}_2$ are $\lceil w_1/k \rceil$ $\lceil w_2/k \rceil$, with w_1 and w_2 the bandwidths of the point case,

$$T_{opt} = T_3 = 3\lceil n/3 \rceil + 3\min(\bar{w}_1, \bar{w}_2) + 2 \qquad (4.2.1.6)$$

is the minimum time we can hope for when the matrices are really banded, producing an O(n) scheme in contrast to O(3n).

Block hexs with balanced binary tree segments can be introduced to reduce propagation delay through the cell and improve overall array performance. However, for k≥4 a binary tree will have latency $\lceil \log_2 k \rceil + 1$ and produces,

$$T_4 = k\lceil n/k \rceil + (\lceil \log_2 k \rceil + 1)\ \min(\bar{w}_1, \bar{w}_2) + k - 1 \ . \qquad (4.2.1.7)$$

As $\lceil \log 4 \rceil = 2$ it follows that the term containing $\min(\bar{w}_1, \bar{w}_2)$ is reduced even though cell latency in T_3 and T_4 are the same. For large n and really banded systems the saving is asymptotically negligible, for dense systems binary tree arrangements can reduce time significantly but at the cost of increased block cell complexity. Generally, however 3×3 blocks tend to outperform higher blocks even with balanced trees.

Array layout can be achieved in a multi-layer arrangement in two main ways. A natural method is to allocate each tree segment to a separate layer so that only the a_i have to be broadcast between levels. Alternatively, each segment can be aligned vertically with one leaf multiplier on each level. Both schemes produce an effective area on each layer $1/k^{th}$ that of the point case but the latter restricts

broadcasting to the plane of each layer and requires only local inter-layer connections for tree fan-in.

A further drawback is the increased number of inputs and outputs. For a k×k block input to be read in unit time we require k×k inputs, and although more connections are available in a multi-level approach small block size simplifies matters. Fortunately, requirements can be reduced to only k connections for each block input by utilising the tree pipeline features. At any one time there are only k c_{ij} elements inside a block hex, thus k groups of k inputs can be pipelined by filling the neutral block spaces with the k-1 gaps left after the first row. A similar argument is used for A_{ij}, but the B_{ij} must all be present at the start of cell computation, and so must be retimed to input the first (k-1) groups before the A_{ij} elements start to arrive. When $k \leq 3$ these folding arrangements follow naturally from the dataflow of the point array supporting the efficiency argument, for k>3 folding is less intuitive. We conclude that the optimal area time trade-off occurs for 3×3 blocks for BMM.

4.2.2 3*3 Block LU Factorisation

Theorems(3.2.2.3) and (4.2.1) give timings for point and 2×2 LU factorisation in the light of Theorem(4.2.2) and with a choice of 3×3 blocks for BMM arrays, does a similar relationship hold for block LU factorisation (BLUF) arrays? To answer this question we develop a 3×3 BLUF array.

Assuming that A is an n×n irreducible diagonally dominant or symmetric positive definite band matrix of bandwidth w=p+q-1, a 3×3 BLUF scheme is derived as follows. First A is partitioned into a 3×3

block matrix with bandwidth $\bar{w}=r+s-1$ with $p\leq 3r$ and $q\leq 3s$ and when $n=3m$ has the form,

$$A = \begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1r} & & \\ A_{21} & & & & & O \\ \vdots & & & & & \\ A_{s1} & & & & & \\ & & & & & \\ O & & & & & A_{mm} \end{bmatrix}_{n\times n} \qquad (4.2.2.1)$$

Choosing A_{11} as the first pivot we produce,

$$A = \begin{bmatrix} I & & & \\ L_{21} & I & & \\ \vdots & & \ddots & \\ L_{s1} & & & I \end{bmatrix} \begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1r} \\ & & & \\ & & A^{(2)} & \end{bmatrix} \qquad (4.2.2.2)$$

where the blocks in $A^{(2)}$ have been modified and,

$$L_{i1}A_{11} = -A_{i1} \quad , \quad i=2(1)s \; . \qquad (4.2.2.3)$$

Continuing the process on the submatrices $A^{(i)}$ for $i=2(1)m-1$ produces the required block LU form. For purposes of illustration, let A be a 6×6 matrix then,

$$\begin{bmatrix} I & O \\ L_{21} & I \end{bmatrix} \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ O & A_{22}+L_{21}A_{12} \end{bmatrix} \qquad (4.2.2.4)$$

with each A_{ij}, L_{ij} 3×3 blocks,

$$L_{21}A_{11} = -A_{21} \rightarrow L_{21} = -A_{21}A_{11}^{-1} \quad , \qquad (4.2.2.5)$$

and

$$A_{11}^{-1} = \frac{1}{D}\begin{bmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{bmatrix} = \frac{1}{D}\,X \qquad (4.2.2.6)$$

where,

$$D = a_{11}a_1+a_{12}a_2+a_{13}a_3$$

and $a_1 = (a_{22}a_{33}-a_{23}a_{32})$, $a_2 = (a_{23}a_{31}-a_{21}a_{33})$, $a_3 = (a_{21}a_{32}-a_{22}a_{31})$

$b_1 = (a_{13}a_{32}-a_{12}a_{33})$, $b_2 = (a_{11}a_{33}-a_{13}a_{31})$, $b_3 = (a_{12}a_{21}-a_{11}a_{32})$

$c_1 = (a_{12}a_{23}-a_{13}a_{22})$, $c_2 = (a_{13}a_{21}-a_{11}a_{23})$, $c_3 = (a_{11}a_{22}-a_{12}a_{21})$

and

$$D.L_{21} = -\begin{bmatrix} a_{41} & a_{42} & a_{43} \\ a_{51} & a_{52} & a_{53} \\ a_{61} & a_{62} & a_{63} \end{bmatrix}\begin{bmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{bmatrix} \qquad (4.2.2.7)$$

to compute the modified A_{22} we form,

$$A_{22}^{(2)} = A_{22}^{(1)} + D.L_{21}(\frac{1}{D}\,A_{12}) \qquad (4.2.2.8)$$

which can be computed in six steps,

Step 1 : compute a_1, a_2, a_3

Step 2 : compute b_1, b_2, b_3, $D = a_{11}a_1+a_{12}a_2$, $t_1 = a_{13}a_3$

$D.\ell_{41} = a_{41}a_1+a_{42}a_2$, $t_2 = a_{43}a_3$

(and $D.\ell_{51}$, $D.\ell_{61}$ similarly).

Step 3 : compute c_1, c_2, c_3, $a_{14} = a_{14}/(D+t_1)$, $D.\ell_{41} = D.\ell_{41}+t_2$

$D.\ell_{42} = a_{41}b_1+a_{42}b_2$, $t_2 = a_{43}b_3$

($D.\ell_{52}$, $D.\ell_{62}$, a_{15}, a_{16} similarly).

Step 4 : compute $a_{24} = a_{24}/(D+t_1)$, $D.\ell_{42} = D.\ell_{42}+t_2$

$D.\ell_{43} = a_{41}c_1+a_{42}c_2$, $t_2 = a_{43}c_3$

$a_{44} = a_{44}-D.\ell_{41}*a_{14}$, $a_{45} = a_{45}-D.\ell_{41}*a_{15}$, $a_{46} = a_{46}-D.\ell_{41}*a_{16}$

($D.\ell_{53}$, $D.\ell_{63}$, a_{25}, a_{26}, a_{54}, a_{55}, a_{56}, a_{64}, a_{65}, a_{66} similarly)

Step 5 : compute $a_{34} = a_{34}/(D+t_1)$, $D.\ell_{43} = D.\ell_{43}+t_2$

$a_{44} = a_{44} - D.\ell_{42} * a_{24}$, $a_{45} = a_{45} - D.\ell_{42} * a_{25}$, $a_{46} = a_{46} - D.\ell_{42} * a_{26}$ (and others)

Step 6 : $a_{44} = a_{44} - D.\ell_{43} * a_{34}$, $a_{45} = a_{45} - D.\ell_{43} * a_{34}$, $a_{46} = a_{46} - D.\ell_{43} * a_{36}$ (and remaining elements of A_{22}).

Thus, six point ips steps are required to update a single 3×3 block assuming computations of a single step are performed in parallel. On the seventh cycle, A_{22} becomes the new pivot and defines the feedback cycle time of the systolic array. Hence A_{11} and A_{22} must be separated by six cycles, and after 6(m-1) cycles block A_{mm} becomes the pivot.

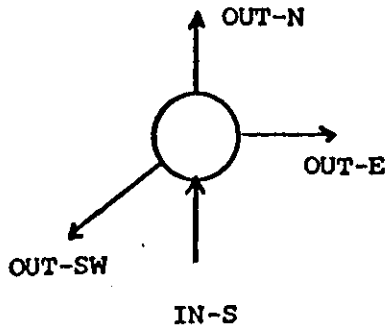
The overall structure of the array is shown in Fig.(4.2.2.1) and is a modified version of the array in Robert [85], extra cells can be added to the right upper boundary to compute LDU instead of LU. As a guide cells compute as follows:-

(i) The upper boundary to the left forms L_{ij} from $D.L_{ij}$ $(i<j)$

(ii) The second line of cells on the upper boundary and to the left compute $D.L_{ij}$, cells to the right $(1/D)(A_{ij}^{(k)})$ for $i \geq j$ and the centre A_{ii}^{-1} and D.

The remaining cells are block ips containing the equivalent of 9 point ips cells each, and are described below.

Cell Definitions



Center (Determinant) Processor

t: INS $a_1,a_2,a_3,a_4,a_5,a_6,a_7,a_8,a_9$
$t_1=a_5a_9-a_6a_8$, $t_2=a_6a_7-a_4a_9$, $t_3=a_4a_8-a_5a_7$

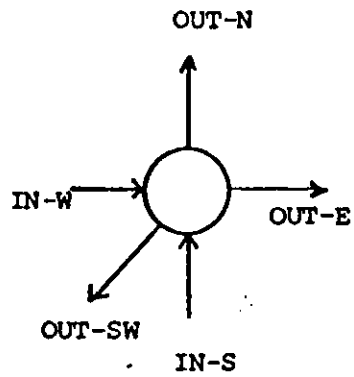
t+1: OUTSW t_1,t_2,t_3
$t_1=a_1t_1+a_2t_2$, $t_2=a_3*t_3$
$t_4=a_3a_8-a_2a_9$, $t_5=a_1a_9-a_3a_7$, $t_6=a_2a_4-a_1a_5$

t+2: OUTN a_1,a_2,a_3,t_1,t_2
OUTSW t_4,t_5,t_6
— OUT E t_1,t_2
$t_7=a_2a_6-a_3a_5$, $t_8=a_3a_4-a_1a_6$, $t_9=a_1a_5-a_2a$

t+3: OUTSW t_7,t_8,t_9
OUTN a_4,a_5,a_6

t+4: OUTN a_7,a_8,a_9

t+5: -



Right Processor (Upper Boundary)

t: INS $a_1,a_2,a_3,a_4,a_5,a_6,a_7,a_8,a_9$

t+1: INW b,c
$a_1=a_1/b+c$, $a_2=a_2/b+c$, $a_3=a_3/b+c$

t+2: OUTN a_1,a_2,a_3
OUTE b,c
OUTSW a_1,a_2,a_3
$a_4=a_4/b+c$, $a_5=a_5/b+c$, $a_6=a_6/b+c$

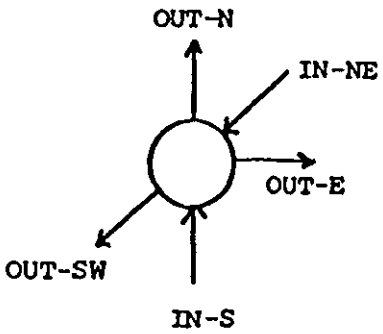
t+3: OUTN a_4,a_5,a_6
OUTSW a_4,a_5,a_6
$a_7=a_7/b+c$, $a_8=a_8/b+c$, $a_9=b+c$

t+4: OUTN a_7,a_8,a_9
OUTSW a_7,a_8,a_9

t+5: -

Left (2nd Line Upper Boundary)

t: INS $a_1,a_2,a_3,a_4,a_5,a_6,a_7,a_8,a_9$
INNE b,c,d
$t_1^{(0)}=ba_1+ca_2$, $t_1^{(1)}=a_3d$
$t_2^{(0)}=ba_4+ca_5$, $t_2^{(1)}=a_6d$
$t_3^{(0)}=ba_7+ca_8$, $t_3^{(1)}=a_9d$



t+1: OUTSW b,c,d

INNE e,f,g

$t_4^{(0)}=ea_1+fa_2$, $t_4^{(1)}=ga_3$, $t_1^{(2)}=t_1^{(0)}+t_1^{(1)}$

$t_5^{(0)}=ea_4+fa_5$, $t_5^{(1)}=ga_6$, $t_2^{(2)}=t_2^{(0)}+t_2^{(1)}$

$t_6^{(0)}=ea_7+fa_8$, $t_6^{(1)}=ga_7$, $t_3^{(2)}=t_3^{(0)}+t_3^{(1)}$

t+2: OUTN $t_1^{(2)}, t_2^{(2)}, t_3^{(2)}$

OUTSW e,f,g

INNE h,i,j

OUTE $t_1^{(2)}, t_2^{(2)}, t_3^{(2)}$

$t_7^{(0)}=ha_1+ia_2$, $t_7^{(1)}=ja_3$, $t_4^{(2)}=t_4^{(0)}+t_4^{(1)}$

$t_8^{(0)}=ha_4+ia_5$, $t_8^{(1)}=ga_6$, $t_5^{(2)}=t_5^{(0)}+t_5^{(1)}$

$t_9^{(0)}=ha_7+ia_8$, $t_9^{(1)}=ga_9$, $t_5^{(2)}=t_5^{(0)}+t_5^{(1)}$

t+3: OUTN $t_4^{(2)}, t_5^{(2)}, t_6^{(2)}$

OUTSW h,i,j

OUTE $t_4^{(2)}, t_5^{(2)}, t_6^{(2)}$

$t_7^{(2)}=t_7^{(0)}+t_7^{(1)}$

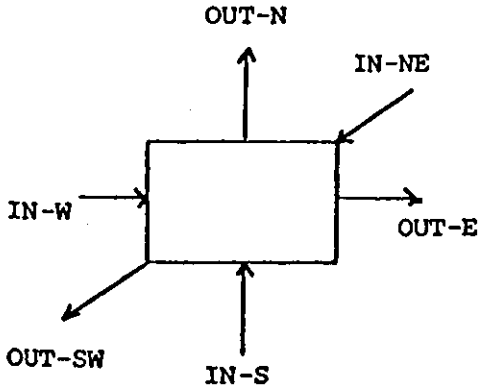
$t_8^{(2)}=t_8^{(0)}+t_8^{(1)}$

$t_9^{(2)}=t_9^{(0)}+t_9^{(1)}$

t+4: OUTN $t_7^{(2)}, t_8^{(2)}, t_9^{(2)}$

OUTE $t_7^{(2)}, t_8^{(2)}, t_9^{(2)}$

t+5: -



Block IPS

t: OUTN $\bar{a}_1,\bar{a}_2,\bar{a}_3,\bar{a}_4,\bar{a}_5,\bar{a}_6,\bar{a}_7,\bar{a}_8,\bar{a}_9$
INS $a_1,a_2,a_3,a_4,a_5,a_6,a_7,a_8,a_9$
OUTE b_3,b_6,b_9
OUTSW c_7,c_8,c_9

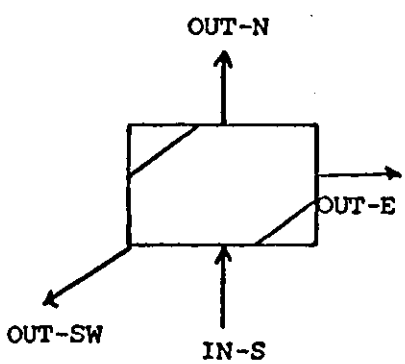
t+1: -

t+2: -

t+3: INW b_1,b_4,b_7
INNE c_1,c_2,c_3
$a_1=a_1+b_1c_1,\ a_2=a_2+b_1c_2,\ a_3=a_3+b_1c_3$
$a_4=a_4+b_4c_1,\ a_5=a_5+b_4c_2,\ a_6=a_6+b_4c_3$
$a_7=a_7+b_7c_1,\ a_8=a_8+b_7c_2,\ a_9=a_9+b_7c_3$

t+4: INW b_2,b_5,b_8
INNE c_4,c_5,c_6
OUTE b_1,b_4,b_7
OUTSW c_4,c_5,c_8
$a_1=a_1+b_2c_4,\ a_2=a_2+b_2c_5,\ a_3=a_3+b_2c_6$
$a_4=a_4+b_5c_4,\ a_5=a_5+b_5c_5,\ a_6=a_6+b_5c_6$
$a_7=a_7+b_8c_4,\ a_8=a_8+b_8c_5,\ a_9=a_9+b_8c_6$

t+5: INW b_3,b_6,b_9
INNE c_7,c_8,c_9
OUTE b_2,b_5,b_8
OUTSW c_4,c_5,c_6
$\bar{a}_1=a_1+b_3c_7,\ \bar{a}_2=a_2+b_3c_8,\ \bar{a}_3=a_3+b_3c_9$
$\bar{a}_4=a_4+b_4c_7,\ \bar{a}_5=a_5+b_4c_8,\ \bar{a}_6=a_6+b_4c_9$
$\bar{a}_7=a_7+b_9c_7,\ \bar{a}_8=a_8+b_9c_8,\ \bar{a}_9=a_9+b_9c_9$



Center cell (1st Line Upper Boundary)

t: -

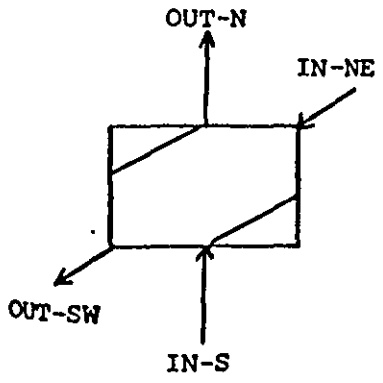
t+1: -

t+2: INS a_1,a_2,a_3,t_1,t_2
$D=t_1+t_2$

t+3: OUTN a_1,a_2,a_3
OUTSW D
INS a_4,a_5,a_6

t+4: OUTN a_4,a_5,a_6
INS a_7,a_8,a_9

t+5: OUTN a_7,a_8,a_9



Left Processor (1st Line Upper Boundary)

t: -

t+1: -

t+2: INNE D
INS a_1, a_2, a_3
$a_1=a_1/D$, $a_2=a_2/D$, $a_3=a_3/D$

t+3: INS a_4, a_5, a_6
OUTN a_1, a_2, a_3
$a_4=a_4/D$, $a_5=a_5/D$, $a_6=a_6/D$

t+4: INS a_7, a_8, a_9
OUTN a_4, a_5, a_6
$a_7=a_7/D$, $a_8=a_8/D$, $a_9=a_9/D$
OUTN a_7, a_8, a_9

The input format for the array is given in Fig.(4.2.2.2), where each data entry is a 3×3 block of elements. Computation starts when A_{11} reaches the determinant cell, requiring a delay of 3min(r,s) for it to filter up through the array. As the last pivot block A_{mm} leaves the determinant cell after 6m=6(n/3) cycles, the total computation time is T=2n+3min(r,s) and when p=3r and q=3s T=2n+min(p,q). We also see from the cell definitions that each cell computes fully for approximately three cycles in six, producing an efficiency of e=1/2.

Theorem 4.2.2.1: [3×3 BLUF] The 3×3 block LU factorisation of an n×n band matrix A can be computed in T=2n+min(p,q) with efficiency e=1/2 using rs block ips cells, where p=3r, q=3s.

Comparing this theorem and theorem (4.2.1) we find that changing from 2×2 BLUF to 3×3 BLUF has no benefits in computation time or efficiency. The result is interesting because it indicates that the increases in efficiency and computation apparent in BMM arrays do not

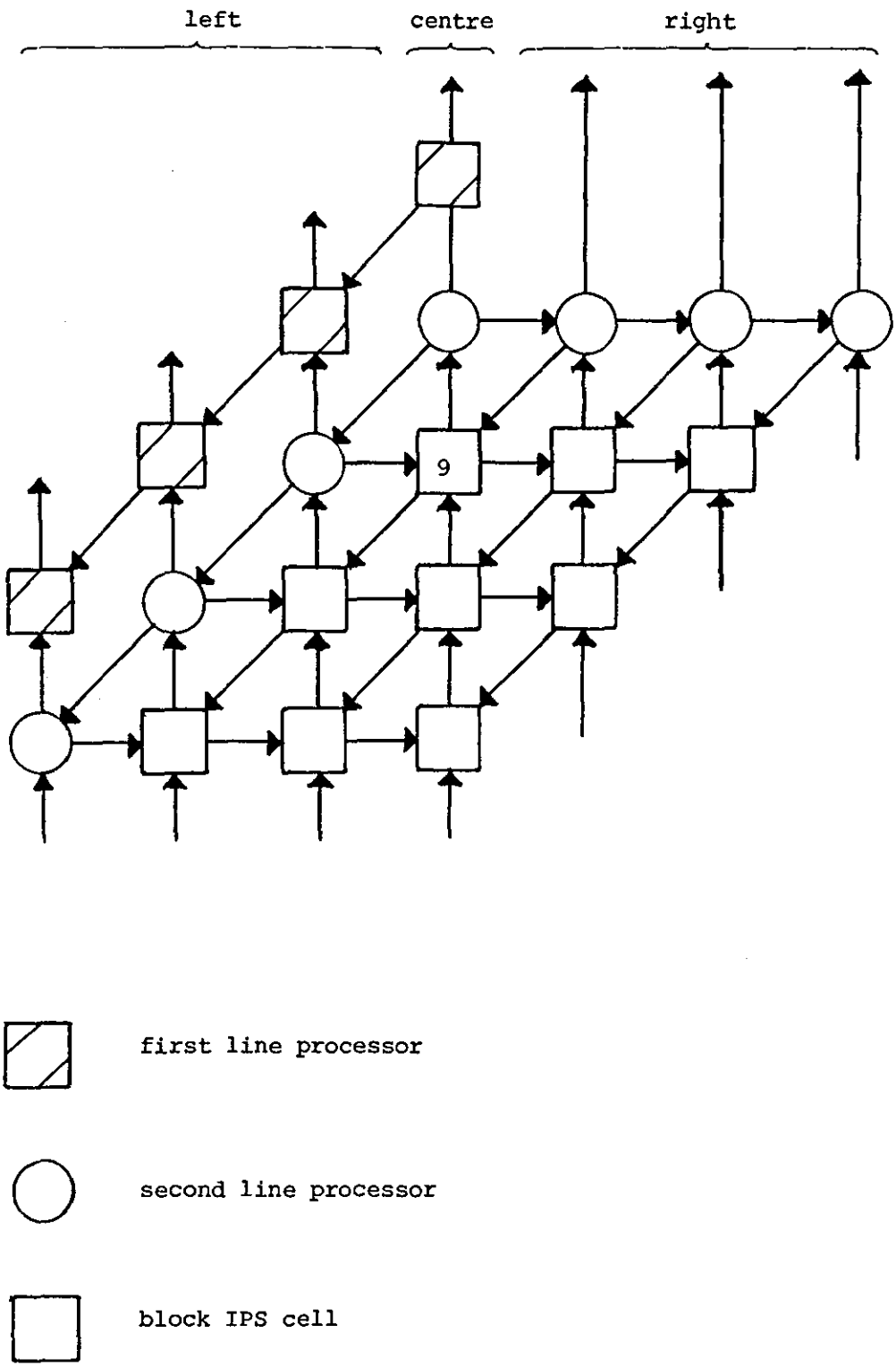
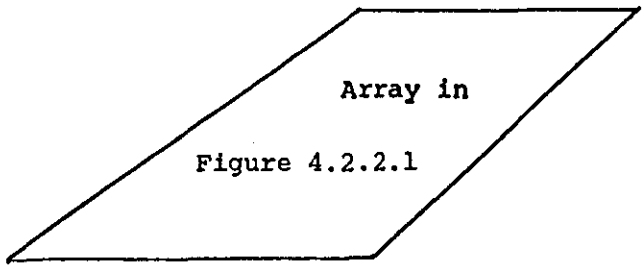
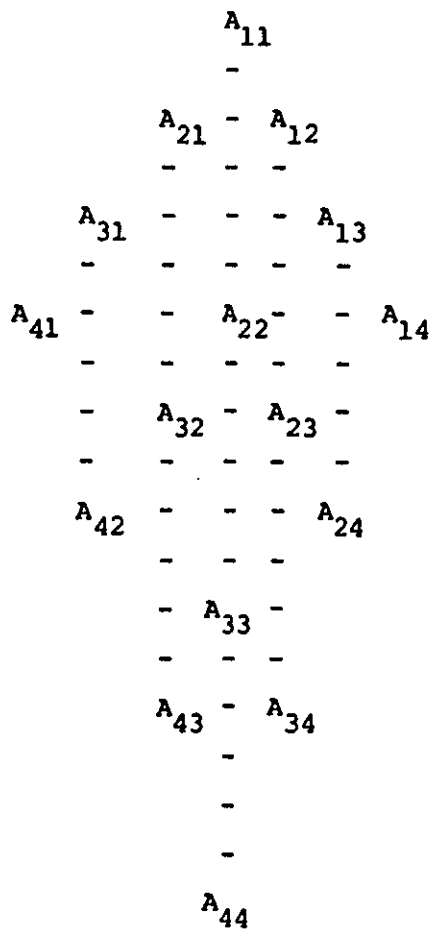


FIGURE 4.2.2.1: Arrangement for 3*3 block LU factorisation
(with p=12, q=12, w=23)



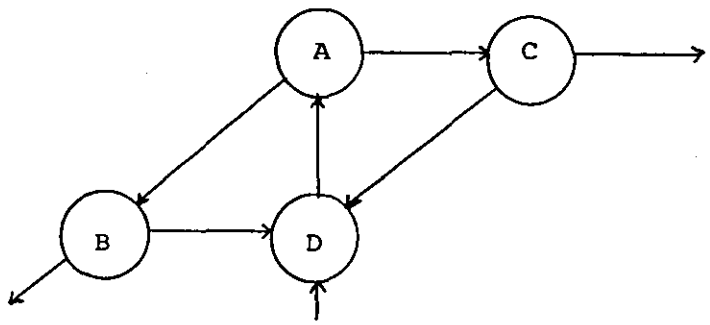


A_{ij} = 9 elements corresponding to block A_{ij}
(can be input in parallel or sequentially depending on bandwidth of the array and cell structure)

N.B. cell structure design implies parallel I/O i.e. 9 input lines per block.

FIGURE 4.2.2.2: Input format for 3*3 LU systolic array

carry over to the BLUF methods, even though the modification part of the BLUF hex is essentially a matrix product form. This mismatch in timings results from feedback in the array and can be understood by considering the smallest feedback loop of the array denoted by the directed graph



NODE A = computation of determinant and inverse of a k×k matrix

NODE B = computation of $D.L_{ji}$ part of new multiplier matrix

NODE C = computes $(1/D)A_{ij}$ part of modification matrix

NODE D = actually computes $A_{jj}=A_{jj}-D(L_{ji})*(1/D)A_{ij}$

The times T_{ABD} and T_{ACD} represent the number of point ips cycles needed to traverse the cycle ABD or ACD and return to A, hence the minimum time for a complete block modification, is

$$\phi = MAX(T_{ABD},T_{ACD}) = T_{ABD} . \qquad (4.2.2.9)$$

This is also the time between block inputs, and directly determines array performance. To improve on the point LU scheme we must satisfy,

$$\frac{\phi n}{k} < 3n \rightarrow \frac{\phi}{k} < 3 . \qquad (4.2.2.10)$$

For the 2×2 block scheme k=2 and ϕ=4 giving 4/2=2<3 and for 3×3 blocks k=3, ϕ=6 giving 6/3=2<3, thus to improve on the 2×2 block scheme $\phi/k<2$ must hold.

Now suppose the computation starts at node A, then this node must compute:

(i) The first column of the product D.X where (1/D)*D.X is the inverse of the pivot block

(ii) As many terms of D as possible

before nodes B and C can start and denote this time by ϕ_1. Node B must compute the first column of $D.L_{ji}$ before Node D can begin operation. If each column of $D.L_{ji}$ is computed in parallel using a binary tree layout the latency of the node is $\phi_2=\lceil \log_2 k \rceil$.

Then Node D requires an entire block ips operation to be performed, and takes $\phi_3=k$ cycles as a binary tree format cannot be used (observe the operation in the 3×3 case).

Consequently for k×k blocks,

$$T_{ABD} = \phi_1+\phi_2+\phi_3 = \phi_1+\lceil \log_2 k \rceil+k \ . \qquad (4.2.2.11)$$

It follows from (4.2.2.10) that,

$$\frac{T_{ABD}}{k} = \frac{\phi_1+\lceil \log_2 k \rceil+k}{k} < 2 \ , \qquad (4.2.2.12)$$

for an improvement on 2×2 block schemes. If we consider 4×4 blocks k=4 then (4.2.2.12) yields,

$$\frac{T_{ABD}}{k} = \frac{\phi_1+\lceil \log_2 4 \rceil+4}{4} = \frac{\phi_1+6}{4} < 2 \qquad (4.2.2.13)$$

implying that $\phi_1<2$ or $\phi_1=1$ for a whole number of cycles. Computing the algebraic form of the 4×4 D.X matrix confirms that a single D.X element, even when binary fanin trees are used, requires at least two sequential multiplications. Thus, $\phi_1\geq 2$ and 4×4 BLUF schemes cannot improve on 2×2 schemes. Notice that this statement is valid even if unlimited hardware is allowed in the computation of D.X by node A.

Corollary 4.2.2.1 BLUF arrays for k×k blocks where $k \geq 4$ cannot improve timing and efficiency over 2×2 or 3×3 BLUF arrays.

Finally we consider the hardware requirements for 2×2 and 3×3 BLUF arrays. In the following discussion the array architecture in Fig. (4.2.2.1) will suffice as a generic block LU matrix architecture.

For the 2×2 scheme we assign hardware requirements of cells as follows:-

(i) block ips cell = 4 point ips cells

(ii) processors of 2nd line-to left = 4 " " "

(iii) processors of 2nd line-to right = 2 " " "

(iv) processors of 1st line-to left = 2 " " "

(v) centre processor (2nd line) = 2 " " "

(vi) centre processor (1st line) = none (delay cell)

Notice that omitting the processors of the first line Fig.(4.2.2.1) is a skewed rectangle with dimensions r and s. Hence amalgamating (iii) and (iv) gives a simple hardware bound on 4rs. Now, the n×n matrix A has bandwidth w=p+q-1 and r and s must be the smallest integers such that:

$$p = \begin{cases} 2r & A \text{ full} \\ 2r-1 & p \text{ odd} \\ 2r-2 & p \text{ even} \end{cases} \qquad q = \begin{cases} 2s & A \text{ full} \\ 2s-1 & q \text{ odd} \\ 2s-2 & q \text{ even} \end{cases}$$

giving block bandwidth $\bar{w}=r+s-1$. From Theorem (3.2.2.3) the point case requires pq point ips cells, so to save or use equal hardware in 2×2 schemes,

$$pq \geq 4rs \; . \tag{4.2.2.14}$$

It follows immediately that when A is really banded and p=2r-1, q= 2r-1 that an extra 2r+2s-1 point ips cells are required in the 2×2 case, and

Table(4.2.2.1) indicates overheads for general r and s.

Similarly for 3×3 blocks hardware requirements are assigned as follows:-

(i) Block ips = 9 point ips

(ii) 2nd processor line (left) = 9 " "

(iii) 2nd line processor (right) = 3 " "

(iv) 1st line processor (left) = 3 " "

(v) centre processor 2nd line = 9 " "

(vi) centre processor 1st line = 1 adder (amalgamated with (v))

This time when (iv) and (iii) are amalgamated they produce a count of 6 point ips, to simplify the argument we add an additional 3 ips to give uniform cell allocation. For 3×3 blocks r and s must be the smallest integers such that,

$$p = \begin{cases} 3r & \text{A full} \\ 3r-1 & \\ 3r-2 & \\ 3r-3 & \text{factor of 3} \end{cases}, \quad q = \begin{cases} 3s & \text{A full} \\ 3s-1 & \\ 3s-2 & \\ 3s-3 & \text{factor of 3} \end{cases}$$

consequently $pq \geq 9rs$, (4.2.2.15)

and Table(4.2.2.2) gives hardware overheads for different r and s.

Intuitively the additional hardware occurs because the addition of a sub(super) diagonal adds a column or row to the rectangle of cells. For the point case, these additions are rows and columns of point ips cells, whereas in the 2×2 block scheme 4-point ips cells are incorporated, and 9-point ips cells for 3×3 blocks. This applies even when the outer blocks of the band include some zero sub(super) diagonals which the point scheme exploits. Also, notice that the values in Table

(4.2.2.1) are weak lower bounds as the amalgamation of the 1st line cells is only exact for r=s. Generally when s>r and extra 2(s-r) point ips cells are required to fill out the remaining processors on the 1st line to the left, and for s<r, 2(r-s) point ips cells must be removed. The bounds in Table(4.2.2.2) are weak upper bounds as we introduced an extra 3(r-1) point ips cells to simplify calculations. It follows that the bounds holds for 2r≥s, and for s>2r 3(s-2r) point ips cells must be added to the architecture. Fortunately s≈r is often the case and the bounds are quite accurate, and we can conclude from the tables that BLUF schemes only use the same hardware as the point scheme when the matrix A is full. In fact the 3×3 case saves n-3 point ips cells and so uses less hardware than the 2×2 case; but for a full matrix $O(n^2)$ point cells are required and the saving is negligible.

2×2 blocks

p \ q	2s	2s-1	2s-2
2r	∅	2r	4r
2r-1	2s	2r+2s-1	4r+2s-2
2r-2	4s	2r+4s-2	4r+4s-4

TABLE 4.2.2.1: Lower bound on additional point ips hardware in 2×2 block LU array than in point scheme

3×3 blocks

p \ q	3s	3s-1	3s-2	3s-3
3r	∅	12r	6r	9r
3r-1	12s	12r+12s+8	6r+12s-2	9r+12s-3
3r-2	6s	12r+6s-2	6r+6s+4	9r+6s-6
3r-3	9s	12r+9s-4	6r+9s-6	9r+9s-9

TABLE 4.2.2.2: Upper bound on point ips hardware added for 3×3 block LU than for point LU scheme

4.2.3 Complex Matrix Problems

To conclude this section on block partitioning we briefly consider systolic arrays for complex matrix product and LU factorisation, where matrix elements have the form,

$$a_{kj} = \begin{cases} a_{kj}^{(1)} & \text{real} \\ a_{kj}^{(1)} + i a_{kj}^{(2)} & \text{complex} \end{cases} \qquad (4.2.3.1)$$

where $a_{kj}^{(1)}, a_{kj}^{(2)} \in \mathbb{R}$, and $i=\sqrt{-1}$, and the basic complex inner product for complex numbers (e+fi), (a+bi) and (c+di) is the 2×2 form,

$$\begin{bmatrix} e & -f \\ f & e \end{bmatrix} = \begin{bmatrix} e & -f \\ f & e \end{bmatrix} + \begin{bmatrix} a & -b \\ b & a \end{bmatrix} * \begin{bmatrix} c & -d \\ d & c \end{bmatrix} \qquad (4.2.3.2)$$

$$(e+fi) = (e+fi) + (a+bi) * (c+di) .$$

Hence a n×n complex matrix product is translated to an (2n)×(2n) real matrix product by expanding each complex element to a 2×2 real matrix. A further re-ordering of rows and columns in the (2n)×(2n) matrix gives a partitioned form,

$$\left[\begin{array}{c|c} A & -B \\ \hline B & A \end{array}\right] \left[\begin{array}{c|c} C & -D \\ \hline D & C \end{array}\right] = \left[\begin{array}{c|c} AC-BD & -(AD+BC) \\ \hline BC+AD & AC-BD \end{array}\right] \qquad (4.2.3.3)$$

with A,C and AC-BD corresponding to the $a_{kj}^{(1)}$ (or real) components of the matrix elements and B,D and BC+AD the $a_{kj}^{(2)}$ (or imaginary) components. Using the symmetry property only AC-BD and BC+AD need to be calculated and can be performed with only 3 n×n real matrix multiplications of the form,

$$\text{a)}\ M_1 = P^{(1)}C, \quad \text{b)}\ M_2 = BP^{(2)}, \quad \text{c)}\ M_3 = P^{(3)}D \qquad (4.2.3.4)$$

$$\text{where}\ P^{(1)} = A-B, \quad P^{(2)} = C-D, \quad P^{(3)} = A+B\ . \qquad (4.2.3.5)$$

From the construction of (4.2.3.3), A,B and C,D have the same

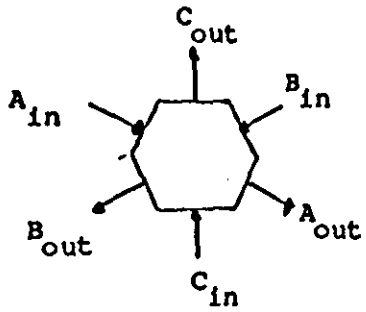
bandwidths of their corresponding complex matrices, i.e. w_1 and w_2 respectively. As matrix addition and subtraction do not affect the bandwidth $P^{(1)}$, $P^{(2)}$ and $P^{(3)}$ have bandwidths w_1 and w_2. Consequently construction of M_1, M_2 and M_3 can be interleaved on a point hex array described by theorem (3.2.1.6) to give efficiency e=1 without increasing computation time significantly when the construction of $P^{(1)}$, $P^{(2)}$ and $P^{(3)}$ is overlapped with the array calculation.

Theorem 4.2.3.1: The matrix product of two $n^{\times}n$ matrices A and B with bandwidths w_1 and w_2 is performed in $T=3n+\min(w_1,w_2)+4$ ips cycles using w_1w_2 ips cells and w_1+w_2-1 pre(post) processing cells.

Proof: (the array is shown in Fig. (4.2.3.1)).

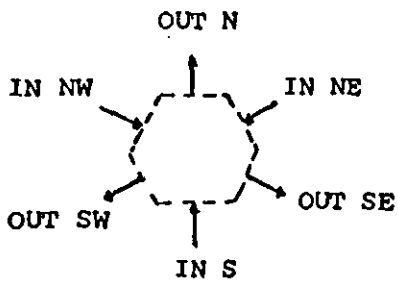
From the above discussion M_1, M_2 and M_3 are computed in $T=3n+\min(w_1,w_2)$ using interleaving and w_1w_2 point ips cells. It remains only to explain how $P^{(1)}$, $P^{(2)}$ and $P^{(3)}$ are produced, and the final output results generated. We start by adding w_1+w_2-1 pre(post) processing cells to the upper boundary of the hexagonal array, which act as pre-processors for the matrix inputs and post-processors for the result matrix outputs, these compute as shown below, where unknown values are assumed zero, and the inputs of Fig.(4.2.3.1) are $P^{(1)}=A$, $P^{(2)}=D$, $P^{(3)}=O$ (null matrix) initially. Thus, pre-processing and post-processing delays are identical at 2 ips cycles, and results from $M_1+M_2=AC-BD$ and $M_2+M_3=BC+AD$ begin to leave the array pre(post) boundary cells after $\min(w_1,w_2)+4$ cycles producing the correct theorem time.

BASIC IPS CELL



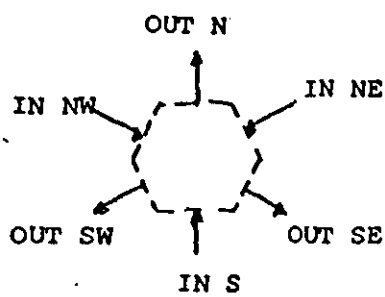
$C_{out} = C_{in} + A_{in}B_{in}$
$B_{out} = B_{in}$
$A_{out} = A_{in}$

CENTER PRE(POST) CELL



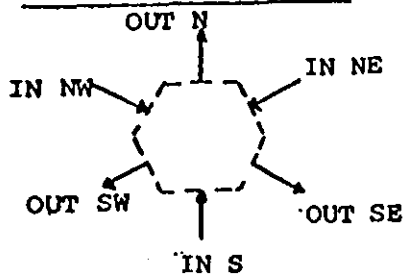
t: IN S A
OUT N B+C
t+1: IN S B
OUT N ZERO
t+2: IN S C
OUT N A+B

LEFT PRE(POST) CELL



t: IN NW A
IN S r_1
OUT SE $\bar{B}$
OUT N r_2+r_3
t+1: IN NW B
IN S r_2
OUT SE $\bar{A}+\bar{B}$
OUT N ZERO
t+2: IN S r_3
OUT SE A-B
OUT N r_1+r_2
$\bar{A}$=A $\bar{B}$=B

RIGHT PRE(POST) CELL



t: IN NE C
IN S r_1
OUT SW $\bar{C}-\bar{D}$
OUT N r_2+r_3

t+1: IN NE D
IN S r_2
OUT SW $\bar{D}$
OUT N ZERO

t+2: IN S r_3
OUT SW C
OUT N r_1+r_2
$\bar{C}=C$, $\bar{D}=D$

r values denote postprocessing and start when first values arrive.

[end of proof].

Because the array efficiency is e=1 and pre(post) processors are relatively simple (consisting of only adders/subtractors and delays), it completes the computation with only four extra ips cycles over the real matrix computation; we conclude that complex calculations are better suited to the traditional hex design. H.T. Kung [84a] considers a linear array implementation using two-level pipelining which requires five instead of three cycles between inputs and adopts a similar strategy to fill some, but not all the wasted cycles.

For matrix factorisation the reciprocal of a complex number is required in addition to the inner product, and is given by,

$$\frac{1}{(a+bi)} = \begin{bmatrix} a & -b \\ b & a \end{bmatrix}^{-1} = \frac{1}{a^2+b^2} \begin{bmatrix} a & b \\ -b & a \end{bmatrix} . \qquad (4.2.3.6)$$

It follows that the (2n)×(2n) matrix obtained by replacing each complex number of the n×n complex matrix by its 2×2 form allows the 2×2 BLUF scheme of Robert [85] to be employed, giving,

Theorem 4.2.3.2 (2×2 complex matrix factorisation)

An n×n complex matrix of bandwidth w can be factorised on a hexagonally connected array in time T=4n+min(2p,2q) and requiring approximately 4pq point ips cells.

Proof: The complex factorisation is equivalent to a real 2×2 block factorisation on a (2n)×(2n) matrix of bandwidth $\overline{w}=2p+2q-1$.

An alternative approach to complex factorisation is to utilise interleaving in a similar manner to the complex product form above. If

$$Ax = b \ , \tag{4.2.3.7}$$

is a complex n×n linear system then its (2n)×(2n) real equivalent is,

$$\begin{bmatrix} C & -D \\ D & C \end{bmatrix} \begin{bmatrix} y \\ z \end{bmatrix} = \begin{bmatrix} c \\ d \end{bmatrix} \tag{4.2.3.8}$$

where C and D are n×n real matrices and x=(y+iz), b=(c+id).

After some manipulation this can be written in the form,

$$\left.\begin{aligned} &\text{a)} \quad Zy = D^{-1}c + C^{-1}d \\ &\text{b)} \quad Zz = D^{-1}d + C^{-1}c \\ &\text{c)} \quad Z = D^{-1}C + C^{-1}D \end{aligned}\right\} \tag{4.2.3.9}$$

which are all subproblems involving n×n real matrices. The factorisation of Z is then a three step process:

$$\left.\begin{aligned} &\text{(i)} \quad \text{Compute } D^{-1}C = K^{(1)} \\ &\text{(ii)} \quad \text{Compute } C^{-1}D = K^{(2)} \\ &\text{(iii)} \quad \text{Factorise } Z = k^{(1)} + k^{(2)} \end{aligned}\right\} \tag{4.2.3.10}$$

As noted in Leiserson [81] a factorisation array can be embedded into a matrix product array, consequently (4.2.3.10) can be solved by interleaving $k^{(1)}, k^{(2)}$ and Z, and making use of the matrix input structure to overlap the summation $k^{(1)} + k^{(2)}$ and the factorisation

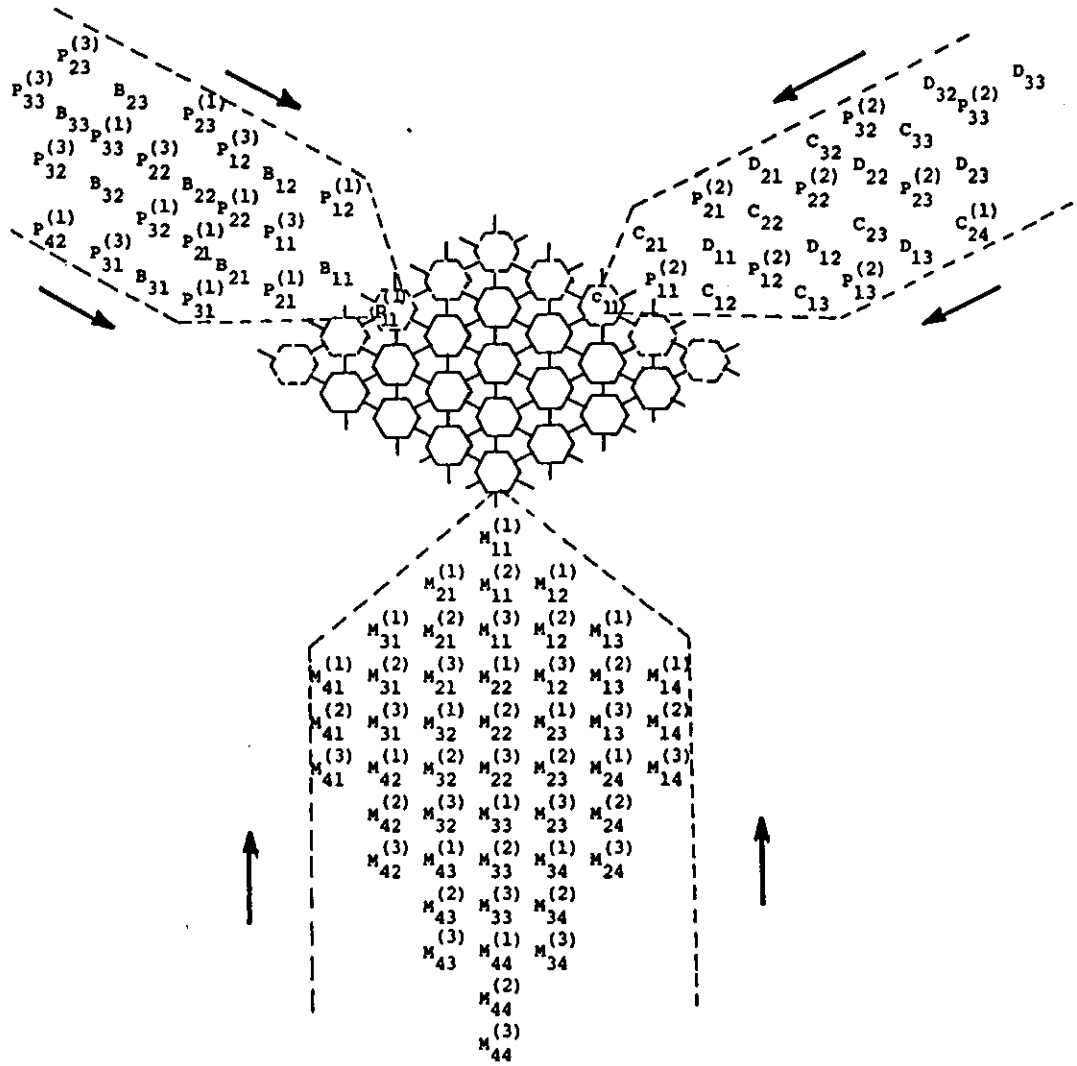


FIGURE 4.2.3.1: Hex connected complex matrix product

of Z. The array is described in Megson & Evans [86h] but relies on the ability to easily form C^{-1}, D^{-1} and the right hand sides of (4.2.3.9) a and b, and the assumption that A is full, which are usually contradictory requirements. When A is banded C^{-1} and D^{-1} would tend to be full and because matrix multiplication is not commutative the summation $k^{(1)} + k^{(2)}$ in conjunction with interleaving requires the maximum size hexagonal array. Hence we have the following theorem.

Theorem 4.2.3.3:

The LU factorisation of an n×n complex matrix with bandwidth w can be performed using point ips computation in time T=3n+(2n-1)+2 cycles and requires $(2n-1)^2$ ips cells.

Proof: (Megson & Evans [86h]).

We conclude that 2×2 block computation is both faster and more area efficient, for banded systems and does not require additional matrix vector arrays for extra computations to modify the righthand side vector of the complex system like (4.2.3.9).

4.3 MATRIX INVERSION BY SYSTOLIC RANK ANNIHILATION

Recently, systolic arrays for arbitrary matrix inversion using hexagonal and orthogonal processor grids requiring O(7n) and O(5n) respectively and $O(n^2)$ processors have been presented (see Rote [85], Robert & Trystram [85]). The latter scheme is almost optimal differing from the theoretical lower bound for matrix inversion by only a few cycles. However, because of their generality, these arrays are often less adaptable to small changes to a coefficient matrix occurring in

iterative type processes. Hence on each iteration a full matrix inverse must be computed. On the other hand rank annihilation techniques are aimed at updating a known inverse when local changes in the coefficient matrix produce global changes in its resulting inverse. As a result rank annihilation is important in a large number of application fields, for instance:

(1) STATISTICS: for updating correlation matrices

(2) GRAPHICS: in spline approximation and refinement

(3) NUMERICAL ANALYSIS: in the repeated solution of problems for which only minor modifications occur in boundary conditions

(4) OPTIMISATION: for solving non-linear systems, and updating the Jacobian matrix.

To employ the rank annihilation technique we require a matrix A with a known inverse A^{-1}, and a matrix B whose elements are only partially different from those of A. The inverse B^{-1} of B is then found by a simple relationship between A^{-1} and B^{-1}. For example, if A and B are n×n matrices and u,v are n component vectors, and B differs from A only by changes in elements along a single row or column then with u and v suitably chosen, B can be written in the form,

$$B = A+uv^T \quad , \tag{4.3.1}$$

and

$$B^{-1} = (A+uv^T)^{-1} \quad , \tag{4.3.2}$$

which after some manipulation (Westlake [68]) yields the Sherman-Morrison or RANK-1 formula,

$$(A+uv^T)^{-1} = A^{-1} - \frac{(A^{-1}u)(v^TA^{-1})}{1+v^TA^{-1}u} \quad . \tag{4.3.3}$$

The basic idea can be extended to produce a relationship between the inverses when A and B differ by changes in m rows or columns producing

the Sherman-Morrison-Woodbury formula or RANK-m method,

$$(A+uv^T)^{-1} = A^{-1} - A^{-1}u(I+v^TA^{-1}u)^{-1}v^TA^{-1}, \tag{4.3.4}$$

with u and v n×m matrices. Clearly when m=1 (4.3.4) reduces to (4.3.3) with $I+v^TA^{-1}u$ a scalar. When m=2 we derive the RANK-2 method and $(I+v^TA^{-1}u)$ is an easily invertible 2×2 matrix. From section (4.2) the difficulty of producing a systolic rank annihilation scheme increases as m becomes larger, because the complexity of forming the m×m matrix $(I+v^TA^{-1}u)^{-1}$ rises significantly. Consequently, we restrict our attention to RANK-1 and 2 schemes and investigate two contrasting designs, i.e., a mesh connected wavefront scheme and a highly concurrent pipelined scheme.

To simplify the discussion we partition the RANK-1 and RANK-2 formulas as follows:

$$\left.\begin{array}{lll} \text{when m=1,} & \text{a)} & B^{-1} = A^{-1} - \dfrac{1}{(1+z)}P \\ \text{where} & \text{b)} & P = xy^t, \quad \text{c) } A^{-1}u = x, \quad \text{d) } v^TA^{-1} = y^T \\ \text{and} & \text{e)} & z = v^Tx = \sum_{i=1}^{n} v_ix_i \end{array}\right\} \tag{4.3.5}$$

$$\left.\begin{array}{lll} \text{and for m=2,} & \text{a)} & B^{-1} = A^{-1} - Py^T \quad \text{or} \quad B^{-1} = A^{-1} - x\bar{P} \\ \text{where} & \text{b)} & P = xC^{-1} \qquad \text{or} \quad \bar{P} = C^{-1}y^T \\ \text{and} & \text{c)} & A^{-1}u = x, \quad \text{d) } v^TA^{-1} = y^T, \quad \text{c) } C^{-1} = (I+v^Tx)^{-1}. \end{array}\right\} \tag{4.3.6}$$

Thus,

$$C^{-1} = \frac{1}{ad-bc}\begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

with,

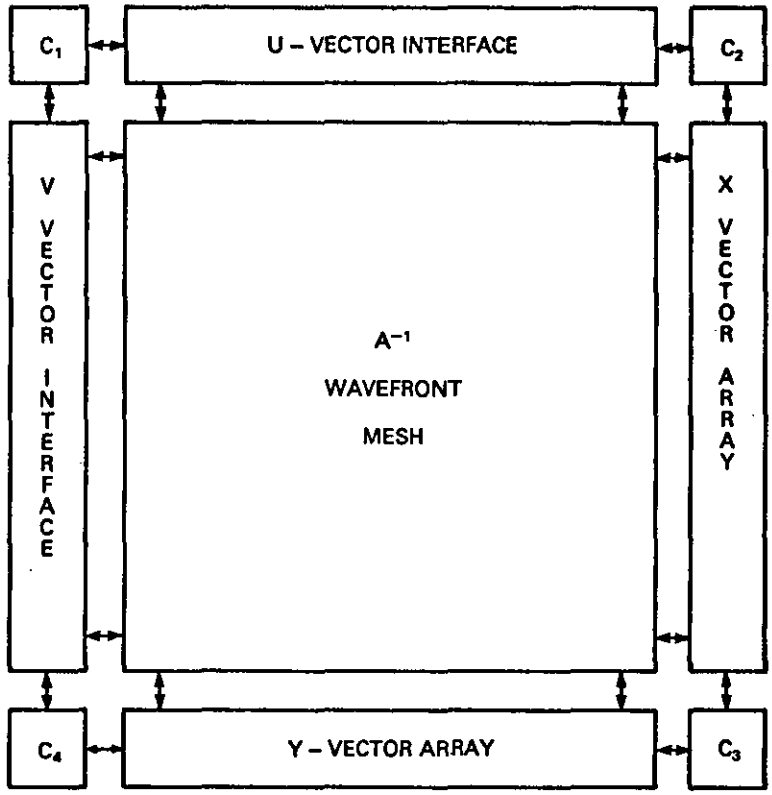
$$\left.\begin{array}{lll} \text{a)} & a = 1 + \sum_{i=1}^{n} v_{1i}x_{i1} & \text{b) } b = \sum_{i=1}^{n} v_{1i}x_{i2} \\ \text{c)} & c = \sum_{i=1}^{n} v_{2i}x_{i1} & \text{d) } d = 1 + \sum_{i=1}^{n} v_{2i}x_{i2} \end{array}\right\} \tag{4.3.7}$$

4.3.1 Mesh Connected Schemes

To define a wavefront model for rank annihilation we introduce a special kind of wavefront processor. The processor is an orthogonally connected square mesh of (n+2)*(n+2) reduced instruction set processing elements. The known inverse A^{-1} is loaded into the n×n mesh of elements embedded inside a Systolic Control Ring (SCR) formed from the first and last rows and columns of the mesh. The processors at grid positions (1,1), (1,n+2), (n+2,n+2) and (n+2,1) are simple controller units capable of generating a number of control signals in the horizontal and vertical directions. The remaining processors perform book-keeping and auxiliary operations as well as relaying control signals. This architecture is shown in Figure(4.3.1.1). The input/output interface consists of cells in the SCR along the first row and column, and the controller initiates computation after A^{-1}, u and v have been loaded into their correct cells.

The last row and column provide auxiliary storage and collect the partial results of (4.3.5b-e) and (4.3.6b-e) for RANK-1 and RANK-2 respectively. Finally the wavefront mesh consists of identical processors receiving a SCR wavefront control from North, South, East or West and performing appropriate inner product operations and data outputs. Consequently the SCR can generate multiple wavefronts in different orientations using only point to point connections around the ring. As variations in processor type are restricted to the boundaries the mesh is also suitable for a VLSI approach to implementation.

For RANK-1 annihilation, and a preloaded mesh, (4.3.5a-e) imposes a strict regime on the computation easily translated to control



GLOBAL ARRANGEMENT OF RANK ANHILATION SYSTOLIC MESH

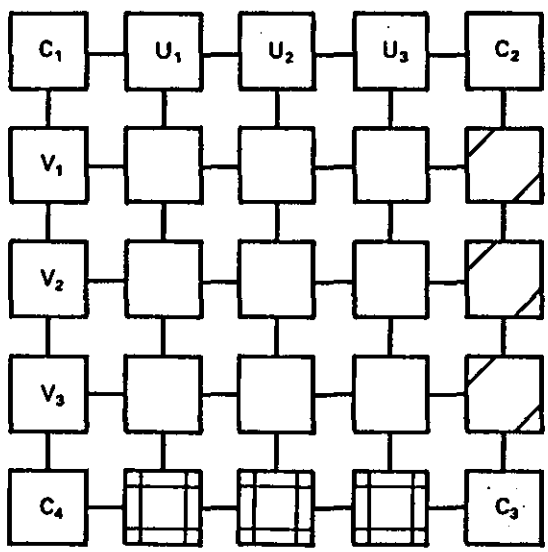


FIGURE 4.3.1.1: 3×3 example mesh connections

wavefronts on the mesh given by the following control algorithm.

RANK-1 MESH CONTROL ALGORITHM:

STEP 1:

a) SCR cell C1 generates a control signal moving right C1→C2, as the control passes through the u_i cells, the value u_i and a control is propagated south down column i. At the same time, a signal C1→C4 is issued causing a zero and control to be propagated from left to right along row i, as it passes through cell v_i. These two wavefronts constructively interfere producing a wavefront W1 moving diagonally from C1→C3, and collecting the partial results x_i, i=1(1)n of $x=A^{-1}u$ systolically from left to right on row i.

b) on the cycle immediately after a) starts another set of signals C1→C2 and C1→C4 are produced reversing the roles of u_i and v_i SCR cells producing a wavefront W2 parallel to W1 and accumulating y_i, i=1(1)n for $v^T A^{-1}=y^T$ along column i of the mesh.

c) W1 and W2 travel at unit speed with W2 one cycle behind W1, consequently the x_i and v_i values are adjacent to each other with v_i one cycle behind x_i.

STEP 2:

On the (n+1)st cycle the first C1→C2, and C1→C4 signals enter C2 and C4 and x_i on row i has i terms left to accumulate and y_j in column j has j+1 terms left to accumulate. On cycle n+2, x_1 is complete and y_1 has one term left to compute, consequently pushing the control values along C2→C3 and C4→C3 allows the x_i and y_i to be loaded into the x-vector and y-vector arrays, with the y_i's one cycle behind the x_i's. The close proximity of the x_i and v_i values implies that the second control value reaching C2 from C1→C2 on cycle n+2 can be employed in computing

(4.3.5e) on its trip C2→C3. If C2 is allowed to initiate the summation with starting result of one, z+1 is computed. After 2n+3 cycles all SCR control values have reached C3, all the y_i and x_i have been stored in vector arrays and C3 contains z+1.

STEP 3:

C3 now takes control, and updates the A^{-1} mesh elements. Control values are sent along C3→C2, and C3→C4 (accompanied by the value z+1). Controls moving along the y-vector array form $y_j=0-y_j/(1+z)$, j=n(-1)1 as they pass through cell j, outputting the control up into column j of the mesh. Likewise controls along the x-vector array simply push x_i into row i of the mesh for i=n(-1)1. The two resulting wavefronts constructively interfere to produce a wavefront W3 travelling diagonally along C3→C1 performing the modification $A^{-1}_{ij}-x_i y_j$ as it moves.

STEP 4:

After 3n+4 cycles SCR controls have reached C4 and C2 and W3 is half-way across the mesh. Pushing controls along C4→C1 and C2→C1 means that all controls arrive back at C1 after 4n+5 cycles and the algorithm terminates.

[End of RANK-1 algorithm].

Now a sequence of r updates can be written as,

$$B = A + \sum_{i=1}^{r} u_i v_i^T , \qquad (4.3.1.1)$$

and in recurrence form,

$$\left.\begin{aligned} A^{(0)} &= A \\ A^{(i)} &= A^{(i-1)} + u_i v_i^T \\ B &= A^{(r)} \end{aligned}\right\} \qquad (4.3.1.2)$$

Theorem 4.3.1.1: r updates of A^{-1} the inverse of n×n matrix A can be performed on a wavefront mesh of $(n+1)^2$ cells incorporating a systolic

control ring in $T=2(n+1)+r(4n+5)$ ips cycles.

Proof:

A^{-1} can be loaded or unloaded in (n+1) cycles, so input/output consumes 2(n+1) cycles altogether.

From (4.3.1.2) when all updates are to distinct rows or columns the u_i, v_i for i=1(1)r can be pre-computed without having to calculate the $A^{(i)}$'s explicitly. The successive loadings of u_i and v_i can be overlapped with a modification in step 4 (above algorithm) using C4→C1 and C2→C1 as loading signals, hence r updates require r(4n+5) cycles giving a total time of $T=r(4n+5)+2(n+1)$ cycles.

Where updates occur on the same row or column more than once the $A^{(i)}$ must be calculated explicitly in order to compute u_i and v_i. If we assume some host machine which supplies u_i and v_i to the mesh exists it must calculate $A^{(i)}$ in the 4n+5 cycles associated with each inverse update. As a row modification is typically of the form,

$$uv^T = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} [v_1\ v_2\ v_3] = \begin{bmatrix} 0 & 0 & 0 \\ v_1 & v_2 & v_3 \\ 0 & 0 & 0 \end{bmatrix} \qquad (4.3.1.3)$$

only n additions are required, and the assumption is valid even for a sequential host machine (the same holds for column modifications). [End of Proof].

Corollary 4.3.1.1: The inverse of an arbitrary matrix B can be computed in $T=4n^2+7n+2$ cycles using the SCR wavefront mesh.

Proof:

Simply put $A^{(0)}=I$ and r=n in (4.3.1.2) and choose $u_i v_i^T$ to be distinct rows (or columns) of B.

Finally, consider the structure of mesh cells, it is clear from the algorithm that all cells compute in a single inner product cycle. The SCR controllers C1, C2, C3 and C4 are simple combinational logic units, while u and v interfaces are registers with additional logic to support the SCR. The x-vector arrays require a register to save x_i and an inner product form to add the partial product term to z+1 as it passes through. Similarly, the y-vector array contains a register for y_i and a subtract/divide cell for modifying y_i using z+1; while the wavefront mesh cell contains a register for A_{ij}^{-1} and performs,

E = W+N*A wavefront W1

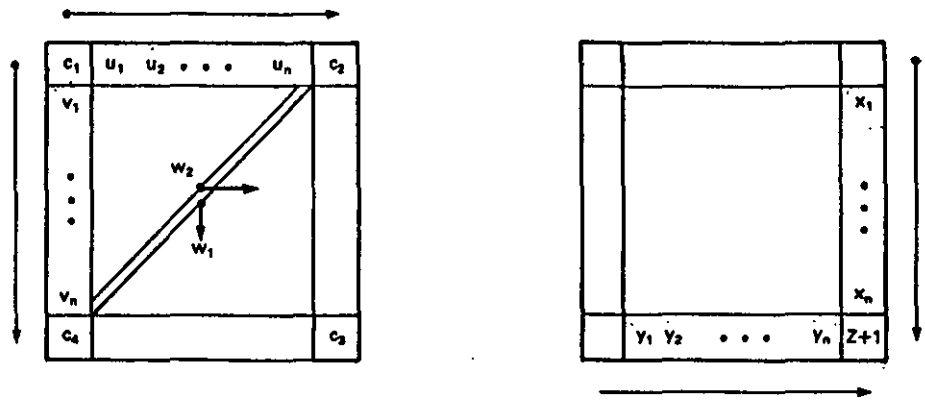
S = N+E*A " W2

A = A+S*E " W3

where $A=A_{ij}^{-1}$, and N,E,S,W are compass directions for input/output controlled by the SCR. Consequently the area of basic cells is bounded by the cost of an inner product cell plus some additional switching logic.

The results indicate that Rank Annihilation for a single update requires more cells and has an intermediate time between the arbitrary inversion schemes of Rote and Robert & Trystram [85]. From Corollary (4.3.1.1) it is evident that general inversion by rank annihilation does not compete with these arbitrary schemes. The explanation is simple and is illustrated by Fig.(4.3.1.2). A RANK-1 update generates only three wavefronts in 4n+5 cycles yielding a low processor efficiency, while the positioning of waves indicates that successive updates cannot be overlapped. In contrast, the arbitrary schemes are based on the Gauss-Jordan algorithm, and like Gaussian elimination in Fig.(3.2.2.2) have successive modifications pipelined to give high processor efficiency.

In an attempt to improve processor efficiency and consequently the



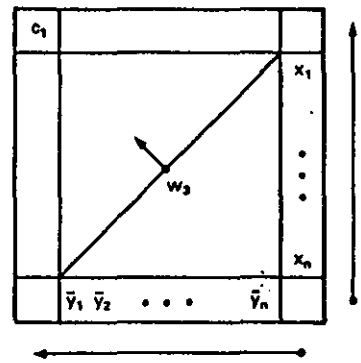
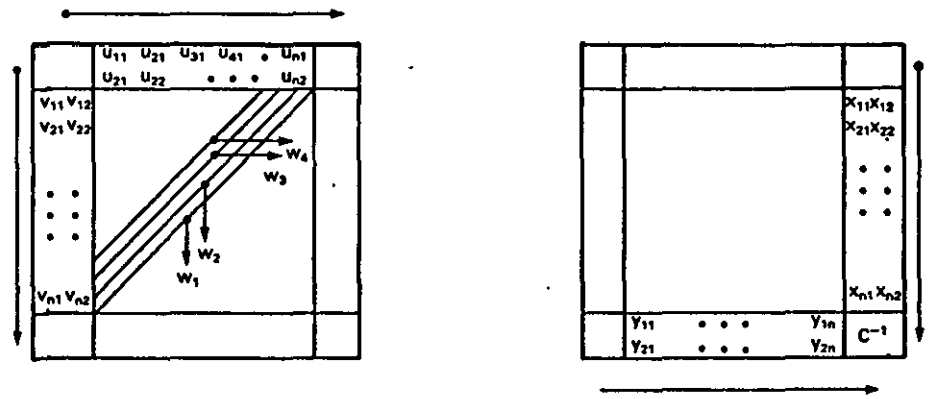


FIGURE 4.3.1.2a: RANK-1 Wavefronts



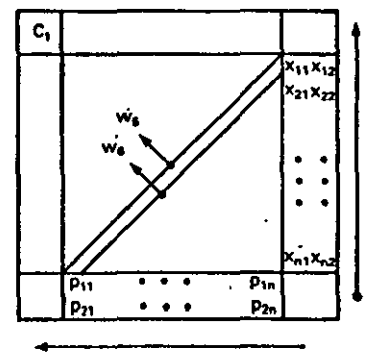


FIGURE 4.3.1.2b: RANK-2 Wavefronts

number of wavefronts generated in a complete circuit of the SCR, the RANK-2 scheme can be implemented on the same mesh with modified boundary cells. The u and v interface cells contain two registers instead of one, while the y-vector array swaps its subtract/divide for two multipliers and an adder and the x-vector array swap requires an extra ips. The controller C3 is no longer a simple logic unit, but implements Cramer's rule to compute the 2×2 inverse in parallel with six ips cells. Now bounding the Y-vector array cells by the cost of two ips cells gives a cell requirement of n^2+4n+6 cells for the mesh. More wavefronts can now be introduced by using a partitioned form of (4.3.6) where,

$$\left.\begin{array}{ll} \text{a)}\ A^{-1}u^{(1)} = x^{(1)} & \text{b)}\ A^{-1}u^{(2)} = x^{(2)} \\ \text{c)}\ v^{(1)T}A^{-1} = y^{(1)T} & \text{d)}\ v^{(2)T}A^{-1} = y^{(2)T} \end{array}\right\} \qquad (4.3.1.4)$$

and $U=[u^{(1)},u^{(2)}]$, $V^T=[v^{(1)T},v^{(2)T}]$, $X=[x^{(1)},x^{(2)}]$, $Y^T=[y^{(1)T},y^{(2)T}]$ A^{-1}, $u^{(1)}$, $u^{(2)}$, $v^{(1)}$ and $v^{(2)}$ are loaded into the mesh and the computation proceeds as follows:-

RANK-2 MESH CONTROL ALGORITHM:

STEP 1:

C1 starts the algorithm by pipelining four control signals along C1→C2 and C1→C4 on successive cycles. On cycle 1 signals entering interface cells u_i and v_i propagate $u_i^{(1)}$ down column i and zero along row i. These component waves constructively interfere producing a resultant wavefront W1 moving diagonally in direction C1→C3, and accumulating $x_i^{(1)}$ from left to right on row i. Similarly cycle 2 signals produce a wavefront W2 one cycle behind W1 accumulating $x_i^{(2)}$ on row i from component waves involving $u_i^{(2)}$. On cycle 3 the roles of u_i and v_i cells are reversed with cell v_i sending $v_i^{(1)}$ along row i and u_i propagating zero down column i producing wavefront W3 one cycle behind

W2 accumulating $y_i^{(1)}$ down column i. Likewise cycle 4 produces wavefront W4 from components $v_i^{(2)}$ and zeroes accumulating the $y_i^{(2)}$ values.

STEP 2:

After n+1 cycles, the first SCR controls reach C2 and C4. $x_i^{(1)}$ and $x_i^{(2)}$ have i and i+1 terms, and $y_i^{(1)}$, $y_i^{(2)}$ have i+2 and i+3 terms left to accumulate. Hence the signals associated with cycles 1 and 2 of step 1 are used to load $x_i^{(1)}$ and $x_i^{(2)}$ into the x-vector array, while cycles 3 and 4 are used to load $y_i^{(1)}$ and $y_i^{(2)}$ into the y-vector array and $v_i^{(1)}$,$v_i^{(2)}$ into the x-vector array. Consequently the signal leaving C2 on cycle n+5, is used to form the values (4.3.7), with a=1, b=0, c=0 and d=1 initially.

STEP 3:

After 2n+5 cycles the last control signals and the 2×2 matrix C have been collected by C3, which takes control computing C^{-1} by Cramer's rule in 2 ips cycles. Now on cycle 2n+8 C3 outputs controls along the SCR in directions C3→C4 and C3→C2. The signal on C3→C4 is accompanied by the first row of C^{-1} and Y-vector array cell i computes $OUT_i^{(1)} = c_{11}*y_i^{(1)}+c_{12}*y_i^{(2)}$ and propagates $OUT_i^{(1)}$ into column i. At the same time $x_i^{(i)}$ is pushed out into row i, and the two components for wavefront W5 moving in direction C3→C1. On the next cycle C3 issues a similar signal and y-vector cells compute $OUT_i^{(2)}=c_{21}*y_i^{(1)}+c_{22}*y_i^{(2)}$, $x_i^{(2)}$ and $OUT_i^{(2)}$ move along row i and column i one cycle behind W5 forming wavefront W6. The inverse elements are modified in two steps by W5 and W6 using

$$A_{ij}^{-1} = A_{ij}^{-1} - x_i^{(1)} OUT_j^{(1)}$$

and

$$A_{ij}^{-1} = A_{ij}^{-1} - x_i^{(2)} OUT_j^{(2)} \quad .$$

STEP 4:

On reaching C4 and C2 signals are sent back to C1 along C2→C1 and

C4→C1, and the interface section can be re-loaded with new u and v matrices, in parallel with the inverse modification. The last signals arrive back at C1 after 4n+10 cycles. STOP.

[End of RANK-2 algorithm].

Analogous arguments to the RANK-1 problem produce:

Theorem 4.3.1.2: r RANK-2 updates on the inverse A^{-1} of an n×n matrix A can be performed on an SCR wavefront mesh of $O((n+1)^2)$ cells in time $T=2(n+2)+r(4n+10)$, and,

Corollary 4.3.1.2: The inverse of an arbitrary n×n non-singular matrix A can be found by RANK-2 annihilation in $T=2n^2+7n+4$ ips cycles.

Proof:

Put r=n/2 and $A^{(0)}=I$ in (4.3.1.2) and update two rows at a time.

We conclude that RANK-2 is twice as fast as RANK-1 for k updates as r=k in Theorem (4.3.1.1) and $r=\lceil k/2 \rceil$ in Theorem (4.3.1.2). The intuitive idea behind rank annihilation is that successive updates of RANK-M will take less time than computing the full inverse of the modified matrix at each stage. Theorem (4.3.1.1) and (4.3.1.2) support this argument as we can write,

$$2(n+1)+r_1(4n+5) < 5r_1n \text{ , for } r_1 \geq 3 \qquad (4.3.1.5)$$

and

$$2(n+2)+r_2(4n+10) < 5r_2n, \text{ for } r_2 \geq 3 \qquad (4.3.1.6)$$

These timings result from the assumption that A^{-1} can be left inside the mesh from one update to another. A typical iterative process using rank annihilation, would use A^{-1} in a computation, determine u and v and then update A^{-1}. This requires A^{-1} to be loaded and unloaded on every update giving the revised relations,

$$r_1(6n+7) > 5r_1n \text{ , for all } r_1 \qquad (4.3.1.7)$$

and

$$r_2(6n+14) > 5r_2n, \text{ for all } r_2 \qquad (4.3.1.8)$$

These relations indicate that it is better to invert each of the $A^{(i)}$ in (4.3.1.2) rather than modify the already known inverse of $A^{(i-1)}$.

Remark: Corollaries (4.3.1.1) and (4.3.1.2) are still valid because general matrix inversion is a special case where each A^{-1} is not used outside the mesh.

4.3.2 Highly Pipelined Rank Annihilation

The above result seems contradictory and arises from the repeated loading/unloading of A^{-1}. A more satisfactory result would be achieved if input, output and updating could be overlapped, and the communication overhead of 2n cycles removed from the left hand sides of (4.3.1.7) and (4.3.1.8). This implies some kind of pipelined scheme capable of computing (4.3.5) and (4.3.6) in parallel. For RANK-1 we notice that (4.3.5b,c,d and e) comprise only of matrix-vector, outer product, and inner products respectively, and define the ordering of computations in the pipe. For instance, (4.3.5c and d) can be computed in parallel but must begin before (4.3.5b and e), which in turn must start before (4.3.5a). From this ordering the pipeline in Fig. (4.3.2.1) is derived for RANK-1 annihilation.

The matrix transpose matrix (m.t.m.) array performs (4.3.5c and d) on an array of 2n-1 special ips cells described below. Using the standard matrix vector array in Fig.(3.2.1.3) and assuming A^{-1} is full Fig. (4.3.2.2) illustrates the dataflow for solving Ax=y and $x^tA=y$. The cells in Fig.(4.3.2.2b) are obtained from the standard cell in Fig. (4.3.2.2a) by a rotation of 180° and a modification of the matrix input connection. Hence from Theorem (3.2.1.4), (4.3.5c and d) can be computed in parallel in 4n ips cycles using 4n-2 ips cells. Assuming

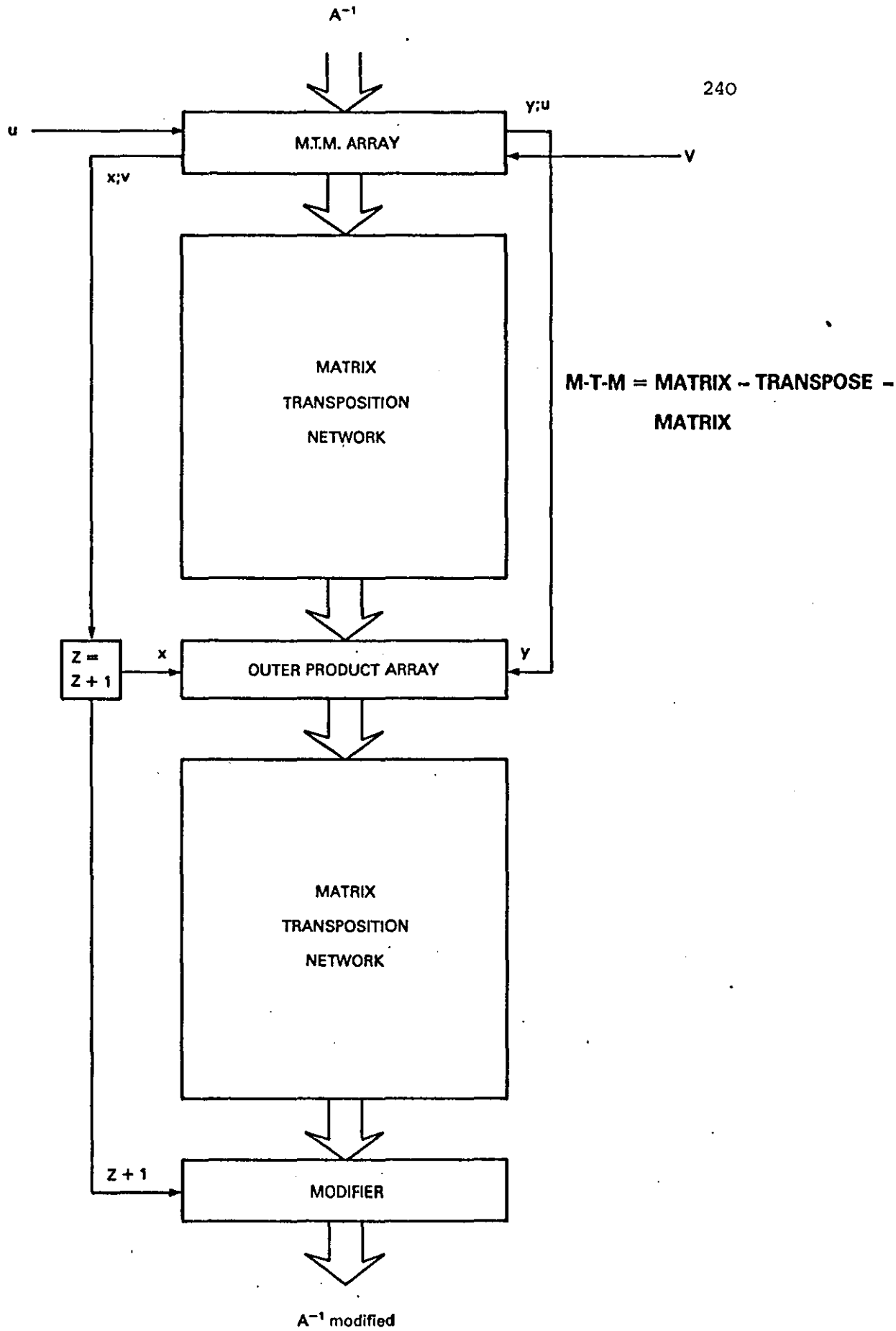
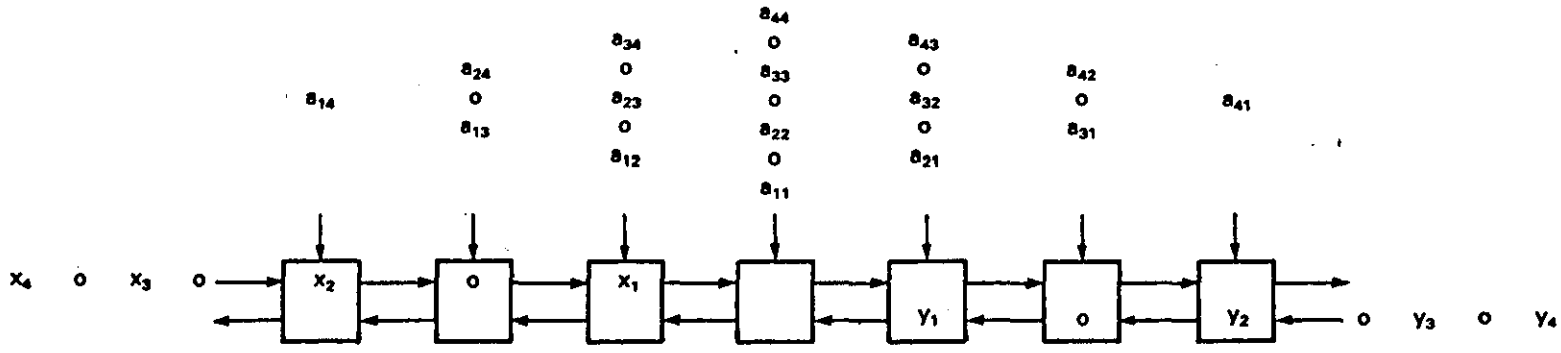
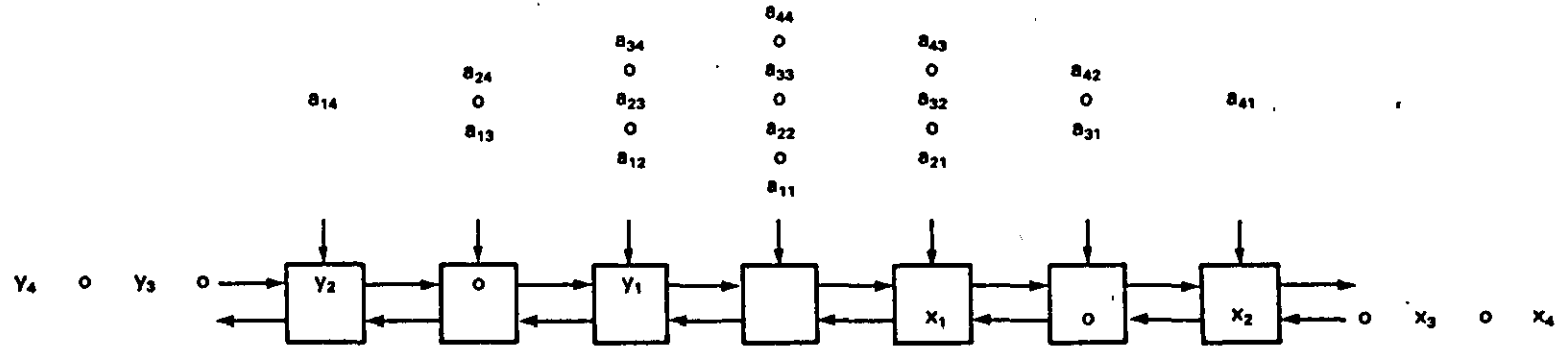


FIGURE 4.3.2.1: 'On the Fly' rank annihilation



a) **ARRAY COMPUTING Ax = y**



b) **ARRAY COMPUTING $x^tA = y^t$**

FIGURE 4.3.2.2: Matrix vector arrays for (n=4)

that A^{-1} is full (for generality) and pipelined between the two arrays to avoid bringing it from the host more than once. The number of cells could be reduced to 2n-1 if the data in Fig.(4.3.2.2b) was rotated about the axis formed from the a_{ii} diagonal inputs and interleaved with inputs of Fig.(4.3.2.2a). However we consider an alternative arrangement which amalgamates both arrays preserving the neutral elements of the A^{-1} input but filling the horizontal ones. This arrangement requires modifications to the internal structure of the cell, so that on one cycle it computes a term from (4.3.5c) and on the next a term from (4.3.5d). In the former case inputs are from the left and outputs to the right, for the latter the reverse is true. Consequently simple switching logic and a delay register to hold the matrix elements inside the cell for two cycles must be added to the basic ips cell. Switching can be implemented by a single control bit tagged to the matrix inputs and is of further value later. Fig.(4.3.2.3) illustrates the array operation and reveals the reason for the more complex cell design. The vectors x and v have been interleaved so that x_i and v_i for i=1(1)n are adjacent. Now z+1 can be easily computed by a simple accumulating inner product cell connected to the left end of the m.t.m. array, a control tagged to the x_1 value can be used to reset the cell to 1 forming z+1 in 2n ips cycles. This leaves only (4.3.5a,b) to calculate. (4.3.5b) is an outer product form which produces a matrix of the form

$$P = \begin{bmatrix} x_1y_1 & x_1y_2 & \cdots & x_1y_n \\ x_2y_1 & & & \\ \vdots & & & \vdots \\ x_ny_1 & x_ny_2 & \cdots & x_ny_n \end{bmatrix} \qquad (4.3.2.1)$$

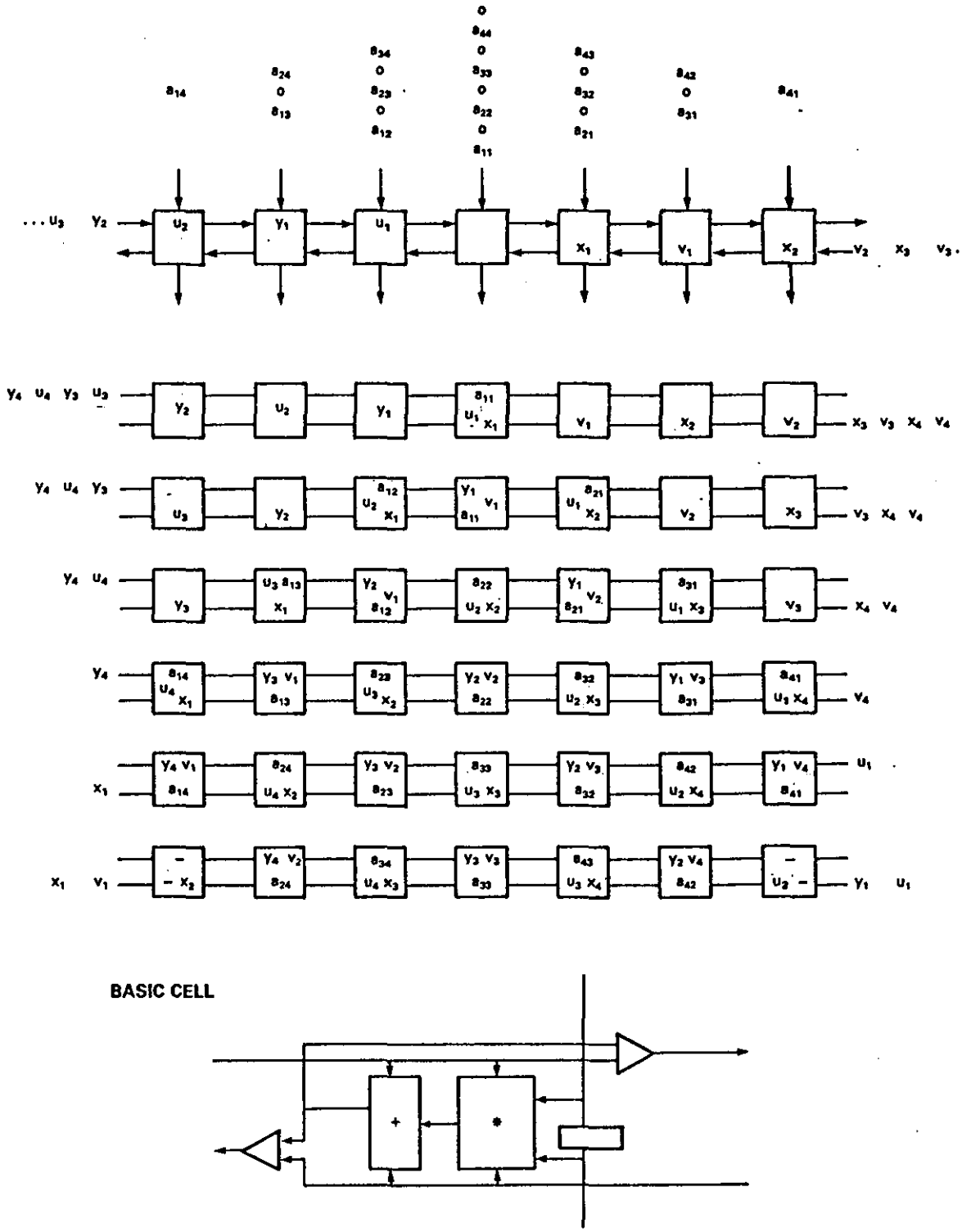
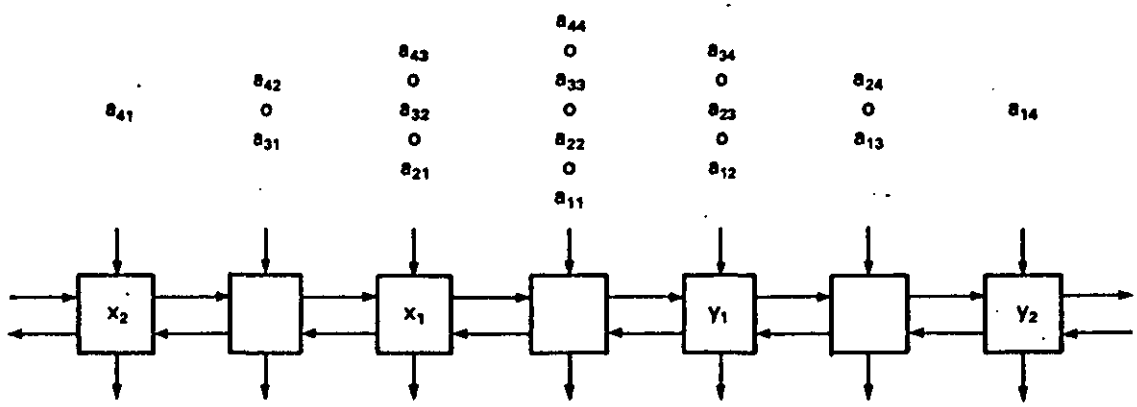


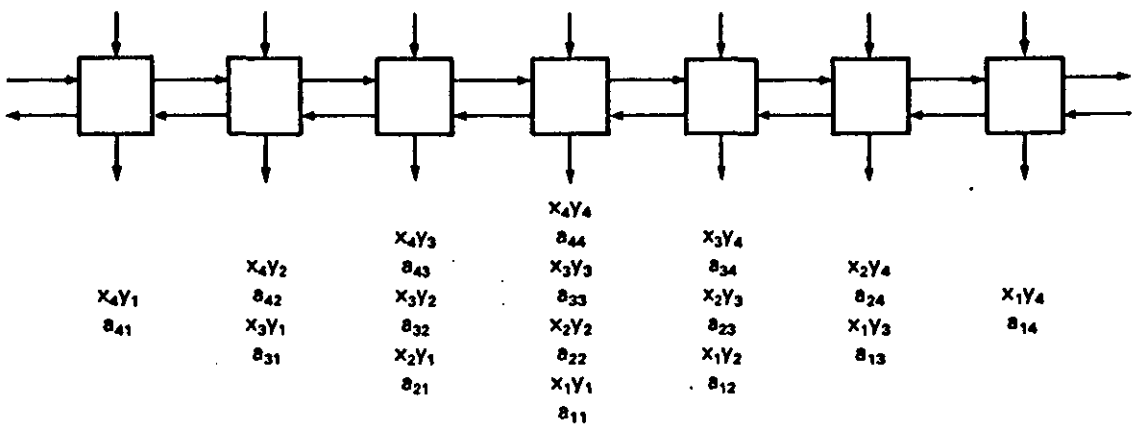
FIGURE 4.3.2.3: Systolic computation of $A^{-1}u=x$ and $v^{t}A^{-1}=y^{t}$ using interleaving

and can be computed by a linear array of 2n-1 multipliers in 2n cycles with a cell producing all the elements of a particular sub(super) diagonal. Notice that in order to compute (4.3.5a) the elements of A^{-1} and P must be locally placed with respect to each other. Ideally we would like to pipeline the A^{-1} input and results from the m.t.m. array into the solver for (4.3.2.1) and this is achieved by the arrangement in Fig.(4.3.2.4). We make use of the retained neutral elements in the A^{-1} data stream by filling them with the P_{ij} values. The control used for switching input and output directions in the m.t.m. can now be utilised again to overwrite the neutral value with the $x_i y_j$ result. The outer product array requires the values of x_i and y_j which must be synchronised with A^{-1}_{ij} to produce correct interleaved output of A^{-1} and P. The total time for x_1 after meeting a^{-1}_{11} in the m.t.m. to reach the centre cell of the outer product array by passing through the z+1 accumulating ips is 2n cycles, consequently a^{-1}_{11} must be delayed 2n-2 cycles between m.t.m. and outer product arrays to retain synchronisation. However, an added constraint is that the outer product generates P^T, when compared to A^{-1}, and the intermediate delay time must be utilised to form $(A^{-1})^T$, so interleaved elements remain adjacent. Clearly the transpose of the A^{-1} inputs is formed by a 180° rotation of elements about the main diagonal inputs (a^{-1}_{ii}), and can be achieved in a number of ways shown in Fig. (4.3.2.5). Intuitively the time for transposition is bounded by the number of cycles required to swap the outer sub(super) diagonal inputs. Assuming a diagonal element shifts its location by one diagonal per cycle transposition requires W cycles for a matrix of bandwidth W, and the area is bounded by W^2 exchange-delay cells. Thus, for A^{-1} full we require $(2n-1)^2$ cells and time 2n-1 cycles. Hence synchronisation in the outer

a) INPUT FORMAT

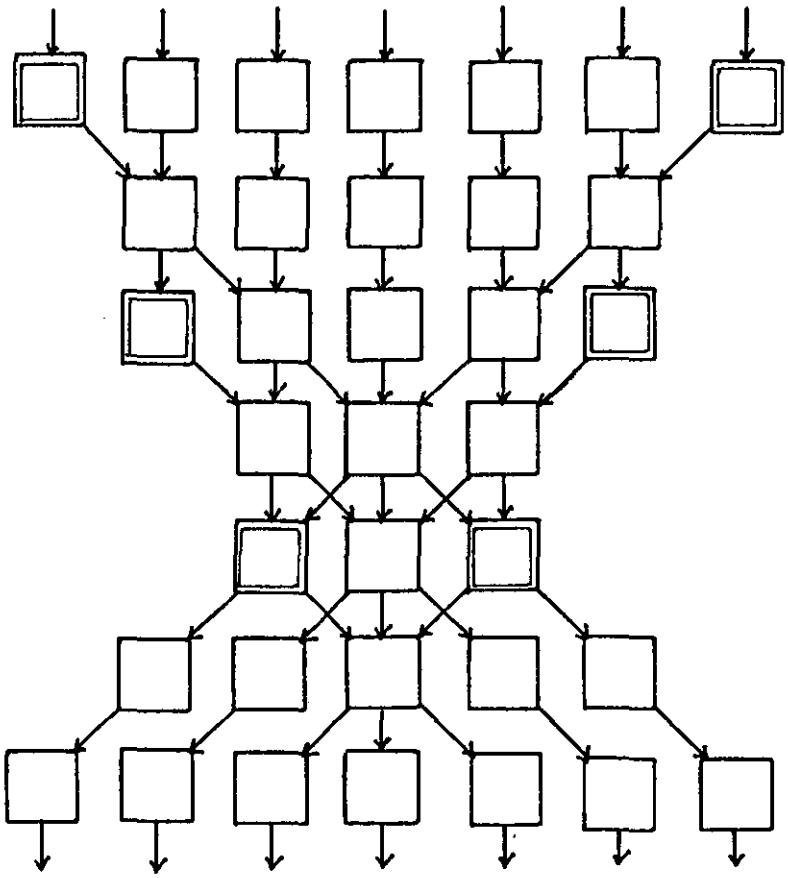


b) OUTPUT FORMAT

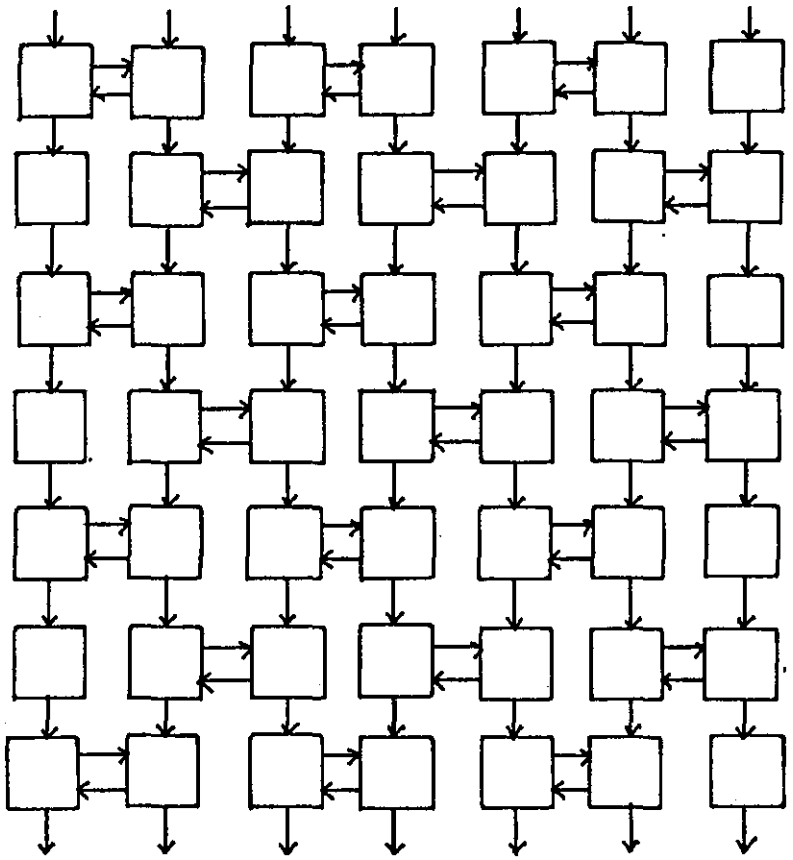


REMARK: BASIC CELL IS A MULTIPLIER WITH EXTRA CONTROL TO INSERT x_iy_j i,j = 1(1)n INTO VERTICAL DATA STREAM

FIGURE 4.3.2.4: Computation of $P=xy^t$



a) intuitive non-planar version



b) array from Ipsen [84]

FIGURE 4.3.2.5: Transposition networks

product array is achieved by placing a transposition network between m.t.m. and outer product arrays and adding an extra delay to the x_i data stream. As A is full the m.t.m. array is symmetrical and y_i values must also be delayed by a cycle.

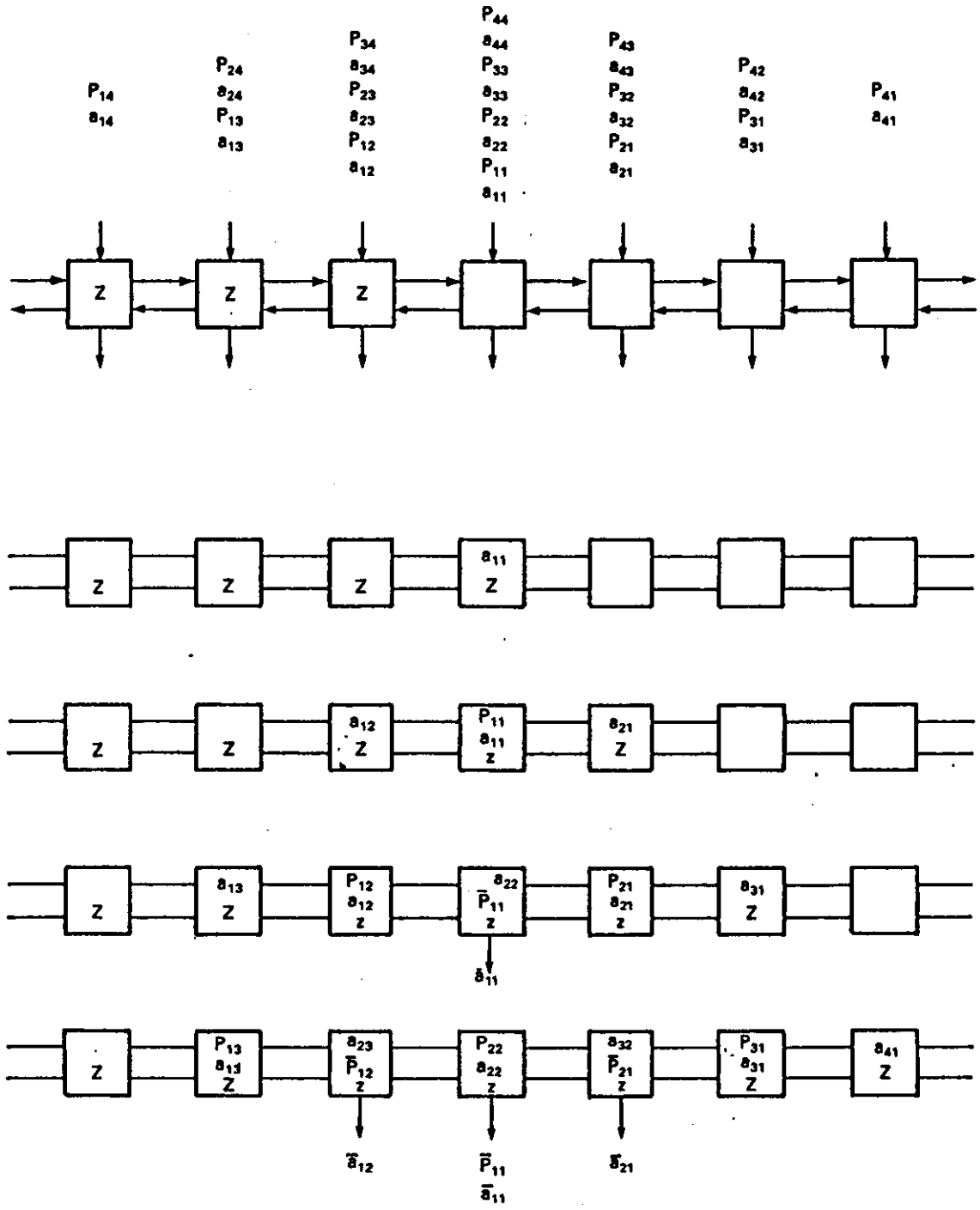
The only task remaining is the final modification of A^{-1} by (4.3.5a) requiring z+1, and the interleaved form of $(A^{-1})^T$ and P. z+1 accumulates one term every two cycles, so that by the time $(a_{11}^{-1})^T$ enters the outer product array, at most $\lceil n/2 \rceil + 1$ terms have accumulated. Now for z+1 and $(a_{11}^{-1})^T$ to meet in the centre cell of the modifier array of Fig.(4.3.2.1) as indicated by Fig.(4.3.2.6), a delay of at most 2n-1 cycles is required. n cycles to accumulate the rest of z+1 and n-1 cycles to filter z+1 to the centre cell of the modifier. This delay time can be utilised effectively by transposing the interleaved data, so that the final output is in the same form as input. The modifier array consists of 2n-1 divide and subtract cells with a loadable register to store z+1. A control value tagged to z+1 can be used to load the register and the control tagged to a_{ij}^{-1} elements to form $a_{ij}^{-1} = a_{ij}^{-1} - \frac{1}{(z+1)} P_{ij}$ as data passes through.

The timing of the array is then derived from the formula,

T = (start-up time) + (pipeline latency) + (output time)

From Fig.(4.3.2.3) the start-up time to synchronise A^{-1}, u and v is simply n-1 cycles. Pipeline latency is the total number of cycles required for a_{11}^{-1} to pass through the whole array, and is given by,

(i)	delay through	m.t.m. array	2 cycles
(ii)	"	outer product array	1 cycle
(iii)	"	modifier array	1 cycle
(iv)	"	transpose network	2n-1 cycles,



WHERE $\bar{A} = A - \frac{1}{z}P$

$\bar{P} = \frac{1}{z}(xy^t) = \frac{1}{z}P$

FIGURE 4.3.2.6: Systolic generation of new inverse

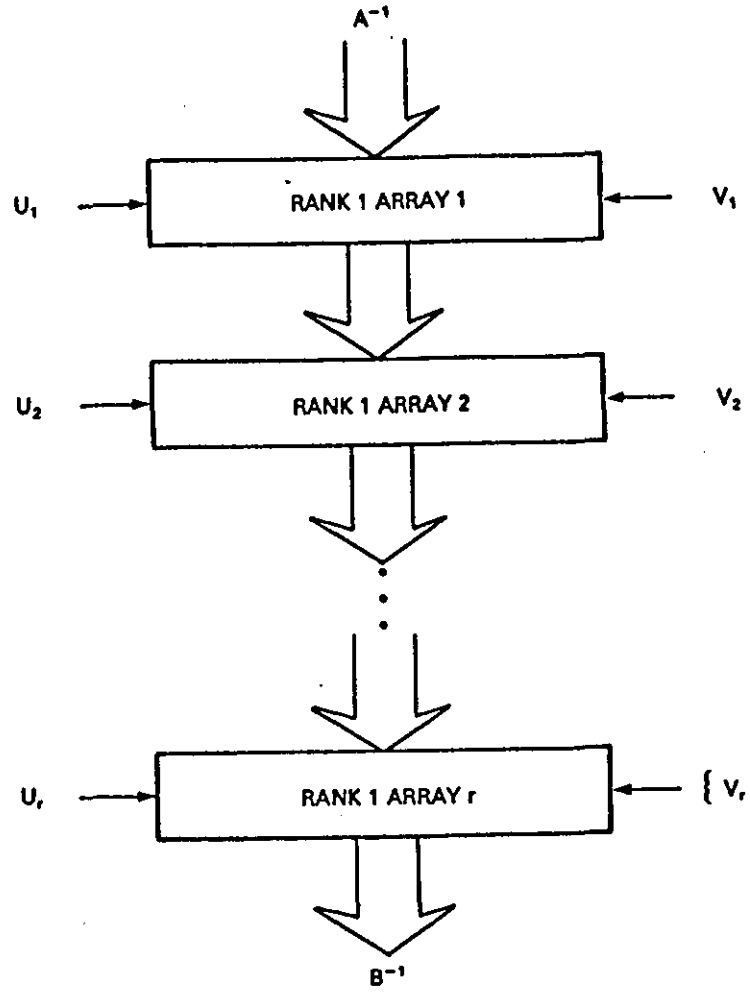


FIGURE 4.3.2.7: A cascaded RANK-1 scheme

giving a total of (4n-2)+4=4n+2 cycles. An extra 2n cycles is required to output all the modified matrix elements, yielding the result.

Theorem 4.3.2.1

The RANK-1 update of A^{-1} the inverse of an n×n matrix A can be performed on a systolic pipeline in T=(3n-1)+(4n+2) cycles. The design requires 3(2n-1)+1=6n-2 ips cells and $2(2n-1)^2$ delay cells from the transposition networks. Comparing this with Theorem (4.3.1.1) for r=1 indicates that the mesh scheme is faster for a single update. However the pipelined scheme uses only O(6n) true ips cells compared with $O(n^2)$ and the $O(n^2)$ delay cells permit a more compact design. Indeed, Fig. (4.3.2.1) admits a natural two layer design by folding the pipeline in half, and placing one transposition network on each level. Furthermore, the efficiency of cells in this new scheme is much improved. In the mesh

case, computation is charted by wavefronts with only those processors on the wavefront active giving poor efficiency. In the pipeline case, the m.t.m. and modifier arrays mimic the wavefront movements of W1, W2 and W3 respectively as A^{-1} pass through their elements producing high efficiency.

High throughput can also be achieved by overlapping computation of different problem instances, which is not possible with the mesh scheme. In fact when the u_i and v_i of (4.3.1.2) can be precomputed r copies of the pipelines can be cascaded to allow both problem instances and successive updates to individual cases to be overlapped. A drawback to cascading is that it increases hardware by a factor of r, which can be significant for large n or r. An alternative, which reduces throughput of distinct problem instances but uses constant hardware is a systolic ring. The ring is formed by noticing that pipeline latency is O(4n) and data length O(2n), as a result the modified inverse elements can be wrapped around to the pipeline inputs for the next update. Even with the ring only half the cells are used at any one time, and so two problems can be interleaved to achieve good cell efficiency. Hence,

Corollary 4.3.2.1:

r RANK-1 updates of A^{-1} the inverse of an n×n matrix A can be performed in T=(3n-1)+r(4n+2) cycles using a RANK-1 pipeline.

The RANK-1 pipeline is easily extended to RANK-2 updating, and for the sake of completeness we briefly outline its operation. Fig.(4.3.2.8) shows the global connection structure. The two m.t.m. arrays compute as shown in Fig.(4.3.2.9). The first array computes the first columns of X and Y, the second the last columns. The output results of M.T.M.1 are delayed by a single cycle to synchronise with M.T.M.2 results, allowing the first row of X to enter C on the same cycle. The locality

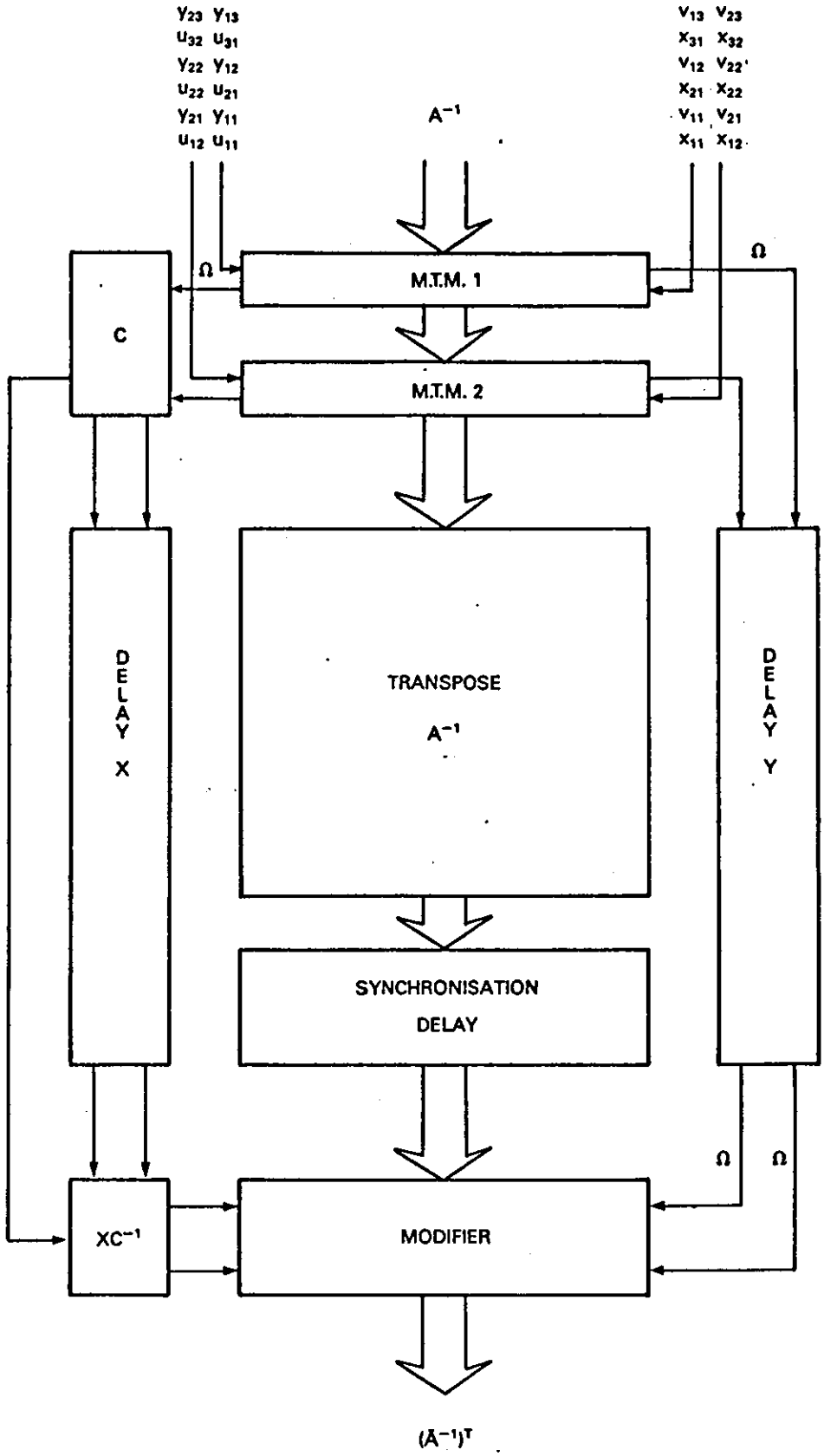


FIGURE 4.3.2.8: RANK-2 systolic inverter

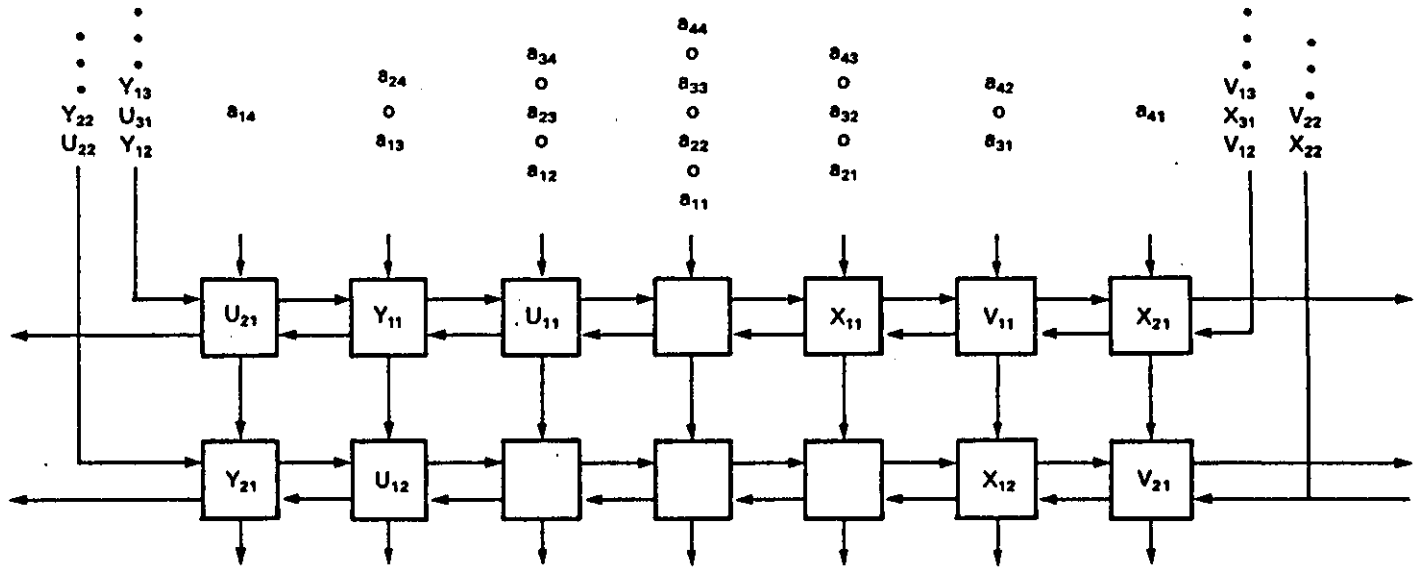


FIGURE 4.3.2.9: RANK-2 M.T.M.1 and M.T.M.2 arrays in starting positions

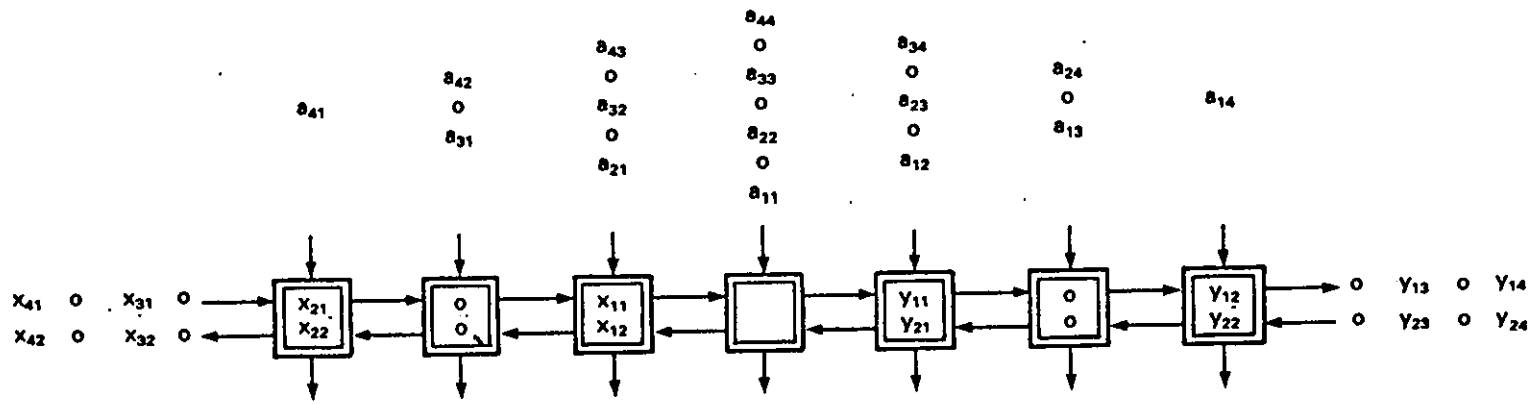


FIGURE 4.3.2.10: RANK-2 modifier array

of X and V is used again to compute (4.3.7) in cell C using four inner products and 2n cycles, with an extra two cycles added for C^{-1} to be computed before it is output to the XC^{-1} cell. Incoming X values are delayed a single cycle in C resynchronising them with Y, before they enter delay queues long enough for C^{-1} to be produced. On leaving its delay queue X is used in the XC^{-1} cell to compute the products,

$$\begin{aligned} P_{i1} &= x_{i1}c_{11} + x_{i2}c_{21} \\ P_{i2} &= x_{i1}c_{12} + x_{i2}c_{22} \end{aligned} \quad , \; i=1(1)n,$$

in a single cycle, using four multipliers and two adders, and delaying the Y values an extra cycle, for synchronisation. The matrix update

$$B_{ij} = A_{ij}^{-1} - [P_{i1}Y_{1j} + P_{i2}Y_{2j}] \quad , \; i,j=1(1)n,$$

is then made in the modifier array of Fig.(4.3.2.10) using a multiplier and subtracter in two ips cycles, with the dummy element in the A^{-1} data stream covering the extra computation time. In a similar manner to xy in the RANK-1 scheme PY^T produces a transposed output requiring formation of $(A^{-1})^T$ while C^{-1} is computed and P and Y^T filter into the modifier for synchronisation. Computation of C^{-1} starts n+1 cycles after a_{11}^{-1} leaves M.T.M.1, and requires 2(n+1) cycles, synchronisation of P and Y requires n cycles (allowing an extra cycle for delay in XC^{-1}). Thus, a total of 4n delays are required to synchronise a_{11}^{-1} with P and Y. Consequently the transposition is easily computed before a_{11}^{-1} reaches the modifier. Now allowing two cycles in the m.t.m. arrays and two cycles in the modifier the total delay through the pipe is 4(n+1), initial synchronisation requires n-1 cycles and output of modified inverse consumes 2n cycles hence,

Theorem 4.3.2.2

The RANK-2 update of the inverse A^{-1} of an n×n matrix A using a RANK-2 pipeline requires $T=(3n-1)+4(n+1)$.

Corollary 4.3.2.2:

r RANK-2 updates can be performed with a RANK-2 pipeline in $T=(3n-1)+4r(n+1)$.

The cell count is simply computed as follows:-

(i)	M.T.M. arrays	4n-2 ips
(ii)	C cell (at most)	8 ips
(iii)	XC^{-1}	4 ips
(iv)	modifier 2 ips per cell	(4n-2) ips

giving a total count of 8n+8 ips with 4n delay between m.t.m. and modifier 4n(2n-1) is an approximate count of delay registers neglecting X and Y delay queues.

In comparison with the RANK-1 scheme RANK-2 requires 2n+6 more ips cells and at least 4n additional delay registers, but modifies two rows or columns in a single update for the loss of only two cycles. The RANK-2 pipe retains the advantages of high efficiency and throughput, but loses the natural partitioning on to two layers. The systolic ring arrangement is also move involved because the modifier outputs the updated matrix in transposed form requiring u and v inputs to be switched on alternate ring cycles.

4.3.3 Choice of Schemes

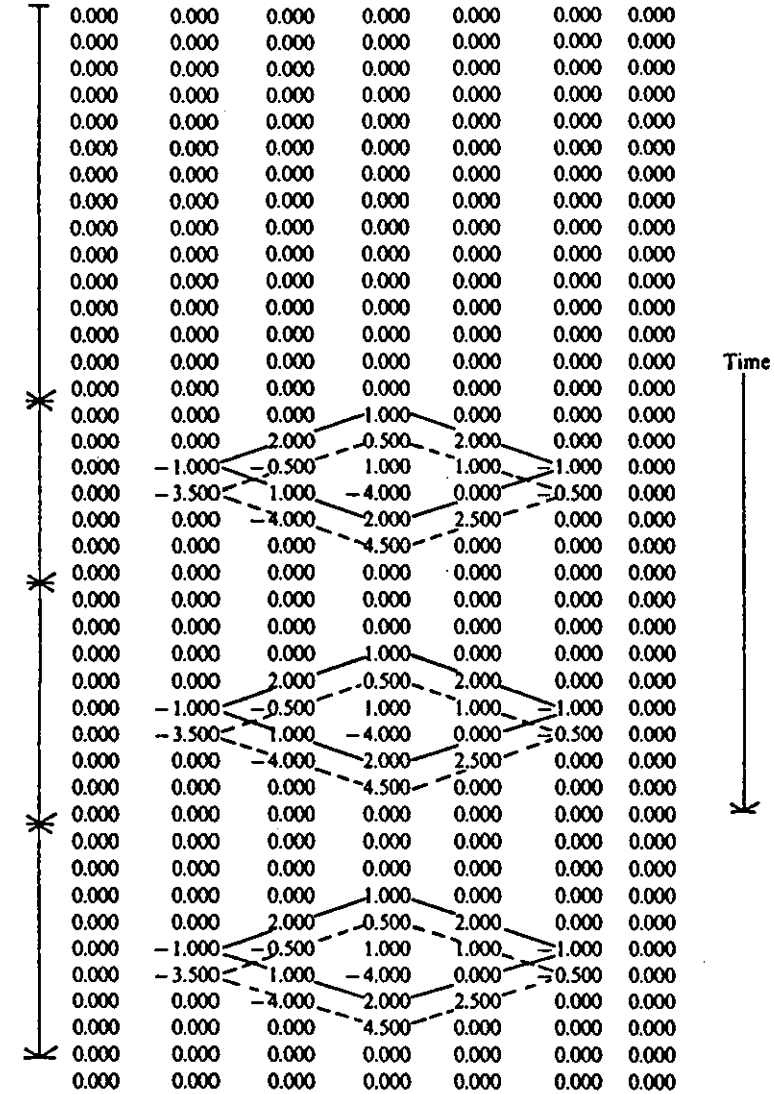
The choice of scheme depends upon whether a general matrix inverse A^{-1} is required or an update to A^{-1} based on local changes in A.

For a general inverse the sequences of modification encoded by u

and v can be precomputed because changes in A occur on distinct rows, and the intermediate updates are not incorporated in a larger computational process. Consequently mesh schemes do not have to be repeatedly unloaded and the ring pipeline schemes can be adopted. The general inversion methods based on Gauss-Jordan permit much more pipelining of successive computations producing O(5n) time which out performs the rank-annihilation schemes using $O(n^2)$ time. From the view point of area, general schemes use $O(n^2)$ cells highly efficiently, while wavefront mesh schemes use $O(n^2)$ inefficiently. In contrast, the rank pipeline systolic rings use O(n) cells and $O(n^2)$ registers with high efficiency, but do not compensate for the increased time.

In the cases when a large number of matrix updates are made, where modification data is derived by using the modified inverse of the previous step, choice of array must be made by comparing the time of a single update by annihilation with that of general inversion. The time trade-off is O(5n)-O(7n) for arbitrary inversion, O(6n) for mesh annihilation and O(7n) for pipelined inversion, giving a much better time relationship. The lack of cell efficiency would favour arbitrary methods over mesh schemes; but reduced cell count and improved throughput of pipelined schemes is preferable to general inversion schemes as two independent updates can be performed simultaneously. (The general scheme requires O(10n) for two updates). Finally the systolic ring idea is easily extended to incorporate more components for an iterative process where computation involving A^{-1} and u,v generation could also be pipelined. The increased delay may permit more problem instances to occupy the ring simultaneously.

Pipelining of separate problem instances.



Output of modified and original inverses interleaved.

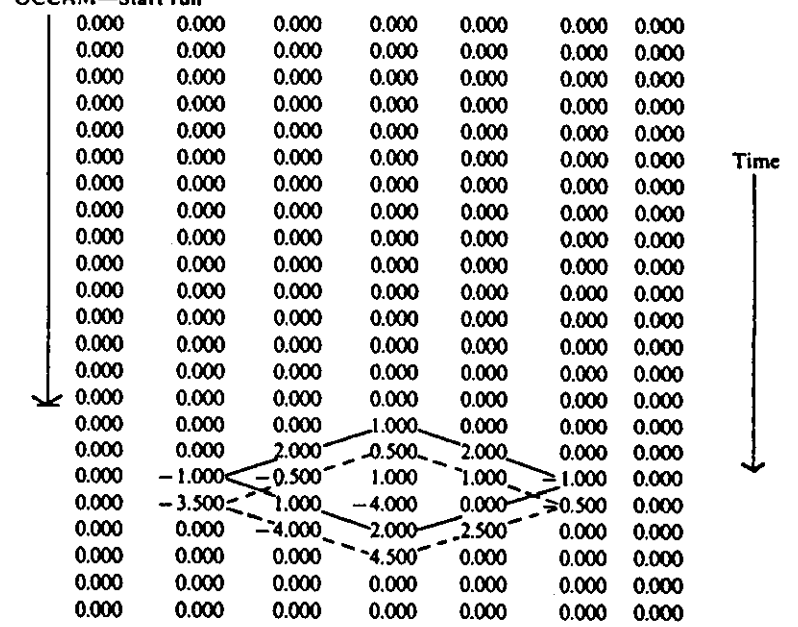


FIGURE 4.3.2.11: Example OCCAM runs

Finally we conclude this section with some remarks about implementation. Fig.(4.3.2.1) incorporates long wires of length proportional to n making the design soft-systolic under current technology. Use of waveguides for optical data transmission, and multi-layers make the RANK-1 pipeline an attractive design, and program 18 in the Appendix is an OCCAM simulation of Fig.(4.3.2.1). The program was tested on a variety of cases and a simple 3×3 example is given below.

Let

$$A = \frac{1}{9}\begin{bmatrix} -2 & 5 & -1 \\ 4 & -1 & 2 \\ -3 & 3 & 3 \end{bmatrix}, \quad A^{-1} = \begin{bmatrix} 1 & 2 & -1 \\ 2 & 1 & 0 \\ -1 & 1 & 2 \end{bmatrix}$$

and put

$$B = \frac{1}{9}\begin{bmatrix} 16 & 14 & 26 \\ 4 & -1 & 2 \\ -3 & 3 & 3 \end{bmatrix}, \quad B^{-1} = \begin{bmatrix} 0.5 & -0.5 & -3.5 \\ 1 & -4.0 & -4.0 \\ -0.5 & 2.5 & 4.5 \end{bmatrix}$$

then $u=(1,0,0)^t$, $v^t=(2,1,3)$ in (4.3.1). The results of the program output are shown in Fig.(4.3.2.11) verifying theorem (4.3.2.1), and indicating that k successive problem instances require $T=(n-1)+(4n+2)+k(2n+3)$ cycles.

4.4 BATS: A BANDED AND TOEPLITZ SYSTEM SOLVER

Section 4.1 considered the X-band matrix vector multiplication problem and the use of D^2-pipes to take advantage of sparsity. In this section we consider the solution of a linear system like (4.1.11) with an X-band coefficient matrix. X-band system solution is a non-trivial task because both triangularisation and factorisation techniques allow

fill-in of zero elements which destroy sparsity and increase bandwidth. As already noted the solution of P.D.E.'s with periodic boundary conditions using finite difference techniques leads to an X-band form matrix, i.e.,

$$A_c = \begin{bmatrix} a_0 & a_1 \cdots a_p & & & a_p \cdots a_1 \\ a_1 & & & & \\ \vdots & & & & \\ a_p & & & & a_p \\ & & & & \vdots \\ & & & & a_1 \\ a_p & & O & & \\ \vdots & & & & \\ a_1 & a_p & & a_p & a_0 \end{bmatrix}_{n\times n} \qquad (4.4.1)$$

Factorising this matrix produces fill-in along the last p rows of the L factor and last p columns of the U factor and a hexagonal systolic array Fig.(3.2.2.3) must treat A_c as a full matrix. Consequently the solution of Toeplitz systems such as circulant, symmetric and skew-symmetric forms like (4.4.1) have received much attention. The most successful designs to date from a systolic view point are the PLP (S.Y. Kung & Hu [83]) and a linear array of Brent and Luk [83]. Both designs have O(n) time and the latter is also sensitive to band structure and can solve both symmetric and skew symmetric forms. The PLP consisting of three tiers of n inner product cells each, and a LIFO structure of $O(n^2)$ cells, is based on Levinsons algorithm and requires O(2n) time. The linear array is based on the Bariess algorithm and consists of $\left\lceil \frac{n}{2} \right\rceil$ super cells each with a control algorithm and requires O(4n) time with six ips per cell and is essentially soft-systolic.

In contrast our technique is based on new factorisation methods developed in Chen [85] and Audish & Evans [85a,b] and permits the

pipelining of solutions to more than one system through the array. The previous schemes cannot pipeline successive instances and for a dedicated Toeplitz solver our method should improve throughput significantly.

We consider the solution of the n×n system,

$$A_c x = f, \tag{4.4.2}$$

where $x=(x_1,x_2,\ldots,x_n)^t$ is unknown and f is the known right hand side. Chen [85] shows that if A_c is strictly diagonally dominant it can be factorised into the form,

$$A_c = \beta_0^{-1} \tilde{L} \tilde{L}^T , \tag{4.4.3}$$

where,

$$\tilde{L} = \begin{bmatrix} \beta_0 & & & & \beta_p & \cdots & \beta_1 \\ \beta_1 & \ddots & & & & \ddots & \vdots \\ \vdots & & \ddots & O & & & \beta_p \\ \beta_p & & & \ddots & & & \\ & \ddots & & & \ddots & & \\ & O & \ddots & & & \ddots & \\ & & & \beta_p & \cdots & \beta_1 & \beta_0 \end{bmatrix}, \tag{4.4.4}$$

(4.4.2) is then solved by the coupled systems,

$$\text{a)}\quad \tilde{L}y = d, \qquad \text{b)}\quad \tilde{L}^T x = y , \tag{4.4.5}$$

where $d=\beta_0 f$.

Now put,

$$L = \begin{bmatrix} \beta_0 & & & & & \\ \beta_1 & \ddots & & & O & \\ \vdots & & \ddots & & & \\ \beta_p & & & \ddots & & \\ & \ddots & & & \ddots & \\ & O & & \beta_p & \cdots \;\; \beta_1 & \beta_0 \end{bmatrix}_{n\times n} \text{and } R = \begin{bmatrix} \beta_p & \cdots & \beta_1 \\ & \ddots & \vdots \\ O & & \beta_p \end{bmatrix}_{p\times p} \tag{4.4.6}$$

then,

$$\tilde{L} = L + \begin{bmatrix} I_p \\ O \end{bmatrix} R [O^T \; I_p] \quad , \tag{4.4.7}$$

where I_p = pth order identity matrix, and O the (n-p)*p null matrix.

We now apply the rank annihilation formula (4.3.4) to yield,

$$\tilde{L}^{-1} = L^{-1} - L^{-1} \begin{bmatrix} R \\ O \end{bmatrix} \left\{ I + [O^T \; I_p] L^{-1} \begin{bmatrix} R \\ O \end{bmatrix} \right\}^{-1} [O^T \; I_p] L^{-1} \quad , \tag{4.4.8}$$

and the coupled system (4.4.5) is solved explicitly viz,

$$\text{a)} \quad y = \tilde{L}^{-1} d \qquad \text{and} \quad \text{b)} \quad x = \tilde{L}^{-T} y \quad . \tag{4.4.9}$$

The method extends easily to the simple banded Toeplitz matrix A_t where,

$$A_t = A_c - \begin{bmatrix} I_p \\ O \end{bmatrix} U [O^T \; I_p] - \begin{bmatrix} O \\ I_p \end{bmatrix} U^T [I_p \; O^T] \quad , \tag{4.4.10}$$

and the corresponding linear system,

$$A_t x = f \quad , \tag{4.4.11}$$

is given by,

$$x - A_c^{-1} \begin{bmatrix} I_p \\ O \end{bmatrix} U [O^T \; I_p] x - A_c^{-1} \begin{bmatrix} O \\ I_p \end{bmatrix} U^T [I_p \; O^T] x = A_c^{-1} f \quad , \tag{4.4.12}$$

$$\text{or} \quad x = y + B_1 U x^{(3)} + B_3 U^T x^{(1)} \quad , \tag{4.4.13}$$

with,

$$y = A_t^{-1} x, \quad B_1 = A_c^{-1} \begin{bmatrix} I_p \\ O \end{bmatrix}, \text{ and } B_3 = A_c^{-1} \begin{bmatrix} O \\ I_p \end{bmatrix} \quad ,$$

where $x^{(1)} = (x_1, \ldots, x_p)^T$, $x^{(2)} = (x_{p+1}, \ldots, x_{n-p})^T$ and $x^{(3)} = (x_{n-p+1}, \ldots, x_n)^T$ and U is a pth order matrix like R but with elements $a_1, \ldots, a_p$. The solution is then determined by premultiplying (4.4.13) by (I_p, O^T) and (O^T, I_p) to produce the system,

$$\left.\begin{aligned} (I_p - M_{1p} U^T) x^{(1)} - M_{11} U x^{(3)} &= y^{(1)} \\ -M_{11} U^T x^{(1)} + (I_p - M_{1p}^T U) x^{(3)} &= y^{(3)} \end{aligned}\right\} \tag{4.4.14}$$

where M_{11} and M_{1p} are the pth order submatrices of A_c^{-1} at the northwest and northeast corners. We then find $x^{(1)}$ and $x^{(3)}$ using (4.4.14) and $x^{(3)}$ with (4.4.13). This latter scheme is more computationally complex than existing systolic schemes for A_t matrix structures but shows that the solution of A_t is simply a linear combination of the solution to (4.4.2) and the first and last p-columns of A_c^{-1}.

4.4.1 A Pipelined Solver

A solution to (4.4.2) can be constructed by a simple pipeline arrangement illustrated in Fig(4.4.1.1), and consists of a triangular inverter for finding L^{-1} and L^{-T}, a rank annihilation pipeline for (4.4.8) and a matrix vector array for the coupled systems (4.4.9). The only new component is the inverter and its operation requires some explanation.

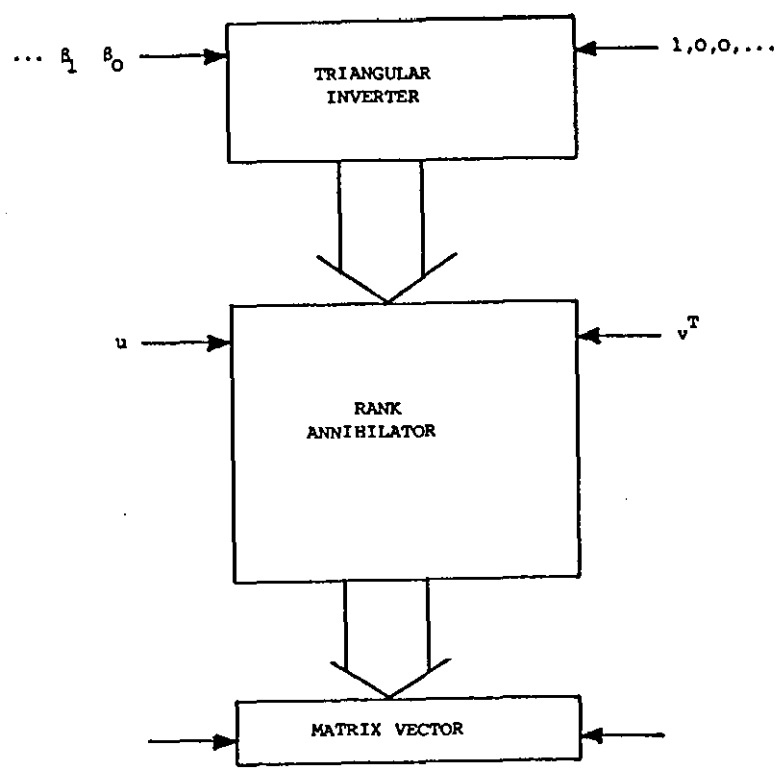


FIGURE 4.4.1.1: Pipelined Topelitz solver

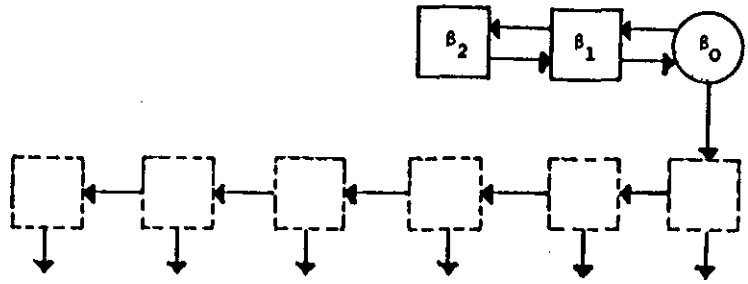
The inverter itself comprises two back to back triangular inverters, one on the left producing upper triangular inverses (or L^T) and one on the right for lower triangular inverses. These two components are operated in mutually exclusive fashion so no conflicts occur in later stages of the pipe and make use of the special form of A_c. Since A_c is a symmetric circulant matrix so is A_c^{-1} and is uniquely defined by its first column hence L^{-1} is found by,

$$L\gamma = (1,0,0,\ldots,0)^T , \qquad (4.4.1.1)$$

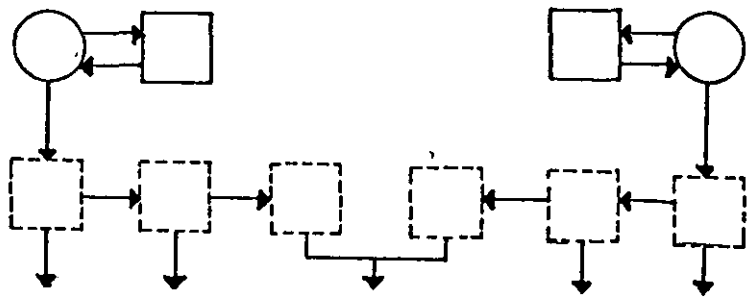
and can be computed on a p-cell backsubstitution array in T=2n+p cycles, producing the solution sequence γ_0 o γ_1 o ... o γ_{n-1}. However, the rank annihilation array accepts input in standard diagonal format,

$$\begin{array}{ccccccccccccc}
 & & & & & & \gamma_0 & & & & & & \\
 & & & & 0 & - & \gamma_1 & & & & & & \\
 & & & 0 & - & \gamma_0 & - & \gamma_2 & & & & & \\
 & & 0 & - & 0 & - & \gamma_1 & - & \gamma_3 & & & & \\
 & 0 & - & 0 & - & \gamma_0 & - & \gamma_2 & - & \gamma_4 & & & \\
 & & 0 & - & 0 & - & \gamma_1 & - & \gamma_3 & & & & \\
 & & & 0 & - & \gamma_0 & - & \gamma_2 & & & & & \\
 & & & & 0 & - & \gamma_1 & & & & & & \\
 & & & & & & \gamma_0 & & & & & &
\end{array} \qquad (4.4.1.2)$$

for L^{-1} and n=5, and a transposed form for L^{-T}. Each inverter component is a bi-linear array, shown in Fig.(4.4.1.2). The top tier is a modified backsubstituter which preloads the constant diagonal values β_i, i=0(1)p, and the bottom tier contains n cells to generate the non-zero portion of the above diamond input. As the don't care slots ('-') can be replaced by the neutral element zero, and the components are operated in mutually exclusive fashion. The off-state of a component generates the zero side of the input automatically.



a) Lower triangular inverter (n=6, p=2)

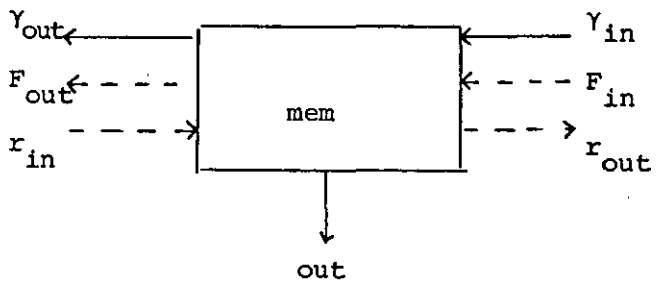


b) Two back-to-back components (n=3,p=1)

FIGURE 4.4.1.2: Triangular inverter

Second tier generating cells consist of loadable registers and simple control logic as defined below.

Cell



Procedure

mem=0

If r_{in} THEN{mem=γ_{in},toggle=TRUE}

If F_{in} THEN mem=0

If toggle THEN out=mem

ELSE out=0

$F_{out}=F_{in}$, $r_{out}=r_{in}$, $\gamma_{out}=\gamma_{in}$

Toggle:=NOT Toggle

As the backsubstituter produces the γ_i values, they are pumped from right to left along the second tier. Associated with and travelling one cycle in front of, γ_0 is a control value (F). This forward control signal on the right to left journey resets cells by putting mem=0, and when F drops off the left end of the array it is immediately input as

a return signal (r). By virtue of the γ_i spacing and its lead on γ_0, r meets each γ_i, i=0(1)n-1, as it moves left to right loading them into generating cells, and locking the cell into an alternating output cycle (see Fig.4.4.1.3). On reaching the rightmost cell, r loads γ_{n-1} which is output only once. Consequently on leaving the array r is pumped back in as F to reset cells forming the remaining input pattern of (4.4.1.2). The control travels in cyclic fashion forming a control ring and indicates that the next solution sequence can be fed right to left along the second tier while the current inverse is still being output. If we allow 2(p+1) additional cycles to load the new parameters into the first tier and output the first result $\bar{\gamma}_0$, the reset control is 2(p+1) cycles in front, causing erroneous loading of the next inverse coefficients. Hence the overlapping of different instances on the same inverter component requires a modified control arrangement. A working version of this modified form is given in the Appendices (program 19) and yields the following theorem.

Theorem 4.4.1.1:

The lower triangular inverter of a symmetric Toeplitz matrix L of bandwidth p requires T=3n+2(p+1) cycles to generate a diagonal input format.

Proof:

We require p+1 cycles to load the parameters representing L, and a further p+1 cycles for the first result to emerge from tier 1 of Fig.(4.4.1.2). This first result requires n cycles to filter through the second tier to its correct position. As the element corresponds to the diagonal and hence longest data sequence in (4.4.1.2) containing 2n elements, T=2n+n+2(p+1) follows immediately.

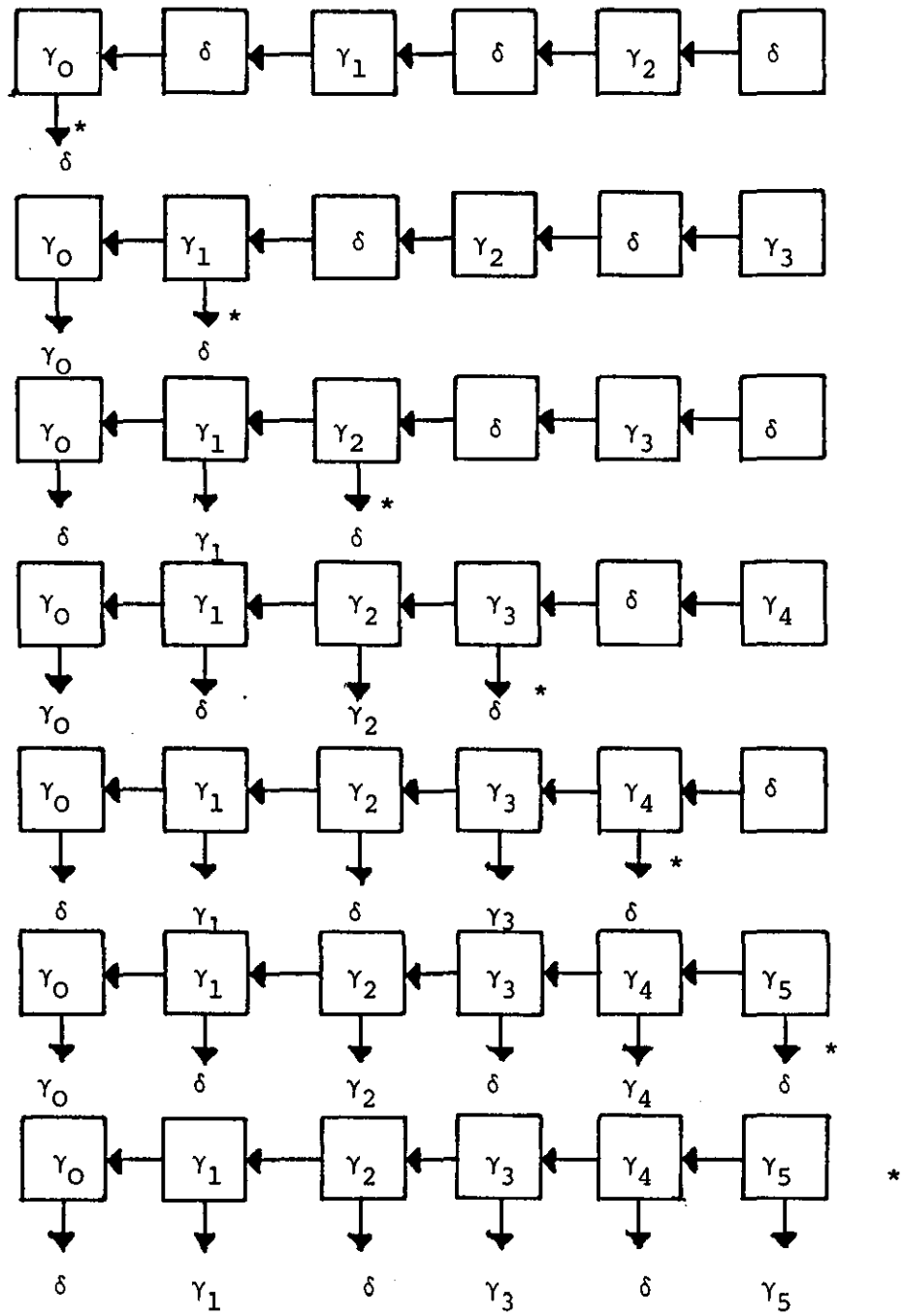


FIGURE 4.4.1.3: Snapshots of inverse output generation

* = control signal.

Corollary 4.4.1.1: Successive matrix outputs of the same inverter components are separated by at least 2(p+1) cycles.

Proof:

Computation in tier 1 is complete after 2(n+p+1) cycles, and tier two cells begin to switch off. Allow 2(p+1) cycles for paramater loading and the computation of the first result of the next instance on tier 1, switch off controls are 2(p+1) cycles in front of the new data. As the leftmost cell of tier 2 is the first and last to output, successive matrix diamond patterns must be separated by 2(p+1) cycles. This completes the description of the inverter.

We can now consider the operation of the pipeline in Fig.(4.4.1.1). Initially the parameters β_i, i=0(1)p are loaded into the right component of the inverter and after n+2(p+1) cycles L^{-1} begins to emerge. These values are pipelined onto the rank annihilator which for simplicity is assumed to be a RANK-1 pipe with a ring capability. Converting L^{-1} to $\tilde{L}^{-1}$ requires p column updates to distinct columns and the vectors u_i and v_i corresponding to (4.3.1.2) can be precomputed. It follows that $\tilde{L}^{-1}$ is computed by p cycles around the ring requiring p(4n+2) cycles using corollary (4.3.2.1) and the fact that initial synchronisation and output are overlapped with other pipe segments. Finally $\tilde{L}^{-1}$ is pipelined onto the matrix vector array to produce (4.4.9a). Now using the left component of the inverter L^{-T} and its first update computations can be overlapped with the last modification of $\tilde{L}^{-1}$ producing an additional time of (p-1)(4n+2) cycles to complete $\tilde{L}^{-T}$. Feeding y back into the matrix vector array to synchronise with $\tilde{L}^{-T}$ produces the result (4.4.9b).

Theorem 4.4.1.2:

The solution of k n×n circulant symmetric matrix systems of the

form $A_c x=f$ with semi-bandwidth p+1 is computed on a Toeplitz solver in $T=(6-4k)n+8kp(n+1)+2$ cycles.

Proof:

$$T = T_1 + T_2 + T_3 \qquad (4.4.1.3)$$

where T_1 = initialization and output latency delays

T_2 = total length of input/output sequence

T_3 = additional delays spent cycling in rank annihilator.

Now $T_1=2(n+p+1)$ as the inverter requires $n+2(p+1)$ to produce the first element of the first L^{-1}, and the first output is delayed by n cycles on its way out of the matrix vector array.

There are k systems and allowing for the inverter delay, each one is represented by two diamond forms like (4.4.1.2) of length 2n separated by 2(p+1) cycles. One diamond represents L^{-1} the other L^{-T}. Thus, a single system has input length 4n+2(p+1) generated by the inverter. Furthermore to retain synchronisation each system must be separated by 2(p+1) cycles. Thus the total input length to the rank annihilator is,

$$T_2 = 2k(2n+p+1) + 2(k-1)(p+1) \ . \qquad (4.4.1.4)$$

Now for a semi-bandwidth p+1 the rank annihilator performs p modifications to L^{-1} and L^{-T} of each system. Cycles of the rank annihilator introduce additional delays effectively lengthening the input incident on the matrix vector array. For the first system as described above this delay is $p(4n+2)+(p-1)(4n+2)=(2p-1)(4n+2)$. Using the fact that the production of $\tilde{L}^{-T}$ of the ith system can be overlapped with $\tilde{L}^{-1}$ of the (i+1)th system for subsequent solutions add a delay $2(p-1)(4n+2)$ each hence,

$$T_3 = (2p-1)(4n+2)+2(k-1)(p-1)(4n+2) \ , \qquad (4.4.1.5)$$

forming the summation (4.4.1.3) and some algebraic manipulation produces the theorem time.

Corollary 4.4.1.2: The solution of a single n×n circulant symmetric matrix system $A_c x=f$ of semi-bandwidth p+1 requires $T=2(4p+1)(n+1)$ cycles using the Toeplitz solver.

Proof: Use k=1 in Theorem(4.4.1.2).

Further improvements to these timings are possible by using a RANK-2 pipeline and by noticing that for p>1 the 2(p+1) delay associated with the inverter can be overlapped with the last update in the rank annihilator. An alternative scheme is to try and interleave the computation of $\tilde{L}^{-1}$ and $\tilde{L}^{-T}$ using the fact that a cycle length is 4n+2 cycles and data length is 2n; thereby halving the delay associated with the rank annihilation. But from Corollary (4.4.1.1) the input diamonds of L^{-1} and L^{-T} are separated by 2(p+1) cycles giving a combined input length of 2n+2(p+1) cycles. Hence even with p=1 interleaving is not possible.

Now p=1 is an interesting problem because A_c becomes a circulant tridiagonal and only a single update is required to L^{-1} and L^{-T}. Consequently terms in (4.4.1.4) and (4.4.1.5) associated with cycling disappear, improving throughput and decreasing computation time. These attributes can be retained for general A_c bandwidths by considering an alternative factorisation method due to Audish and Evans [85b]. The idea is to factorise A_c such that,

$$A_c = Q_1, Q_2, \ldots, Q_p, Q_p^T, \ldots, Q_1^T \quad , \qquad (4.4.1.6)$$

where,

$$Q_i = \begin{bmatrix} 1 & & & \alpha_i \\ \alpha_i & 1 & & \\ & & \ddots & \\ & & & 1 \end{bmatrix} \quad , \; i=1(1)p \qquad (4.4.1.7)$$

substituting for A_c in (4.4.2) allows x to be computed using the coupled systems,

$$\left.\begin{aligned} &i=1, \; Q_1 y_1 = f \\ &1<i\leq p, \; Q_i y_i = y_{i-1} \; , \\ &p<i\leq 2p, \; Q^T_{2p-i+1} y_i = y_{i-1} \; , \end{aligned}\right\} \qquad (4.4.1.8)$$

where y_i, i=1(1)2p are auxiliary vectors and $x=y_{2p}$. By a simple extension of the solution method in (4.4.7)-(4.4.9), (4.4.1.8) reduces to 2p matrix-vector multiplications. It follows that (4.4.2) is solved by interpreting A_c as 2p special circulant problems of semi-bandwidth $\bar{p}+1$ (with $\bar{p}=1$). Hence,

Theorem 4.4.1.3:

The solution of $A_c x=f$ where A_c is an n×n symmetric circulant matrix of semi-bandwidth r requires, $T=2(2r+3)n+8r+2$ using the Toeplitz solver, and the factorisation (4.4.1.6).

Proof:

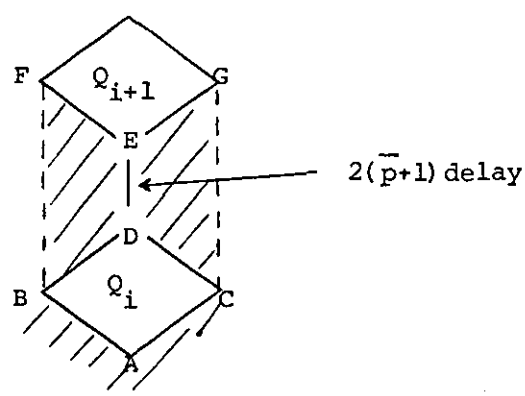
Using the factorisation above we have 2r problems with semi-bandwidth $\bar{p}=1$. So the inverter latency is $n+2(\bar{p}+1)=n+4$, and a single pass through the rank-1 pipe costs 4n+2 cycles. For 2r problems, the input length is $2r(2n)+(2r-1)2(\bar{p}+1)=4rn+4(2r-1)$. All these values pass through the matrix vector array generating the same number of outputs which filter out of the array with an additional n cycles delay. Summing these delays gives $T=2(2r+3)n+8r+2$. This answer is verified

by applying Theorem (4.4.1.2) with k=r and p=$\bar{p}$+1. Note that k=r not 2r because the proof of Theorem (4.4.1.2) assumes each system computes L^{-1} and L^{-T} but the solutions in (4.4.1.8) are a special case using only L^{-1}. Consequently 2r problems can be compressed into the space and time of r problems by using the spare L^{-T} places. Now solving k problems of semi-bandwidth r+1 using the old factorisation is equivalent to solving rk problems of semi-bandwidth 2 with the new factorisation, and Theorem (4.4.1.2) yields the speed-up inequality

$$(6-4k)n+8kr(n+1)+2 > (6-4kr)n+8kr(n+1)+2 \ . \qquad (4.4.1.9)$$

It follows that for r>1 the new factorisation is faster. Furthermore the method can be applied to different problems of varying bandwidth. For example, the time to solve $\bar{k}$ problems of semi-bandwidth r_i+1, i=1(1)$\bar{k}$ is given by Theorem (4.4.1.2) with $k = \sum_{i=1}^{\bar{k}} r_i$ and p=1.

Throughout the discussions so far we have assumed that successive matrix inputs arriving at the matrix vector array in Fig.(4.4.1.1) synchronise. For a p>1 column rank annihilation strategy this is trivial to arrange using (4.4.9) because there is plenty of time for y in (4.4.9a) to filter out of the array and then be pumped back to synchronise with $\tilde{L}^{-T}$ in (4.4.9b). For (4.4.1.8) the arrival of successive y_i, i=1(1)2p is a time critical problem. The typical structure of two solutions Q_i and Q_{i+1} is given by,

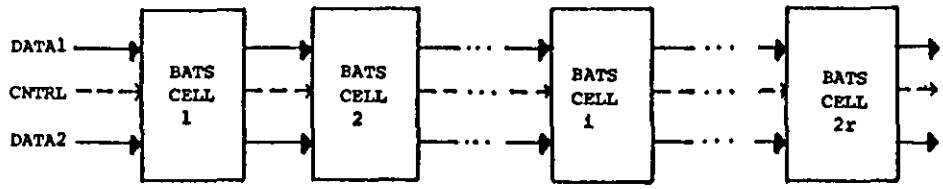


where the shaded regions are zero elements used solely for synchronisation and AB=AC=BD=EF=n as the matrix vector array has 2n-1 cells. In general, the computation of $y_i=Q_i^{-1}y_{i-1}$ starts when the element at A enters the array. After a further n cycles elements along BC have entered implying that the last component of y_{i-1} has been input and the first component of y_i has been output. Now the elements of BDEF have the property of never modifying results (i.e., neutral computation). Consequently the synchronisation of known elements of y_i with Q_{i+1} can be overlapped with computation of unknown y_i values, by a simple feedback loop with a delay $2(\bar{p}+1)+1=5$ cycles as $\bar{p}=1$. Note that the loop must be switchable to allow the input of the initial vector (y_1) of a new problem. A similar argument verifies that successive rank annihilations can start in the same way, and we conclude that no additional delays are required over those incorporated into the theorems.

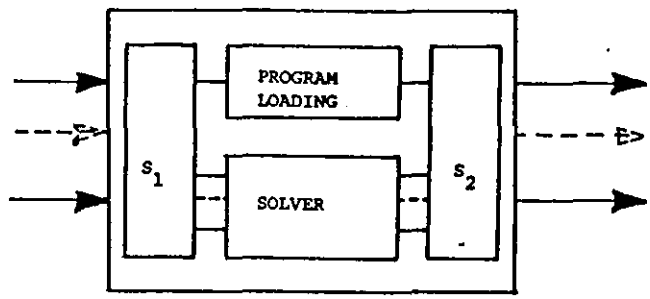
4.4.2 A Linear Array Scheme

An alternative method for the solution of (4.4.1.8) is a linear array of the form shown in Fig.(4.4.2.1) which makes use of the symmetry in A_c and the simple structure (4.4.1.7). The array itself consists of 2r BATS cells and computes in a time $T=\ell+2rc$ where ℓ=total input length of data and c=cell latency. Each BAT cell is made up of a computation part solving one $Q_iy_i=y_{i-1}$ problem and a parameter part which is responsible for loading and saving the correct α_i associated with each cell. Finally, two trivial switching networks are used to route data between cells and cell components.

The computation part of the BATS pipeline solves the generic problem of the form,



a) BATS linear array



S1,S2 simple switching networks

b) BATS cell structure

FIGURE 4.4.2.1: BATS pipeline

$$By = v \ , \tag{4.4.2.1}$$

with y unknown and v known n×1 vectors respectively, while B has the form,

$$B = \begin{bmatrix} 1 & & & \alpha \\ \alpha & 1 & & \\ & \ddots & \ddots & \\ & & \alpha & 1 \end{bmatrix}_{n\times n} \tag{4.4.2.2}$$

the context of the problem determined by the substitution $\alpha = \alpha_i$ depending on cell, thus it is necessary only to discuss the solution of (4.4.2.1). From Pickering [84] we can write,

$$B = (\bar{B}+P) = \bar{B}(I+\bar{B}^{-1}P) \ . \tag{4.4.2.3}$$

where $\bar{B}$ is lower triangular and,

$$P = \begin{bmatrix} 0 & & \alpha \\ & 0 & \\ 0 & & 0 \end{bmatrix}, \quad I+\bar{B}^{-1}P = \begin{bmatrix} 1 & & \alpha \\ & 0 & -\alpha^2 \\ & & \alpha^3 \\ & & \vdots \\ & & 1+(-\alpha)^n \end{bmatrix}$$

and solve the coupled system,

$$\text{a) } \bar{B}u = v \qquad \text{b) } (I+\bar{B}^{-1}P)y = u\,, \tag{4.4.2.4}$$

instead of (4.4.2.1).

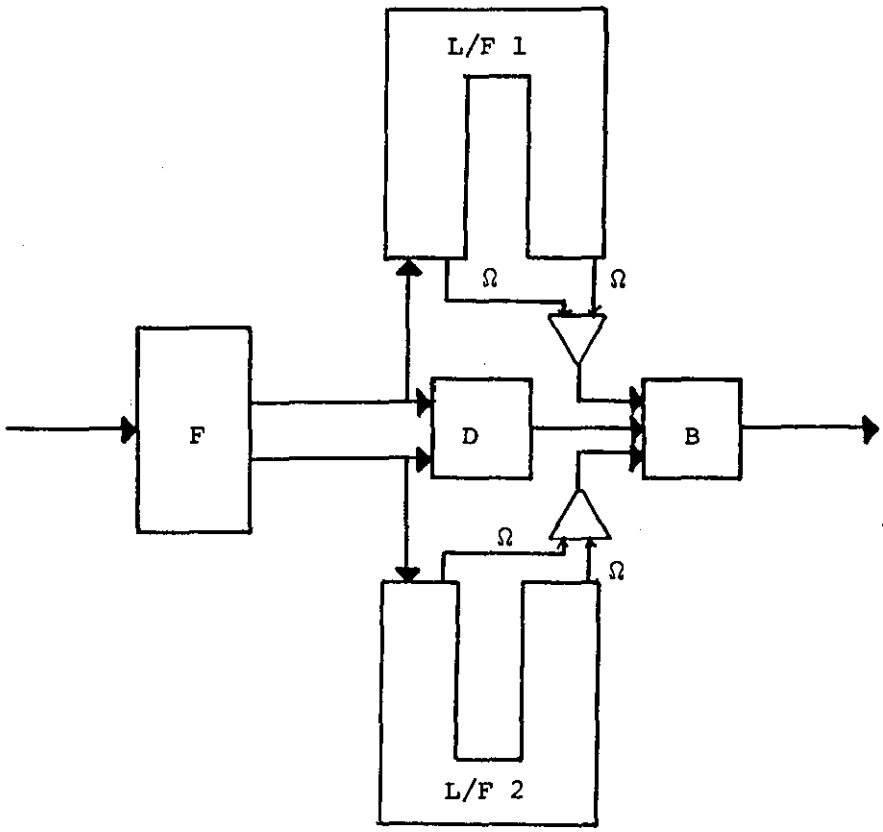
Now using only the value α the solution of (4.4.2.4a) and construction of $(I+\bar{B}^{-1}P)$ can be computed in parallel by the structure in Fig.(4.4.2.2). The design consists of three cells and two memory sets of n bi-directional delay registers each. The cells are controlled by two tag bits associated with the input v, and memory by a single control bit c_1 which remains constant during computation. The tag bits mark the start and finish of data and can be used to generate cell controls during computation as summarized below.

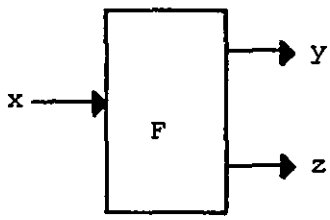
t_1	t_2	Action
0	0	Normal computation
0	1	Reset cells/disable c_1
1	0	Enable c_1 and load y_n
1	1	-

where data is input in the order

$$\begin{array}{lccccccc} \text{DATA} & v_n & \cdots & v_3 & v_2 & v_1 & & \\ t_1 & 1 & \cdots & 0 & 0 & 0 & \longrightarrow & \\ t_2 & 0 & \cdots & 0 & 0 & 1 & . & \end{array} \tag{4.4.2.5}$$

Now the BATS cell computesas follows:- The L/F stores act as FIFO queues



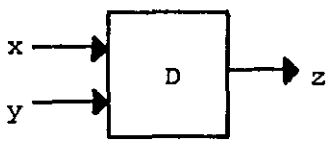


(i) $y=x-\alpha g_i$ $i=1(1)n$

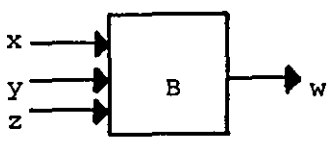
$g_i=y$

(ii) $a=o-\alpha a$

Initially $g_o=0$, $a=-1$



$z = x/(1-y)$



IF t_1 THEN ysave=y

IF $\bar{t}_1$ THEN w=ysave

ELSE w=x-(z*ysave)

where t_1 is tag bit control before entering L/F, $\bar{t}_1$ controls generated after leaving L/F store.

FIGURE 4.4.2.2: O(n) BATS cell structure

initially. The F-cell computes (4.4.2.4a) and generates $(I+\bar{B}^{-1}P)$ by the sequence $-\alpha,\alpha^2,\ldots,(-\alpha)^n$ piping the results into the L/F memory, while the D-cell continually performs subtract/divides to find y_n. The D-cell result is valid only after n+1 cycles at which time u_n and its associated tag bits from v_n have been stored. Now the method of solution for (4.4.2.4b) depends on the BATS cell position in the array, which in turn depends on the computation in (4.4.1.8) to be performed next. If we are computing $Q_p y_p = y_{p-1}$ the next calculation is $Q_p^T y_{p+1} = y_p$ and output must be in a reversed order to input, otherwise it is in the same order. It follows that when the tag enables c_1, if $c_1=1$ then L/F stores act as LIFO and if $c_1=0$ they act as FIFO. Consequently we use tag t_2 to reset the F-cell and B-cell before computation and tag t_1 to load y_n from the D-cell into B-cell. Thus, the control c_1 is sufficient to switch the direction of vector output computing (4.4.2.4b) by forward or backward recursion as necessary. The B-cell can then use the tag bits to decide whether to send y_n as first or last output, $t_1 \wedge \bar{t}_2$ implying first and $t_2 \wedge \bar{t}_1$ last. Finally when the vector is reversed the role of the tag bits must also be reversed to prevent incorrect control signals, this is achieved by the formula,

$$\left.\begin{aligned} t_1 &= (t_1 \wedge \bar{c}_1) \vee (t_2 \wedge c_1) \\ t_2 &= (t_2 \wedge \bar{c}_1) \vee (t_1 \wedge c_1) \end{aligned}\right\} \qquad (4.4.2.6)$$

and is implemented by trivial combinational logic. Fig.(4.4.2.3) indicates the necessary c_1 assignments and data reversals. The latency of the BATS cell is clearly c=n+2 allowing n cycles to fill the L/F stores and two cycles for the delay through the F- and B-cells. In terms of area the BATS cell requires 2n L/F registers and a total of four ips cell equivalents for F-, D- and B-cells with extra logic for

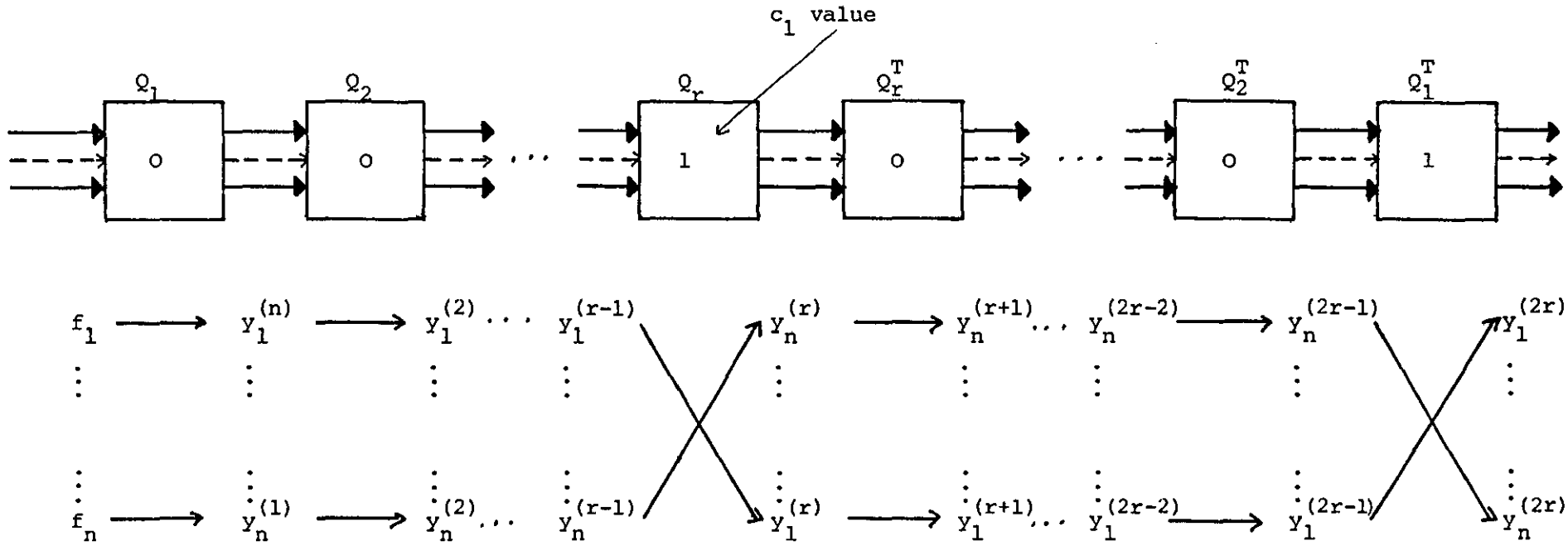


FIGURE 4.4.2.3: Data reversal and L/F control allocation in BATS pipe

loading parameters and reversing tags assumed negligable. Consequently the design is termed an O(n) BATS cell. The latency of the BATS pipe in Fig.(4.4.2.1) using this O(n) cell is 2r(n+2) giving a computation time $T=\ell+2r(n+2)$, where ℓ is determined by the size of the problem n and the cost of loading the α_i, i=1(1)r parameters. Adopting a simple addressing scheme modified from Umeo [85] and using the fact that computation uses only a single data input line α_i loading is achieved as follows. The right handside f in (4.4.1.8) and the α_i parameters are input to the left boundary of the linear array in the form,

$$\text{DATA} \equiv f_n \ldots f_2\ f_1\ \alpha_1\ \alpha_2\ \cdots\ \alpha_r\ , \tag{4.4.2.7}$$

using a single connection, while a set of addresses associated with each α_i is input on the remaining connection with the form,

$$\text{ADDR} \equiv \underbrace{0\ \ \ldots\ \ 0}_{n}\ 1,2,\ \ldots\ r\ . \tag{4.4.2.8}$$

The parameter section of each BATS cell contains an address describing its position in the array and a comparator. The parameter section checks each input address with its stored address, loading the α_i when they match. Synchronisation for a number of problems is then achieved by piping the DATA and ADDR through the L/F stores and utilising the unused tag bit combination ($t_1=t_2=1$) to disable the cell computation as α_i and the addresses pass through. The f_i and zero addr values are replaced by true results of u_i and $(-\alpha)^i$ respectively as explained above. We conclude that k problems have input length $\ell=k(n+r)+(k-1)n$ provided each address and hence its associated parameter is allocated to two different cells; for example by numbering cells from left to right for i=1(1)2r and allocating cell i with address i for $i\leq r$ and with address 2r-i+1 for i>r.

Theorem 4.4.2.1:

The solution of k systems of the form $A_c x=f$ where A_c is an n×n symmetric circulant system of semi-bandwidth r+1 can be computed on a linear array of 2r O(n) BATS cells in $T=(2k+2r-1)n+r(4+k)$.

Proof: [summation of timings in the above discussion].

Where the extra (k-1)n delays are required to allow for the emptying of a cell LIFO, a simple comparison using Theorems (4.4.1.2) and (4.4.2.1) gives the relation,

$$(6-4kr)n + 8kr(n+1) + 2 > (2k+2r-1)n + r(4+k) \ . \qquad (4.4.2.9)$$

indicating that the O(n) BATS cell and array is faster than both previous methods using the relation (4.4.1.9).

4.4.3 P-Cyclic and Double Pipe Schemes

There are many variations on the O(n) BATS cell which utilise further features of the special factorisation of Evans & Audish [86]. In particular the condition $0<|\alpha_i|<1$, i=1(1)2r is often satisfied in practice. Consequently if a bound $\alpha=\max|\alpha_i|$ can be found for a special set of problems the BATS cell can be modified to improve speed and throughput using the P-cyclic properties of (4.4.2.2).

Now consider (4.4.2.1) where B is a block p-cyclic matrix with p=n and 1×1 block sizes. The block Jacobi iteration matrix is $B_J=L+U$ with form,

$$B_J = \begin{bmatrix} 0 & & & \alpha \\ \alpha & & & \\ & \ddots & & \\ & & & \alpha \\ & & \alpha & 0 \end{bmatrix} \ , \ 0<\alpha<1 \ , \qquad (4.4.3.1)$$

and L and U strictly lower and upper triangular matrices. We can choose a matrix,

$$B_J(\beta) = \begin{bmatrix} O & & & \bar{\beta}^{(n-1)}\alpha \\ \beta\alpha & & O & \\ & \ddots & & \\ O & & \beta\alpha & O \end{bmatrix} \qquad (4.4.3.2)$$

and $$B_J(\beta) = \beta L + \bar{\beta}^{(n-1)} U \qquad (4.4.3.3)$$

such that,

$$B_J^n(\beta) = B_J^{(n)} \quad , \quad \text{for all } \beta \neq 0 \ , \qquad (4.4.3.4)$$

Implying that $B_J(\beta)$ has eigenvalues independent of β making B p-cyclic and consistently ordered. Now,

Theorem 4.4.3.1

Let an $n \times n$ matrix A be a consistently ordered p-cyclic matrix with non-singular diagonal submatrices. If $\omega \neq 0$ and λ is a zero eigenvalue of $L_\omega = (I-\omega L)^{-1}\{\omega U + (1-\omega)L\}$ and if μ satisfies,

$$(\lambda+\omega-1)^p = \lambda^{p-1}\omega^p\mu^p \ , \qquad (4.4.3.5)$$

then μ is an eigenvalue of B_J, and conversely, if μ is an eigenvalue of B_J and λ satisfies (4.4.3.5), λ is an eigenvalue of L_ω.

Proof: [see Varga [62] pp.106-107].

With $\omega=1$, L_ω is the iteration matrix in the Gauss-Seidel algorithm and a simple relationship between the eigenvalues of B_J and L_1 when p=n is given by

$$\lambda^n = \lambda^{(n-1)}\mu^n \quad \text{or} \quad \lambda = \mu^n \ , \qquad (4.4.3.6)$$

thus from (4.4.3.2),

$$||B||_\infty = |\alpha| \leftrightarrow \rho(B) \leq |\alpha| < 1$$

and $$||B||_2 = \sqrt{|\alpha|^2} = |\alpha|, \ \rho(B) \leq |\alpha| < 1,$$

and using (4.4.3.6) $\rho(L_1) < \rho(B)$ implying that each coupled system in (4.4.1.8) can be solved by iteration. Considering the relative error

of successive approximations to (4.4.2.1) gives,

$$||y^{(k)}-y|| \approx \rho(L_1)^k||y^{(0)}-y||$$

for k iterations, and using (2.4.3.12) with $\rho(L_1)=\rho(\lambda)<|\alpha|$

$$k \geq \frac{t}{-\log_{10}\rho(L_1)} = \frac{t}{-n\log_{10}|\alpha|} \quad . \tag{4.4.3.7}$$

Thus $k\geq 1$ for $n>>t$ implying that 1 or 2 iterations of the Gauss-Seidel method are the minimum number of iterations.

The Gauss-Seidel iteration can be written in the form,

$$\left.\begin{aligned} &a)\quad y_1^{(k)} = v_1-\alpha y_n^{(k-1)} \\ &b)\quad y_i^{(k)} = v_i-\alpha y_{i-1}^{(k)} \quad , \; i=2(1)n \end{aligned}\right\} \tag{4.4.3.8}$$

and by repeated substitution in (4.4.3.8b)

$$y_n^{(k)} = v_n-\alpha v_{n-1}+\alpha^2 v_{n-2},\ldots,(-1)^{n-i}\alpha^{n-i}v_{n-i}+(-\alpha)^n y_n^{(k-1)} \tag{4.4.3.9}$$

now if y_n is the exact solution of the nth unknown and r_n is the error term,

$$y_n^{(k-1)} = y_n+r_n$$

and (4.4.3.9) is,

$$y_n^{(k)} = v_n-\alpha v_{n-1} \ldots(-1)^{n-i}\alpha^{n-i}v_{n-i},\ldots,(-\alpha)^n(y_n+r_n) \tag{4.4.3.10}$$

but as $0<|\alpha|<1$ and sufficiently large $n\geq s$, $|\alpha^s|=0$ truncating (4.4.3.10).

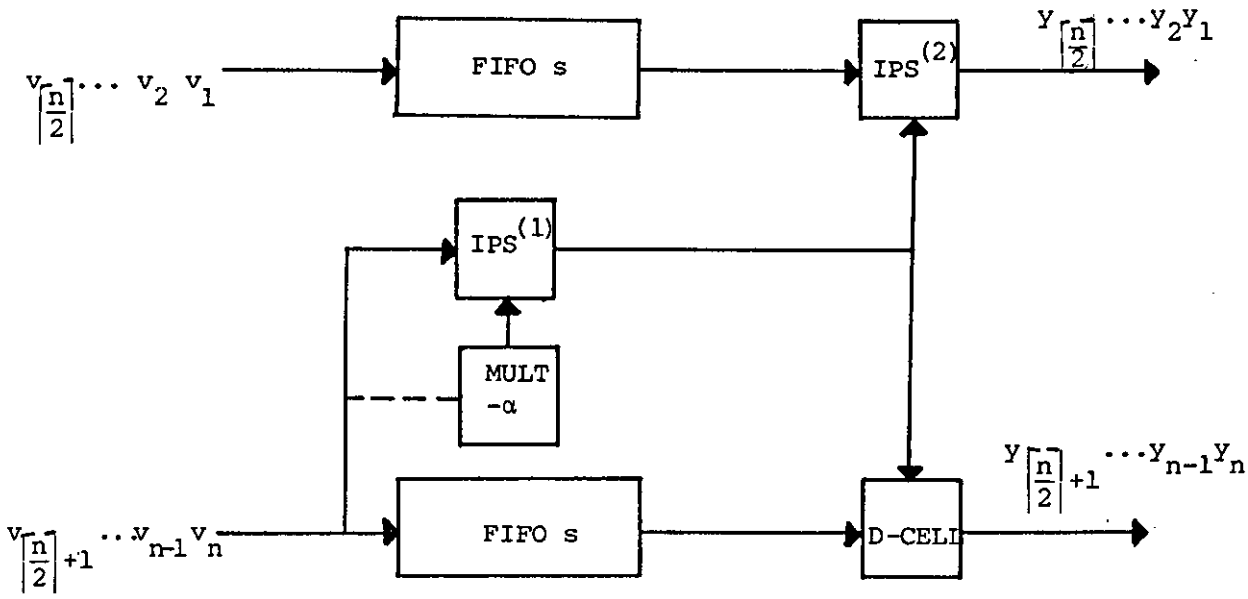
Hence $y_n^{(k)}=v_n-\alpha v_{n-1}+\alpha^2 v_{n-2},\ldots,(-1)^i\alpha^i v_i,\ldots,(-\alpha)^{s-1}v_{n-s+1}$ (4.4.3.11)

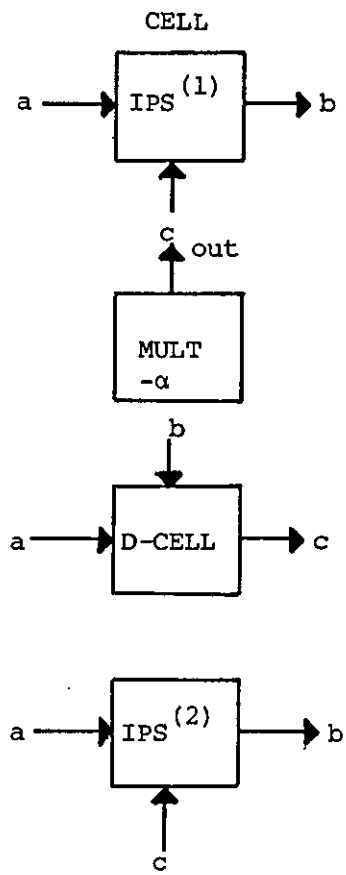
It follows that $y_n^{(1)}=y_n$ after only a single iteration, and yields a direct method of solution involving two steps.

(i) compute (4.4.3.11)

(ii) construct the forward recurrence (4.4.3.8)

A generic cell capable of computing both the Q and Q^T forms in (4.4.1.8) is given in Fig.(4.4.3.1). Notice that data is split into two input streams $v_1,v_2,\ldots,v_{\lceil n/2\rceil}$ and $v_n,v_{n-1},\ldots,v_{\lceil n/2\rceil+1}$ with the implicit





COMMENT

(accumulate y_n)
b=a-b*c
t=b
(initially t=0)

α is preloaded, and a=1
initially
a=0-αa
out=a

IF t_1 THEN {bsave=b,c=bsave}
ELSE c=(a-bsave)/α

IF t_1 THEN {csave=c}
b=a-b*csave
t=b

FIGURE 4.4.3.1: P-cyclic BATS cell

assumption that $s \leq \lceil n/2 \rceil$ which is reasonable for large n. Operation of the cell is simple. The FIFOs act as delay queues for both data and control tags while y_n is computed. On emerging from the queues IPS(2) computes half the results using the forward recurrence, and as y_n is available the D-cell computes the remaining terms by a backward recursive procedure. Calculation of y_n is performed by two cells, the multipliers form the powers α^i reducing (4.4.3.11) to a simple dot product accumulated by IPS(1). As only s terms must be accumulated we need to delay the substitution phase by s cycles giving the size of the FIFO queues. By pipelining control tags associated with data through the FIFOs control can be divided into pre and post-FIFO processing, and controls the IPS(1), Mult, IPS(2) and D-cell as summarized below.

Pre-FIFO control

t_1	t_2	Action
0	0	Normal computation (next polynomial term)
0	1	Initialise Mult and IPS(1)
1	0	Load first polynomial term in IPS(1)
1	1	Null

Post-FIFO control

t_1	t_2	Action
0	0	Calculate y_i, y_{n-i+1} in IPS(2) and D-cell
0	1	Copy α to IPS(2) and D-cell
1	0	Load y_n into IPS(2) and D-cell (set y_n for output)
1	1	-

as in the O(n) BATS cell $t_1=t_2=1$ can be used to disable cells while

parameter data pass through the cell. The two level pipelining of control and data in FIFOs increases throughput, as total input length $\ell=k(\lceil n/2\rceil+r)$ for k problems. While the latency of the P-cyclic cell is c=(s+1) hence,

Theorem 4.4.3.2:

The solution of k circulant symmetric systems of form $A_c f=x$ where $A_c=Q_1\cdots Q_r Q_r^T\cdots Q_1^T$ is an $n\times n$ matrix with semi-bandwidth r+1 is solved in $T=k(\bar{n}+r)+2r(s+1)$ using 2r p-cyclic cells, where,

$$\bar{n} = \begin{cases} s \text{ for} & s>\lceil n/2\rceil \\ \lceil n/2\rceil & \text{otherwise} \end{cases}$$

and $\alpha^s=0$ when $\alpha=\max_{1\le i\le 2r}(|\alpha_i|)$ is selected from the Q_i in (4.4.1.7).

Proof:

Generally data is split into two streams $v_1,v_2,\ldots,v_{\bar{n}}$ and $v_n,\ldots,v_{n-\bar{n}+1}$. When $\bar{n}>\lceil n/2\rceil$ some data is repeated for input but retains smooth data flow. For $\bar{n}\le\lceil n/2\rceil$ argument is the same as the above discussion.

In hardware terms each cell requires 2s registers in the FIFO queues and the equivalent of four inner products for the computation, giving a total of 4rs registers and 8r ips equivalents for the full array.

The basic idea of the p-cyclic cell can be extended to produce a double pipe scheme for solving circulant systems, and more generally, a recursive decoupling scheme producing a systolic tree arrangement (similar to Section 4.1). The double pipe is derived by considering the system,

$$Tx = z \ , \qquad (4.4.3.12)$$

of the form,

$$\begin{bmatrix} 1 & \alpha & & & & \\ & 1 & & & & \\ & & \alpha & & & \\ & & & 1 & & \\ & & & & \alpha & \\ \alpha & & & & & 1 \end{bmatrix}_{n\times n} \begin{bmatrix} x_1 \\ \vdots \\ x_{n/2} \\ x_{\frac{n}{2}+1} \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} z_1 \\ \vdots \\ z_{n/2} \\ z_{\frac{n}{2}+1} \\ \vdots \\ z_n \end{bmatrix} \qquad (4.4.3.13)$$

where $0<\alpha<1$ as before. We partition the system so,

$$\left[\begin{array}{c|c} P & B \\ \hline B & P \end{array}\right] \begin{bmatrix} x_1 \\ \hline x_2 \end{bmatrix} = \begin{bmatrix} \tilde{z}_1 \\ \hline \tilde{z}_2 \end{bmatrix} \qquad (4.4.3.14)$$

which after some simple eliminations yields,

$$\left.\begin{array}{ll} \text{a)} & (P-BP^{-1}B)x_1 = \tilde{z}_1 - BP^{-1}\tilde{z}_2 \\ \text{b)} & (P-BP^{-1}B)x_2 = \tilde{z}_2 - BP^{-1}\tilde{z}_1 \end{array}\right\} \qquad (4.4.3.15)$$

denoting

$$\left.\begin{array}{ll} \text{a)} & \bar{P} = (P-BP^{-1}B) \\ \text{b)} & \bar{z}_1 = \tilde{z}_1 - BP^{-1}\tilde{z}_2 \\ \text{c)} & \bar{z}_2 = \tilde{z}_2 - BP^{-1}\tilde{z}_1 \end{array}\right\} \qquad (4.4.3.16)$$

representing the decoupled system,

$$\left[\begin{array}{c|c} \bar{P} & 0 \\ \hline 0 & \bar{P} \end{array}\right] \begin{bmatrix} x_1 \\ \hline x_2 \end{bmatrix} = \begin{bmatrix} \bar{z}_1 \\ \hline \bar{z}_2 \end{bmatrix} \qquad (4.4.3.17)$$

where,

$$\bar{P} = \begin{bmatrix} 1 & \alpha & & & & \\ & 1 & & & & \\ & & \ddots & & & \\ & & & \frac{n}{2}+1 & & \\ & & & & \ddots & \\ (-\alpha) & & & & & 1 \end{bmatrix} \qquad (4.4.3.18)$$

now denoting,

$$\left.\begin{aligned} &\text{a)} \quad P_1(n/2) = \alpha(z_{\frac{n}{2}+1} - \alpha z_{\frac{n}{2}+2}, \ldots, +(-\alpha)^{\frac{n}{2}-1} z_n) \\ &\text{b)} \quad P_2(n/2) = \alpha(z_1 - \alpha z_2, \ldots, +(-\alpha)^{\frac{n}{2}-1} z_{\frac{n}{2}}) \end{aligned}\right\} \qquad (4.4.3.19)$$

We conclude that there is no need to compute P^{-1} explicitly and that (4.4.3.17) has the strictly upper triangular form,

$$\begin{bmatrix} 1 & \alpha & & & & & \\ & 1 & & & & & \\ & & \alpha & & & & \\ & & & 1 & & & \\ & & & & 1 & \alpha & \\ & & & & & & \alpha \\ & & & & & & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_{n/2} \\ x_{\frac{n}{2}+1} \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} z_1 \\ \vdots \\ z_{n/2} - P_1(n/2) \\ z_{\frac{n}{2}+1} \\ \vdots \\ z_n - P_2(n/2) \end{bmatrix} \qquad (4.4.3.20)$$

when the vanishing point $\alpha^s = 0$ satisfies $s \leq n/2$. Now using the principle of the p-cyclic cell to solve a system of the form,

$$\begin{bmatrix} 1 & \alpha & & & \\ & 1 & & & \\ & & \alpha & & \\ & & & 1 & \\ & & & & \alpha \\ & & & & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_{n/2} \end{bmatrix} = \begin{bmatrix} z_1 \\ \vdots \\ z_{n/2} - P_1(n/2) \end{bmatrix} \qquad (4.4.3.21)$$

repeated forward substitution yields,

$$x_1 = z_1 - \alpha z_2 - \alpha^2 z_3, \ldots, (-\alpha)^{s-1} z_s \ . \qquad (4.4.3.22)$$

Hence,

$$P_2(n/2) = \alpha x_1 \ , \text{ and} \qquad (4.4.3.23)$$

consequently, (4.4.3.21) is solved by a backward recursive scheme

producing $x_{n/2},\ldots,x_{\frac{n}{4}+1}$ and a forward recursive procedure producing $x_1,\ldots,x_{\frac{n}{4}}$. With analogous reasoning

$$x_{\frac{n}{2}+1} = z_{\frac{n}{2}+1} - \alpha z_{\frac{n}{2}+2} + \alpha^2 z_{\frac{n}{2}+3},\ldots,+(-\alpha)^{s-1} z_{\frac{n}{2}+s-1} \qquad (4.4.3.24)$$

and

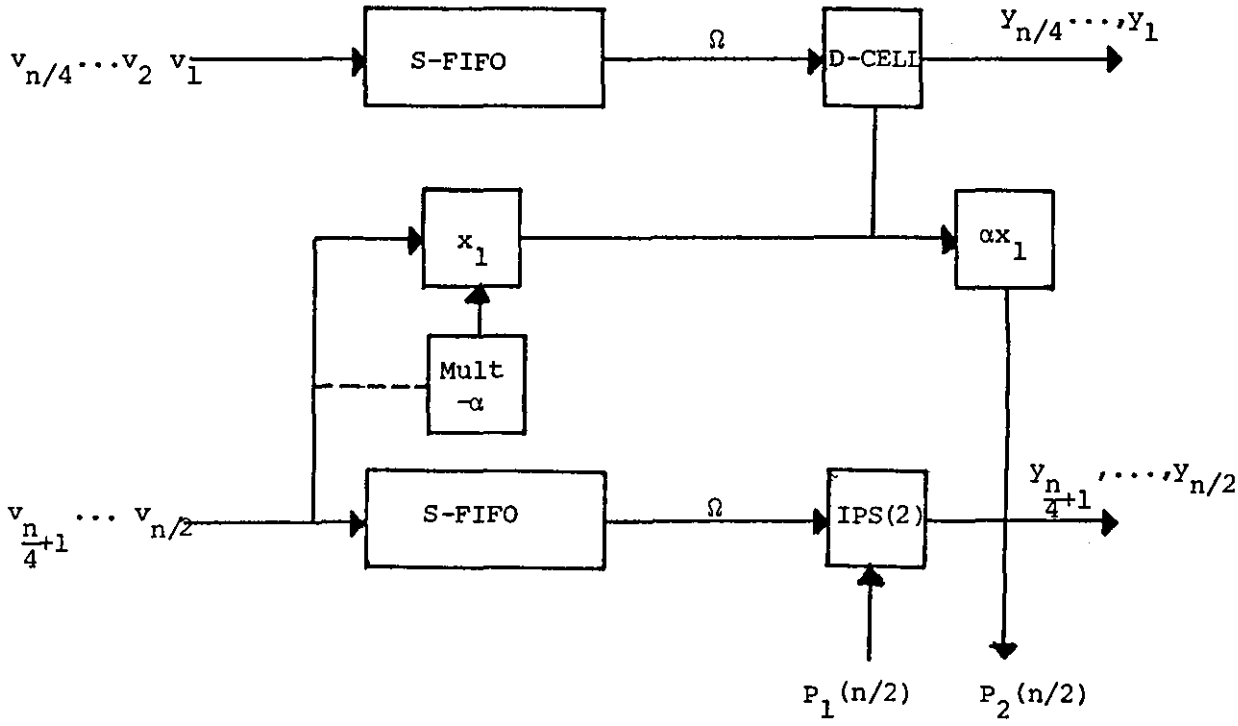
$$P_1(n/2) = \alpha x_{\frac{n}{2}+1} \quad . \qquad (4.4.3.25)$$

giving $x_n,\ldots,x_{\frac{3n}{4}+1}$, and $x_{\frac{n}{2}+1},\ldots,x_{\frac{3n}{4}}$. Thus using two modified BATS cells as shown in Fig.(4.4.3.2) a double pipe BATS solver is produced. We use one cell for each decoupled system, and modify the cells to produce P_1 or P_2 (depending on input data) and to pass the results between each other. The first action of the cell is to compute the modification to the righthand side of (4.4.3.13), and it is clear that x_1 and $x_{\frac{n}{2}+1}$ are available one cycle before P_1 and P_2. Consequently an additional delay is added to synchronise the loading of x_1 and $x_{\frac{n}{2}+1}$ into the D-cells with the loading of P_1 and P_2 into the IPS(2) parts of the respective BATS cells. The timing of the double pipe BATS array is derived directly from Theorem (4.4.3.2) by substituting s+1 for s and putting,

$$\bar{n} = \begin{cases} s, & \lceil n/4 \rceil < s \leq \lceil n/2 \rceil \\ \lceil n/4 \rceil, & s \leq \lceil n/4 \rceil \end{cases} \qquad (4.4.3.26)$$

Clearly data is not repeated when $\bar{n}=\lceil n/4 \rceil$, and gives the best results. In terms of IPS equivalents the double pipe requires twice as many cells, doubling the hardware requirements to 16r ips cells. 4s FIFO registers are also required.

a) Modified cell



b) Inter-cell connections

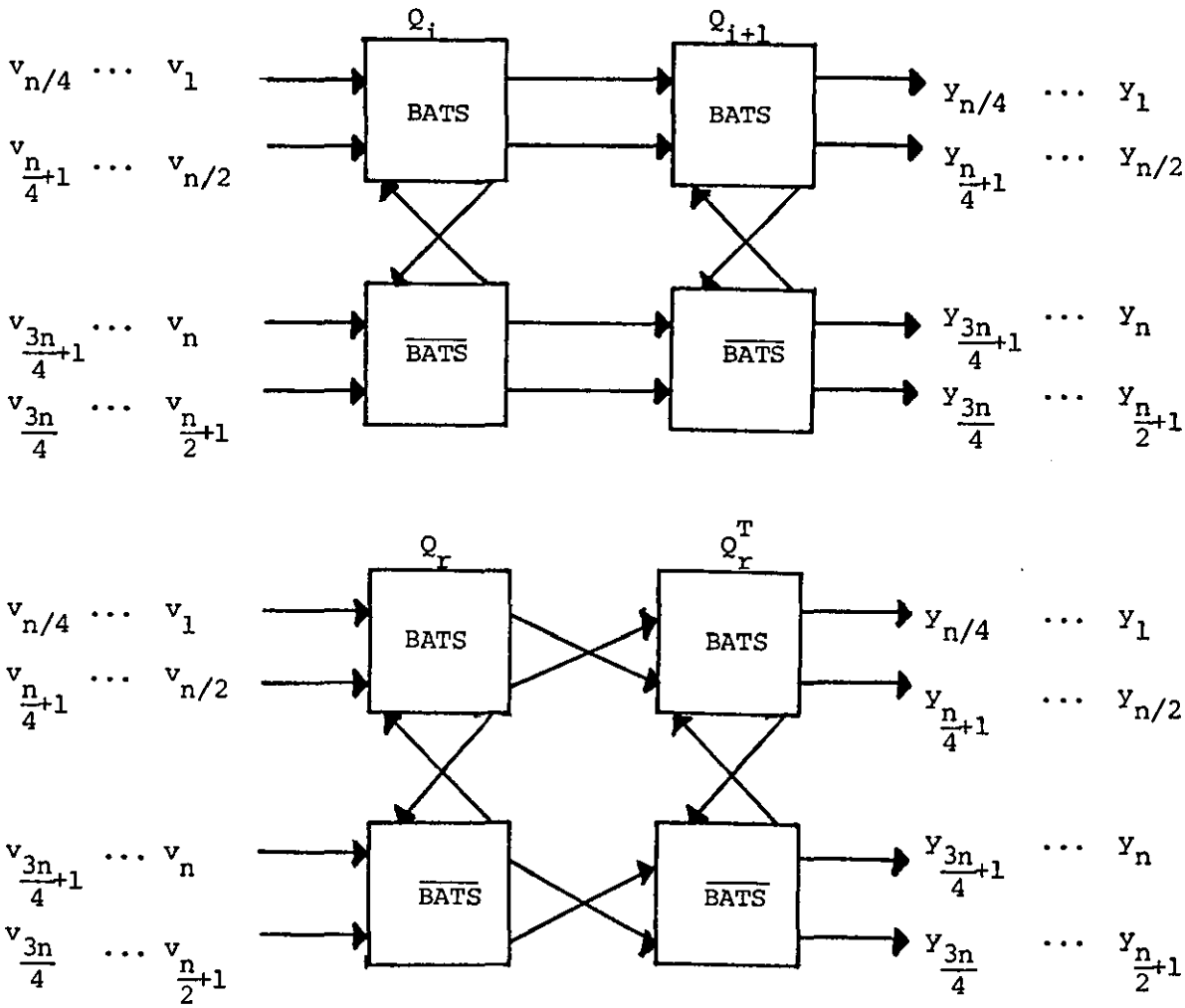


FIGURE 4.4.3.2: Modified P-cyclic cell

4.4.4 Comparison of Methods

A summary of timings and hardware requirements for solving (4.4.2) using the arrays discussed above are listed in Table (4.4.4.1). From this table it is clear that the Toeplitz pipe using rank annihilation is slowest and uses most hardware. While the O(n) BATS cell pipeline and its modification using the P-cyclic properties of A_c make them comparable to the Brent and Luk (BL) and S.Y. Kung and Hu (KH) schemes. For instance, when k>2r-1 the O(n) BATS cell has a time bounded above by that of the BL method and from below by the KH scheme. In terms of IPS cell equivalents the O(n) BATS cell is far superior to both BL and KH, and for register usage bounded below by BL and above by KH. Consequently the O(n) BATS scheme gives intermediate performance between the two methods.

The results for the Toeplitz pipe are disappointing but easily understood. First consider Chen's factorisation yielding the coupled systems (4.4.5), which are more difficult to solve than straightforward lower and upper triangular factors. The motivation behind the method is to quickly invert the special forms using rank-annihilation converting the solution to simple matrix-vector problems which are easily pipelined. This breaks the sequential dependency of solving by forward and backward substitution which is normally associated with factorisation methods. Hence throughput should increase hopefully bringing overall computation time down. However the latency of the RANK-1 scheme is more than the computation time for both the BL and KH methods making a speedup impossible. Tracing the computation further shows that the computation of (4.3.5e) (an inherently sequential task) is the root cause of the problem. Notice that throughput is increased over the BL scheme as a new

problem can be input every 2(n+p+1) cycles in the Toeplitz pipe rather than 4n cycles in BL, and is also comparable to the KH method for small p. Consequently, the delay and extra hardware associated with rank annihilation creates overheads which destroy any advantages from pipelining. The new factorisation of Audish & Evans [85b] simplifies the data flow through the Toeplitz pipe and improves the cell efficiency, and timing. The improvement is due to the factorisation structure which demands that only a single RANK-1 elimination occurs on each component coupled system, before its solution. A_c itself and hence its factors can be encoded by a small number of parameters. The Toeplitz pipe expands this compact form into a full matrix format increasing hardware and computation time due to the increases in component pipe latencies. Again, this traces back to the rank annihilator which makes no assumptions about matrix structure. The O(n) BATS cell and linear array is a design specifically aimed at the Audish & Evans [85b] factorisation. The compact representation of the factors is retained by using a parameter loading scheme, and a modified method for solving the quasi-tridiagonal form in Pickering [84]. The minimum amount of additional data required is produced reducing design area and simplifying systolic data flow, while retaining the throughput of the Toeplitz pipe and KH schemes. As the solution of each component system in (4.4.1.8) uses a single BATS cell the pipeline size is related to the semi-bandwidth r+1 rather than n in the other schemes, a vital attribute of systolic arrays. The only drawback being that each cell contains O(n) registers.

The p-cyclic cell produces a truly problem size independent array but relies on special features of the factorisation process (discussed below), which may not always be satisfied. In particular we assume that

METHOD	TIME	CELLS	REGISTERS
Brent & Luk	4nk	3n	2n
S.Y. Kung & Hu (PLP)	2nk	3n	$O(n^2)$
Toeplitz pipe (Chen's method)	(6-4k)n+ 8kr(n+1)	10n+r-3	$2(2n-1)^2$
Audish & Evans method	(6+4kr)n+ 8kr+2	"	"
O(n) BATS pipe	(2k+2r-1)n r(4+k)	8r	4rn
P-cyclic cell	$k(\lceil n/2 \rceil + r) +$ $2r(s+1)$	8r	4rs

k=number of consecutive problems
r+1=semi-bandwidth
n=problem size
s=vanishing point $\alpha^s=0$ for $0<|\alpha|<1$

Times and cell counts multiples of IPS equivalents

TABLE 4.4.4.1: Timing and hardware for Toeplitz solvers

t \ α	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
2	2	3	4	6	7	10	13	21	44
4	4	6	8	11	14	19	26	42	88
5	5	8	10	13	17	23	33	52	110
6	6	9	12	16	20	28	39	62	132
7	7	11	14	18	24	32	46	73	153
8	8	12	16	21	27	37	52	83	175
9	9	13	18	23	30	41	59	93	197
10	10	14	20	26	34	46	65	104	219
11	11	16	22	28	37	50	72	114	241
12	12	18	23	31	40	55	78	124	263
13	13	19	25	33	44	59	84	135	285
14	14	21	27	36	47	64	91	145	306
15	15	22	29	38	50	68	97	155	328
16	16	23	31	41	54	73	104	166	350

TABLE 4.4.4.2: Tabulation of vanishing point s for values of α and decimal places t

$\alpha^s<\varepsilon$ where $\varepsilon=1\times10^{-t}$ and $s = \dfrac{\log_{10}\varepsilon}{\log_{10}\alpha}$.

the α_i, i=1(1)r parameters satisfied $0<|\alpha_i|<1$. To derive a fixed sized cell we then chose $\alpha=\max(\alpha_i)$, with cell design based on the principle that $\alpha^s=0$ for s sufficiently large. This vanishing point(s) clearly depends on the word length of calculations and Table (4.4.4.2) gives an indication of its value. As the systems considered are normally of the order n>1000, $s<\lceil n/2 \rceil$ is a reasonable assumption when $0<|\alpha|<1$. Thus, the p-cyclic scheme is a fast area efficient design superior to all the others when n>>r and s. The only problem with the p-cyclic cell and its double pipe modification is that the backward recursive procedures adopted are unstable due to the division by α. In order to derive a stable p-cyclic cell further concepts developed in the next chapter are required.

Now so far we have been quite vague about the methods of factorization to produce (4.4.3) and (4.4.1.6). The BL and KH methods operate on the matrix A_c directly, whereas the techniques investigated here assume a factorisation has been completed before the solution starts. In Chen's method multiplying out (4.4.3) produces the non-linear system of equations,

$$\left.\begin{aligned}
&\beta_0^2+\beta_1^2+\cdots+\beta_{p-1}^2+\beta_p^2 = a_0\\
&\beta_0\beta_1+\beta_1\beta_2+\cdots+\beta_{p-2}\beta_{p-1}+\beta_{p-1}\beta_p = a_1\\
&\beta_0\beta_2+\beta_1\beta_3+\cdots+\beta_{p-3}\beta_{p-2}+\beta_{p-2}\beta_p = a_2\\
&\vdots \qquad\qquad\qquad\qquad\qquad\qquad \vdots\\
&\beta_0\beta_p \qquad\qquad\qquad\qquad\qquad\qquad = a_p
\end{aligned}\right\} \qquad (4.4.4.1)$$

which must be solved to produce β_i, i=1(1)r. Likewise in the Audish & Evans factorisation the form,

$$A_k = Q_k A_{k-1} Q_k^T \qquad (4.4.4.2)$$

with $A_k=A_c$, k=p+1 and A_{k-1} a modified matrix of semi-bandwidth k-1 such that,

$$Q_k = \begin{bmatrix} \gamma_k & 1 & & O \\ & \ddots & \ddots & \\ & & \ddots & 1 \\ O & & & \gamma_k \end{bmatrix} \qquad (4.4.4.3)$$

and

$$A_{k-1} = \begin{bmatrix} \gamma_1 & \gamma_2 \cdots \gamma_{k-1} & & \gamma_{k-1} \cdots \gamma_2 \\ \gamma_2 & & & \\ \vdots & & O & \\ \gamma_{k-1} & & & \gamma_{k-1} \\ & \ddots & & \vdots \\ & & & \gamma_2 \\ \gamma_{k-1} & O & & \\ \vdots & & & \\ \gamma_2 \cdots \gamma_{k-1} & & \cdots \gamma_2 & \gamma_1 \end{bmatrix} \qquad (4.4.4.4)$$

multiplying out (4.4.4.2) yields the nonlinear system,

$$\left.\begin{aligned} (\gamma_2+\gamma_1\gamma_k)\gamma_k + (\gamma_1+\gamma_2\gamma_k) &= a_1 \\ (\gamma_1+\gamma_2\gamma_k)\gamma_k + (\gamma_2+\gamma_3\gamma_k) &= a_2 \\ \cdots\cdots\cdots\cdots\cdots\cdots & \\ (\gamma_{k-3}+\gamma_{k-2}\gamma_k)\gamma_k+(\gamma_{k-2}+\gamma_{k-1}\gamma_k) &= a_{k-2} \\ (\gamma_{k-2}+\gamma_{k-1}\gamma_k)\gamma_k + \gamma_{k-1} &= a_{k-1} \\ \gamma_{k-1}\gamma_k &= a_k \end{aligned}\right\} \qquad (4.4.4.5)$$

relating the coefficients of A_k to the elements of Q_k and A_{k-1}. Solving (4.4.4.5) and repeating the process on A_{k-1} yields a sequence of k-1 factorisations producing (4.4.1.6) according to,

$$A_{k-z+1} = Q_{k-z+1}A_{k-z}Q^T_{k-z+1}, \quad z=1(1)k-1. \qquad (4.4.4.6)$$

requiring the solution of a non-linear system at each step.

Audish [81] discusses an iterative procedure to solve (4.4.4.1) by repeated computation of a (p+1)*(p+1) matrix vector problem, and (4.4.4.6) is solved by the Newton Raphson method using a (k-z+1)*(k-z+1) Jacobian matrix at each step. (See Audish & Evans [86]). Both methods converge quite quickly from good initial approximations, and the solutions of the coupled systems in (4.4.1.8) are stable as long as the A_{k-z} matrices are all strictly diagonally dominant.

For example put,

$$A_6 = (a_1, a_2, \ldots, a_6)$$

where $a_1 = 52614328$, $a_2 = 34066223$, $a_3 = 10825702$, $a_4 = 1801441$

$a_5 = 1501038$, $a_6 = 5040$,

so A_6 is factorisable yielding,

$$A_6 = Q_6 Q_5 Q_4 Q_3 Q_2 Q_2^T Q_3^T Q_4^T Q_5^T Q_6^T$$

The values of the Q_{r-k+1} are given by,

$$\gamma_6 = 8, \gamma_5 = 7, \gamma_4 = 6, \gamma_3 = 5, \gamma_2 = 1, \gamma_1 = 3$$

with each non-linear solution requiring five iterations. Normalizing (4.4.4.3) yields,

$$\alpha_6 = 1/8, \alpha_5 = 1/7, \alpha_4 = 1/6, \alpha_3 = 1/5, \alpha_2 = 1/3 ,$$

hence $\alpha = \max(\alpha_i) = 1/3$ and the number of s registers in the p-cyclic cell is obtained from Table (4.4.4.2) for some accuracy t.

A more realistic example is,

$$A_3 = (34, -16, 1)$$

and $$A_3 = Q_3 A_2 Q_3^T$$

yields $\gamma_3 = -1.7819687$, $\gamma_2 = -0.2032583$, $\gamma_1 = 2.7609056$,

with γ_2 and γ_1 the coefficients of A_2 requiring seven iterations. Using (4.4.4.9) with $a_0 = \gamma_1$ and $a_1 = \gamma_2$ and (4.4.4.10) indicates that $0 < |\alpha_i| < 1$ in our standard form (4.4.1.7).

When A_c has a quin or tri-diagonal circulant form the resulting non-linear equations are reduced to a special form which can be solved directly. For instance to produce (4.4.3) for a quindiagonal A_c (in compact form $A_c=(a_o,a_1,a_2)$), $\beta_o^{\frac{1}{2}}\tilde{L}$ is given by,

$$\left.\begin{aligned} a_o &= \beta_o^2+\beta_1^2+\beta_2^2 \\ a_1 &= \beta_o\beta_1+\beta_1\beta_2 \\ a_2 &= \beta_o\beta_2 \end{aligned}\right\} \qquad (4.4.4.7a)$$

and from Evans & Hadjidimos [80] it follows that,

$$\left.\begin{aligned} \beta_o &= d+e+f \\ \beta_1 &= 2(d-e) \\ \beta_2 &= d+e-f \end{aligned}\right\} \qquad (4.4.4.7b)$$

where,

$$\begin{aligned} 4d &= (a_o+2a_1+2a_2)^{\frac{1}{2}} \\ 4e &= (a_o-2a_1+2a_2)^{\frac{1}{2}} \\ 4f &= [2(a_o-6a_2)+\{(a_o+2a_2)^2-4a_1^2\}^{\frac{1}{2}}]^{\frac{1}{2}}. \end{aligned} \qquad (4.4.4.8)$$

Similarly, for tridiagonal systems,

$$\left.\begin{aligned} &\text{a)} \quad a_o = \beta_o^2+\beta_1^2 \\ &\text{b)} \quad a_1 = \beta_o\beta_1 \end{aligned}\right\} \qquad (4.4.4.9)$$

and substituting for β_1 in (4.4.4.9a) gives,

$$\beta_o^4-a_o\beta_o^2+a_1^2 = 0 ,$$

with the trivial solution,

$$\beta_o = [0.5\{a_o\pm(a_o^2-4a_1^2)^{\frac{1}{2}}\}]^{\frac{1}{2}} \qquad (4.4.4.10)$$

with β_1 given by (4.4.4.9b).

From (4.4.4.7) it follows that a quindiagonal A_c has the factorized form,

$$A_c = QQ^T , \qquad (4.4.4.11)$$

with,

$$Q = \begin{bmatrix} \beta_0 & & & \beta_1 & \beta_0 \\ \beta_1 & & & & \beta_2 \\ \beta_2 & & 0 & & \\ & 0 & & \beta_2 & \beta_1 & \beta_0 \end{bmatrix} \qquad (4.4.4.12)$$

and factorising Q such that,

$$Q = Q_1 Q_2 \ , \qquad (4.4.4.13)$$

and,

$$Q_1 = \begin{bmatrix} 1 & & & \gamma_2 \\ \gamma_2 & & 0 & \\ & 0 & & 1 \end{bmatrix} \ , \quad Q_2 = \begin{bmatrix} \gamma_0 & & & \gamma_1 \\ \gamma_1 & & 0 & \\ & 0 & & \gamma_0 \end{bmatrix}$$

gives,

$$\left.\begin{array}{ll} \text{a)} & \beta_0 = \gamma_0 \\ \text{b)} & \beta_1 = \gamma_1 + \gamma_2\gamma_0 \\ \text{c)} & \beta_2 = \gamma_2\gamma_1 \end{array}\right\} \qquad (4.4.4.14)$$

substituting for γ_0 and γ_2 in (4.4.4.14) yields the quadratic,

$$\gamma_1^2 - \gamma_1\beta_1 - \beta_0\beta_2 = 0 \ , \qquad (4.4.4.15)$$

giving,

$$\gamma_1 = 0.5\{\beta_1 \pm (\beta_1^2 + 4\beta_0\beta_2)^{\frac{1}{2}}\} \qquad (4.4.4.16)$$

substituting in (4.4.4.14) a,c produces,

$$A_c = Q_1 Q_2 Q_2^T Q_1^T \qquad (4.4.4.17)$$

which after normalising Q_2 produces the Audish & Evans factorisation. Likewise the Chen factorisation on a tridiagonal given by (4.4.4.9) produces an Audish & Evans form directly. We conclude that for restricted bandwidths the BATS pipeline competes very favourably with the BL and KH schemes, but for general bandwidths the overhead associated with the factorisation creates a serious disadvantage. This problem is compounded

by the difficulty in extending the techniques to skew-symmetric matrices, of the form,

$$A_{-c} = \begin{bmatrix} a_0 & a_{-1} & \cdots & a_{-p} & & a_1 & \cdots & a_p \\ a_1 & & & & & & & a_1 \\ \vdots & & & & & & & \\ a_p & & & & & & & a_{-p} \\ & & & & & & & \vdots \\ a_{-p} & & & & & & & a_{-1} \\ \vdots & & & & & & & \\ a_{-1} & \cdots & a_{-p} & & a_p & \cdots & a_1 & a_0 \end{bmatrix} \qquad (4.4.4.18)$$

which the BL and KH schemes solve easily. Again limited Chen type factorisations for quindiagonal and tridiagonal by direct methods are possible see Evans [83a],[80b], which extend to Audish and Evans type forms by a similar argument to that above. But generally a factorisation of the form (4.4.1.7) is not available.

Finally, the real advantage of these new factorisations is their utilisation of hardware. We have already seen that the O(n) and p-cyclic arrays are sensitive to bandwidth. But the Audish & Evans factorisation with the form (4.4.1.8) is also suitable for multipass computation, permitting an arbitrary number of cells to be used. Consequently when systems of order n>1000 are considered, and area usage is a premium the overhead in factorisation is compensated by the fact that BL and KH schemes require O(n) cells.

4.5 SUMMARY

In this chapter the concept of a soft-systolic algorithm and OCCAM simulation has been introduced by considering new designs for traditional algorithms which map easily into multi-layer environments. Section 4.1

introduced the double pipe a natural two layer design with planar sub-arrays on each level with local broadcasting between layers. The extension to general D^n-pipe and n level arrangements was dicussed, proving that systolic matrix vector computation was a limited version of a parallel tree arrangement already well known. In the form of a matrix product array the double pipe mapping extended to 2-D arrays illustrating that a two layer design reduced hardware while retaining the computation time of the traditional array.

Section 4.2 examined methods for overcoming feedback loop problems in factorisation and substitution (forward, backward recursive) schemes for solving linear systems. This led to block partitioning of array cells and explicit block computation. Block hex cells and inner products were introduced and shown to utilise hardware more efficiently than traditional schemes, while multi-layer layout was achieved at the internal cell level rather than global array organisation. While examination of the feedback loop length determined the optimal block size to be 3×3, showing that no improvement in computation could be gained for higher block sizes. An assessment of additional hardware required in arrays due to the inclusion of zero sub(super) diagonals of outer block diagonals was also considered. Finally, block schemes were used to produce efficient matrix product and factorisation arrays for complex matrix problems utilising implicit and explicit block computations respectively.

Section 4.3 considered the more difficult problem of matrix inversion and in particular rank annihilation. A special form of wavefront processor incorporating a Systolic Control Ring (SCR) was used to examine RANK-1 and RANK-2 wavefront schemes. Then a dedicated

systolic pipe utilising soft-systolic long wire heuristic was developed, which folded naturally into two levels to form a point to point systolic ring for repeated inverse updates. The new array reduced the number of true ips cells by trading them for delay cells, but could not compete with existing arrays for arbitrary matrix inversion in terms of speed. Arbitrary schemes requiring O(n) time for full inversion and the rank annihilator O(n) for a single update and $O(n^2)$ time for arbitrary inversion. However the rank annihilation array was well suited to multipass type computation while reducing hardware.

Section 4.4 considered the solution of circulant and Toeplitz matrices. These systems are interesting because fill-in during the factorisation process forces traditional arrays for dense matrices to be used even though the matrices have well defined sparsity. The rank annihilation array was adopted to construct a pipelined solver based on a new factorisation method rather than the Levinson and Bariess schemes already investigated. The main idea being to factorise the circulant matrix into a number of easily invertible circulant factors. Two algorithms were considered, the first producing circulant lower and upper triangular factors, the second replacing the triangular forms by a sequence of r bi-diagonal circulant matrix factors. Both methods could be solved on the same systolic pipeline with the latter scheme improving efficiency, throughput and computation time. An alternative was then developed removing the rank annihilator and making use of the special bi-diagonal structure of the second factorisation. This new array retained the throughput of the pipelined scheme, decreased computation time, and required a number of cells proportional to the matrix semi-bandwidth. Some simple modifications to this array using

special properties of the matrix yielded a more compact and faster array and extended to double pipe implementation. Finally, the new arrays were compared with existing methods. A significant factor in the comparison was that existing methods used the subject matrix directly while the new schemes required a factorisation before computation. The factorisation itself required the solution of non-linear systems of equations, making it difficult for the new arrays to compete even though the linear array was faster and used less ips cells. However the new arrays were suitable for multipass computations where area is the main consideration, (with computation time secondary,) the effects of the factorisation are not as significant making the new methods attractive.

CHAPTER 5

SYSTOLIC QUADRANT INTERLOCKING (QI) METHODS

"I saw four Angels standing at the four corners of the Earth ..."

Book of Revelations
(New Testament).

The recent rapid development of systolic arrays for problems in Linear Algebra has uncovered a variety of algorithms which produce different area time tradeoffs. However, this development of systolic algorithms has to date been largely restricted to existing sequential algorithms like the LU factorisation, Gaussian Elimination and QR-decomposition. There is no fundamental reason other than simplicity which indicates that a sequential to systolic algorithm conversion gives the best systolic array. Indeed, the BATS pipeline in Chapter 4 questioned this implicit assumption of array designers.

In this chapter we pose the following question 'are the Quadrant Interlocking (QI) methods better suited to systolic computation than basic sequential methods already employed?'

Intuitively QI methods are divided into three groups for solving linear systems:

1) Quadrant Interlocking Factorisations (QIF)

2) Quadrant Interlocking Eliminations (QIE)

3) Quadrant Interlocking Iteration (QII),

and have a number of advantages. Firstly, they are aimed at a parallel machine from the outset, and secondly, they have improved performance over other parallel implementations for solving linear systems. The QI algorithms themselves have been well studied, see Hatzopoulos [79], Shanehchi [80], Evans & Hadjidimos [80], Sojoodi-Haghighi [81], Evans & Sojoodi-Haghighi [82], Evans & Levin [85]. Our task here is to transfer these improvements to systolic array implementations.

5.1 SYSTOLIC QUADRANT INTERLOCKING FACTORISATIONS (SQIF)

Consider the linear system,

$$Ax = b \ , \tag{5.1.1}$$

where A is a non-singular n×n matrix and x and b are n×1 vectors with x unknown and b known. The basic idea of the QI factorisation is to factorise A such that,

$$A = WZ \ , \tag{5.1.2}$$

where it is observed that matrices W and Z have a Quadrant Interlocking structure defined by the relationships,

$$W_{ij} = \begin{cases} 1 & i=j \\ 0 & i=1(1)\lfloor n/2 \rfloor, \ j=(i+1)(1)(n-i+1) \\ 0 & i=m(1)n, \quad j=(n-i+1)(1)(i-1) \\ w_{ij} & \text{otherwise,} \end{cases} \tag{5.1.3}$$

and

$$Z_{ij} = \begin{cases} z_{ij} & i=1(1)\lfloor (n+1)/2 \rfloor, \ j=i(1)(n-i+1) \\ z_{ij} & i=m(1)n, \quad j=(n-i+1)(1)i \\ 0 & \text{otherwise,} \end{cases} \tag{5.1.4}$$

where $m=n+1-\lfloor n/2 \rfloor$ and $\lfloor \ell \rfloor$ denotes largest integer $<\ell$. For example, when n=5,

$$W = \begin{bmatrix} 1 & & 0 & & 0 \\ w_{21} & 1 & & 0 & w_{25} \\ w_{31} & w_{32} & 1 & w_{34} & w_{35} \\ w_{41} & 0 & & 1 & w_{45} \\ 0 & & 0 & & 1 \end{bmatrix} \ , \ Z = \begin{bmatrix} z_{11} & z_{12} & z_{13} & z_{14} & z_{15} \\ & z_{22} & z_{23} & z_{24} & \\ 0 & & z_{33} & & 0 \\ & z_{42} & z_{43} & z_{44} & \\ z_{51} & z_{52} & z_{53} & z_{54} & z_{55} \end{bmatrix}$$

and when n=4

$$W = \begin{bmatrix} 1 & 0 & 0 \\ w_{21} & 1 & 0 & w_{24} \\ w_{31} & 0 & 1 & w_{34} \\ 0 & 0 & 1 & \end{bmatrix} \ , \qquad Z = \begin{bmatrix} z_{11} & z_{12} & z_{13} & z_{14} \\ & z_{22} & z_{23} & \\ 0 & & & 0 \\ & z_{32} & z_{33} & \\ z_{41} & z_{42} & z_{43} & z_{44} \end{bmatrix}$$

(5.1.1) is then solved by substituting for A and solving the coupled systems,

$$\text{a)}\quad Wy = b\ , \qquad \text{b)}\quad Zx = y\ . \tag{5.1.5}$$

Now assuming no pivoting is required the computation of W and Z can be described as follows.

Let, $W=[w_1,w_2,\ldots,w_n]$ and $Z^T=[z_1,z_2,\ldots,z_n]$

with,

$$W_i = \begin{cases} \underbrace{[0,\ldots,0}_{i-1},1,w_{i+1,i},\ldots,w_{n-i,i},0,\ldots,0]^T & i=1(1)\left\lfloor\frac{n-1}{2}\right\rfloor \\ \underbrace{[0,\ldots,0}_{i-1},1,0,\ldots,0]^T & i=\begin{cases}\frac{n}{2} & n\text{-even},\\ \frac{n+1}{2} & n\text{-odd},\end{cases} \\ \underbrace{[0,\ldots,0}_{n-i+1}\,w_{n-i+2,i},\ldots,w_{i-1,i},1,0,\ldots,0]^T & i=\left\lfloor\frac{n+4}{2}\right\rfloor(1)n, \end{cases}$$

and

$$Z_i = \begin{cases} \underbrace{[0,\ldots,0}_{i-1}\,z_{ii},\ldots,z_{i,n-i+1},0,\ldots,0]^T & i=1(1)\left\lfloor\frac{n+1}{2}\right\rfloor \\ \underbrace{[0,\ldots,0}_{n-i}\,z_{i,n-i+1},\ldots,z_{ii},0,\ldots,0]^T & i=\left\lfloor\frac{n+3}{2}\right\rfloor(1)n. \end{cases}$$

Thus,

$$A = WZ = \sum_{i=1}^{n} W_i Z_i^T\ , \tag{5.1.6}$$

consequently A is factorised using at least $\lfloor(n-2)/2\rfloor$ stages consisting of computing two W_i and two Z_i vectors and updating a submatrix of A at each stage as follows. At the kth stage, the vectors,

$$[w_1,w_2,\ldots,w_{k-1},w_{n-k+2},\ldots,w_n]$$

are known and we denote,

$$A_k = A - \sum_{i=1}^{k-1} W_i Z_i^T - \sum_{i=n-k+2}^{n} W_i Z_i^T \tag{5.1.7}$$

it follows that the first and last k-1 rows and columns are zero hence,

$$\left.\begin{aligned} z_{kj} &= a_{kj}^{(k)} \\ z_{n-k+1,j} &= a_{n-k+1,j}^{(k)} \end{aligned}\right\} j=k,n-k+1 \qquad (5.1.8)$$

and

$$\begin{bmatrix} z_{kk} & z_{n-k+1,k} \\ z_{k,n-k+1} & z_{n-k+1,n-k+1} \end{bmatrix} \begin{bmatrix} w_{j,k} \\ w_{j,n-k+1} \end{bmatrix} = \begin{bmatrix} a_{jk}^{(k)} \\ a_{j,n-k+1}^{(k)} \end{bmatrix}, \qquad (5.1.9)$$

$$j=k+1(1)n-k$$

hence,

$$A_{k+1} = A_k - W_k Z_k^T - W_{n-k+1} Z_{n-k+1}^T . \qquad (5.1.10)$$

After factorisation the coupled system (5.1.5a) is solved as follows:

$$Wy = b \quad \Rightarrow \quad \sum_{i=1}^{n} y_i W_i = b^{(1)} \qquad (5.1.11)$$

and

$$\left.\begin{aligned} y_k &= b_k^{(k)} \\ y_{n-k+1} &= b_{n-k+1}^{(k)} \end{aligned}\right\} \quad k=1(1)\lfloor n/2 \rfloor \qquad (5.1.12)$$

where,

$$b^{(k)} = b^{(k-1)} - y_{k-1} w_{k-1} - y_{n-k+2} w_{n-k+2} . \qquad (5.1.13)$$

Similarly, (5.1.5b) is given by solving the 2×2 system,

$$\begin{bmatrix} z_{\ell-k+1,\ell-k+1} & z_{\ell-k+1,n-\ell+k} \\ z_{n-\ell+k,\ell-k+1} & z_{n-\ell+k,n-\ell+k} \end{bmatrix} \begin{bmatrix} x_{\ell-k+1} \\ x_{n-\ell+k} \end{bmatrix} = \begin{bmatrix} y_{\ell-k+1} \\ y_{n-\ell+k} \end{bmatrix} \qquad (5.1.14a)$$

where $\ell=\left\lfloor \frac{n-1}{2} \right\rfloor$, and setting

$$y_j = y_j - z_{j,\ell-k+1} x_{\ell-k+1} - z_{j,n-\ell+k} x_{n-\ell+k} \qquad (5.1.14b)$$

for $j=1(1)\ell-1$ and $j=(\ell+k+1)(1)n$.

Now the construction of a systolic array for this problem requires the representation of dataflow for the various 2×2 systems to be arranged to retain a regular communication structure. A natural type of basic cell based on 2×2 system operations suggests itself. However, the

locality of data varies from stage to stage during the factorisation. The first stage uses data in the four corners of A, and the last stage up to four adjacent neighbours at the centre of A. Consequently a data permutation is required to smooth out these locality variations.

Tylavsky [85] introduces a permutation on rows and columns of A which is admirably suitable for our intentions, and is illustrated below. When n=6

$$
\begin{array}{c}
\begin{array}{cccccc} 1 & 6 & 2 & 5 & 3 & 4 \end{array} \\
\begin{array}{c} 1 \\ 6 \\ 2 \\ 5 \\ 3 \\ 4 \end{array}
\left[\begin{array}{cc:cc:cc}
1 & 0 & & & & \\
0 & 1 & & & & \\
\hdashline
w_{21} & w_{26} & 1 & 0 & & \\
w_{51} & w_{56} & 0 & 1 & & \\
\hdashline
w_{31} & w_{36} & w_{32} & w_{35} & 1 & 0 \\
w_{41} & w_{46} & w_{42} & w_{45} & 0 & 1
\end{array}\right]
\left[\begin{array}{cc:cc:cc}
z_{11} & z_{16} & z_{12} & z_{15} & z_{13} & z_{14} \\
z_{61} & z_{66} & z_{62} & z_{65} & z_{63} & z_{64} \\
\hdashline
 & & z_{22} & z_{25} & z_{23} & z_{24} \\
 & & z_{52} & z_{55} & z_{53} & z_{54} \\
\hdashline
 & & & & z_{33} & z_{34} \\
 & & & & z_{43} & z_{44}
\end{array}\right]
= \bar{W}\bar{Z} = \bar{A}
\end{array}
\qquad (5.1.15)
$$

and for n=5

$$
\left[\begin{array}{cc:cc:c}
1 & 0 & & & \\
0 & 1 & & & \\
\hdashline
w_{21} & w_{25} & 1 & 0 & \\
w_{41} & w_{45} & 0 & 1 & \\
\hdashline
w_{31} & w_{35} & w_{32} & w_{34} & 1
\end{array}\right]
\left[\begin{array}{cc:cc:c}
z_{11} & z_{15} & z_{12} & z_{14} & z_{13} \\
z_{51} & z_{55} & z_{52} & z_{54} & z_{53} \\
\hdashline
 & & z_{22} & z_{24} & z_{23} \\
 & & z_{42} & z_{44} & z_{43} \\
\hdashline
 & & & & z_{33}
\end{array}\right]
= \bar{W}\bar{Z} = \bar{A} \qquad (5.1.16)
$$

where $\bar{W}$,$\bar{Z}$ and $\bar{A}$ are permuted forms of W,Z and A.

It immediately follows by multiplying out $\bar{W}$ and $\bar{Z}$, to produce $\bar{A}$ that the QIF method is a permuted form of 2×2 BLUF. Consequently the SQIF algorithm can be described by three steps:

Step (i) Permute A to $\bar{A}$

Step (ii) Pass $\bar{A}$ through 2×2 block array of Robert [85] to produce $\bar{W}$,$\bar{Z}$

Step (iii) Permute $\bar{W}$ and $\bar{Z}$ to produce W and Z.

As only simple row and column permutations are employed steps (i) and (iii) constitute host pre- and post-processing. This allows step (ii) to dominate computation costs when array input data is generated using pointers rather than explicit row and column interchanges in the host memory. Furthermore, the equivalence of 2×2 block schemes and QIF methods allows the use of the block partitioning theorems in Chapter 4. It follows that the improved performance of QIF schemes carries over to SQIF implementations.

The permutation technique also suggests that a variety of patterns other than the QI structure exist. Different patterns being produced by different permutations or by selecting larger block sizes. It is evident from Chapter 4 that increasing block size will not improve array performance for a SQIF method. New permutations on the other hand could yield new arrays but the SQIF form produces nearest neighbour data orderings which minimise communication problems and maximise efficiency. Soft-systolic arrays at present would be the only way of utilising non-local data orderings while maintaining array efficiency.

In establishing the above relationships we assumed that no pivoting was required during factorisation. From (5.1.8)-(5.1.10) it follows that the factorisation breaks down only if any of the values,

$$\det\left(\begin{bmatrix} \bar{z}_{kk} & z_{n-k+1,k} \\ z_{k,n-k+1} & z_{n-k+1,n-k+1} \end{bmatrix}\right) = 0 \qquad (5.1.17)$$

or equivalently,

$$a_{kk}^{(k)} a_{n-k+1,n-k+1}^{(k)} - a_{n-k+1,k}^{(k)} a_{k,n-k+1}^{(k)} = 0 \qquad (5.1.18)$$

as

$$\det(\bar{W}).\det(\bar{Z}) = \det(\bar{W}).0 = 0 = \det(\bar{A}) \qquad (5.1.19)$$

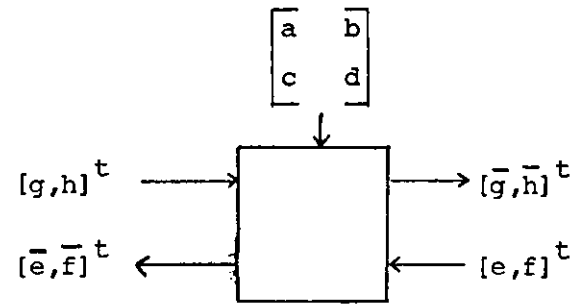
follows by applying definition (2.2.5) to the 2×2 block upper

triangular matrix $\bar{Z}$. Evans and Hatzopoulos [79] prove that a contradiction to (5.1.17) is always found by using pivoting as long as A (and hence A_k, $k=1(1)\lfloor(n-1)/2\rfloor$) is non-singular. Furthermore if A is diagonally dominant (5.1.17) never occurs and no pivoting is required. Applying the permutation method to yield a 2×2 block LU factorisation and extending Theorem (2.3.1) the uniqueness of the LU form yields the same results directly.

Now consider the solution of the coupled systems (5.1.5). The permuted form (5.1.15) indicates that $\bar{W}$ and $\bar{Z}$ are 2×2 block lower triangular and upper triangular forms respectively. It follows that (5.1.5) can be solved by a substitution array of the form in Fig. (3.2.2.1) incorporating 2×2 block ips cells. Modifying the recurrence (3.2.2.1) for block computations yields.

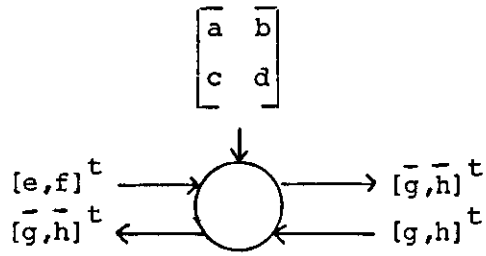
$$\left.\begin{aligned} Y_i^{(1)} &= [0,0]^t \\ Y_i^{(k+1)} &= Y_i^{(k)} + L_{ik}X_k \quad , \; i=1(1)\lceil n/2\rceil \\ X_i &= L_{ii}^{-1}(B_i - Y_i^{(i)}) \end{aligned}\right\} \qquad (5.1.20a)$$

where $B_i=[b_{2i-1},b_{2i}]^t$, $Y_i=[y_{2i-1},y_{2i}]^t$, and $X_i=[x_{2i-1},x_{2i}]$ and the block ips is defined by,



$$\left.\begin{aligned} \bar{e} &= e+(ag+bh) \\ \bar{f} &= f+(cg+dh) \\ \bar{g} &= g, \; \bar{h}=h \end{aligned}\right\} \qquad (5.1.20b)$$

and the boundary cell by,



$$\left.\begin{array}{l} D=(ad-bc)^{-1} \\ k=e-g,\ p=f-h \\ \bar{g}=D(dk-bp) \\ \bar{h}=D(ap-ck) \end{array}\right\} \quad (5.1.20c)$$

These operations can be implemented in a number of ways. The boundary cell is the most complex, but assuming the elements a,b,c,d are input on the same cycle its calculation can be pipelined to produce a cell cycle time bounded by a two point ips cycle, as follows,

t: $D=ad-bc$

t+1: $\bar{a}=a/D$, $\bar{b}=b/D$, $\bar{c}=c/D$, $\bar{d}=d/D$, $k=e-g$, $p=f-h$

t+2: $\bar{g}=\bar{d}k-\bar{b}p$, $\bar{h}=\bar{a}p-\bar{c}k$

and has area bounded by 10 point ips cells. The block ips cell can be implemented using the equivalent of 4 point ips cells using a single ips cycle plus the time for addition (giving a cycle bounded by two point ips cycles).

Theorem 5.1.1: A 2×2 block triangular solver computes Lx=y where L is a n×n lower triangular matrix of bandwidth q, in a time $T<2\{2\lceil n/2\rceil+\lceil q/2\rceil+1\}$ point ips cycles using at most $4\lceil q/2\rceil+6$ point ips cells.

Proof:

Using Theorem (3.2.2.1) and substituting $\lceil n/2\rceil$ and $\lceil q/2\rceil$ for the block form of L, and adding a cycle to allow D to be formed initially in the boundary cell gives $2\lceil n/2\rceil+\lceil q/2\rceil+1$. Multiply this timing by 2 as each cell requires at most two point ips cycles yields the upper bound T. There are $\lceil q/2\rceil$ block ips cells including the boundary cell. Allowing 4 point ips cells per block cell and an extra 6 for the boundary gives the area bound.

Overlapping block ips computations produces a further improvement. Notice that steps t+1 and t+2 of the pipelined boundary cell constitute the two ips cycles necessary for synchronisation. As a single point ips consists of multiply and add, two sequential additions or subtractions can be performed on one ips cycle. Now, staggering the calculation of k and p, such that k occurs in the first half cycle and p in the second half cycle, allows $\bar{e}$ and $\bar{f}$ block ips computations to be pipelined. That is,

$$\begin{aligned} t: &\quad t_0=ag+bh \\ t+1: &\quad \bar{e}=e+t_0, \quad t_1=cg+dh \\ t+2: &\quad \bar{f}=f+t_1 \end{aligned}$$

at the end of the second cycle $\bar{e}$ is ready and half a cycle later $\bar{f}$ is available. It follows that a,b and c,d as well as e,f can be input sequentially. Notice that g and h must be output in parallel and as k and p are available at the end of the same cycle is possible. This new arrangement does not alter the computation time of Theorem (5.1.1) but reduces cell count to $2\lceil q/2 \rceil+8$ as only two point ips cells are required in a block ips cell. Consequently a 2×2 block solver uses approximately the same hardware as the point case (with an overhead for the boundary cell), and computes with almost the same time.

An alternative approach to solving (5.1.5) is to notice that the inverse of the 2×2 system in (5.1.9) is computed explicitly during the factorisation. Using the permuted form of A this known inverse is utilised to generate the 2×2 block form,

$$\bar{A} = L\bar{D}U \qquad (5.1.21)$$

where for n=6,

$$\bar{A} = \begin{bmatrix} 1 & 0 & & & & \\ 0 & 1 & & & & \\ w_{21} & w_{26} & 1 & 0 & & \\ w_{51} & w_{56} & 0 & 1 & & \\ w_{31} & w_{36} & w_{32} & w_{35} & 1 & 0 \\ w_{41} & w_{46} & w_{42} & w_{45} & 0 & 1 \end{bmatrix} \begin{bmatrix} d_{11} & d_{16} & & & & \\ d_{61} & d_{66} & & & & \\ & & d_{22} & d_{25} & & \\ & & d_{52} & d_{55} & & \\ & & & & d_{33} & d_{34} \\ & & & & d_{43} & d_{44} \end{bmatrix}$$

$$L \qquad\qquad \bar{D}$$

$$\begin{bmatrix} 1 & 0 & z_{12} & z_{15} & z_{13} & z_{14} \\ 0 & 1 & z_{62} & z_{65} & z_{63} & z_{64} \\ & & 1 & 0 & z_{23} & z_{24} \\ & & 0 & 1 & z_{53} & z_{54} \\ & & & & 1 & 0 \\ & & & & 0 & 1 \end{bmatrix}$$

$$U$$

and $\bar{D}$ is composed of the diagonal blocks of $\bar{Z}$ and $U=\bar{D}^{-1}\bar{Z}$ with $L=\bar{W}$. Substituting (5.1.21) in the permuted form of (5.1.1) then yields the coupled systems,

$$\text{a) } L\bar{y}=\bar{b}, \quad \text{b) } \bar{D}\bar{s}=\bar{y}, \quad \text{c) } U\bar{x}=\bar{s} \tag{5.1.22}$$

and the simple point ips substitution array in Fig.(3.2.2.1) can be used directly to solve (5.1.22a, and c). (5.1.22b) is simply solved by inverting $\bar{D}$, which is known from the generation of U, and performing 2×2 matrix-vector calculations.

Finally we remark that (5.1.21) must not be confused with the alternative factorisation

$$A = WDZ \tag{5.1.23}$$

in Shanehchi [80] where (5.1.9) is replaced by the relations,

$$\begin{bmatrix} a_{kk}^{(k)} & a_{n-k+1,k}^{(k)} \\ a_{k,n-k+1}^{(k)} & a_{n-k+1,n-k+1}^{(k)} \end{bmatrix} \begin{bmatrix} w_{jk} \\ w_{j,n-k+1} \end{bmatrix} = \begin{bmatrix} a_{jk}^{(k)} \\ a_{j,n-k+1}^{(k)} \end{bmatrix} \tag{5.1.24}$$

$$\left.\begin{aligned} z_{kj} &= a_{kj}^{(k)} / a_{kk}^{(k)} \\ z_{n-k+1,j} &= a_{n-k+1,j}^{(k)} / a_{n-k+1,n-k+1}^{(k)} \end{aligned}\right\} j=k(1)n-k+1 \tag{5.1.25}$$

and

$$\left.\begin{aligned} d_{kk} &= a_{k,k} \\ d_{n-k+1,n-k+1} &= a_{n-k+1,n-k+1} \end{aligned}\right\} \tag{5.1.26}$$

Here D is a true point diagonal matrix. Applying the QI permutation produces,

$$\bar{A} = \bar{W}\hat{D}\bar{Z} \tag{5.1.27}$$

where $\bar{W}$ has the same structure as in (5.1.15), $\hat{D}$ is a permuted point diagonal matrix and $\bar{Z}$ has a similar form to that in (5.1.15) except that $z_{ii}=1$ $i=1(1)n$. Hence solving the associated permuted coupled system,

$$\text{a)}\ \ \bar{W}\bar{y} = \bar{b},\ \ \text{b)}\ \hat{D}\bar{s} = \bar{y},\ \ \text{c)}\ \bar{Z}\bar{x} = \bar{s} \tag{5.1.28}$$

requires a 2×2 block triangular solver and no advantage is gained over the original QIF method of (5.1.2). Indeed the solution process is more complex and is not offset by hardware savings which utilise the property $z_{ii}=1$ for $i=1(1)n$.

5.2 A MODIFICATION OF THE QIF METHOD

Now consider an alternative method for solving (5.1.1) where the QI factors in (5.1.2) have the form,

$$W = \begin{bmatrix} 1 & 0 & 0 & 0 \\ w_{21} & 1 & 0 & w_{24} \\ w_{31} & w_{32} & 1 & w_{34} \\ w_{41} & 0 & 0 & 1 \end{bmatrix}, \qquad Z = \begin{bmatrix} z_{11} & z_{12} & z_{13} & z_{14} \\ 0 & z_{22} & z_{23} & 0 \\ 0 & 0 & z_{33} & 0 \\ 0 & z_{42} & z_{43} & z_{44} \end{bmatrix}$$

for n=4 and,

$$W = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ w_{21} & 1 & 0 & 0 & w_{25} \\ w_{31} & w_{32} & 1 & w_{34} & w_{35} \\ w_{41} & w_{42} & 0 & 1 & w_{45} \\ w_{51} & 0 & 0 & 0 & 1 \end{bmatrix} \quad , Z = \begin{bmatrix} z_{11} & z_{12} & z_{13} & z_{14} & z_{15} \\ 0 & z_{22} & z_{23} & z_{24} & 0 \\ 0 & 0 & z_{33} & 0 & 0 \\ 0 & 0 & z_{43} & z_{44} & 0 \\ 0 & z_{52} & z_{53} & z_{54} & z_{55} \end{bmatrix}$$

for n=5. More formally let,

$$W = [w_1, w_2, \ldots, w_n] \text{ and } Z^T = [z_1, z_2, \ldots, z_n]$$

as before and,

$$W_i = \begin{cases} [\underbrace{0,\ldots,0}_{i-1}\ 1\ w_{i+1,i},\ldots,w_{n-i+1,i}\ 0,\ldots,0]^t & i=1(1)\lfloor n/2 \rfloor \\ [\underbrace{0,\ldots,0}_{i-1}\ 1\ 0,\ldots,0]^T & i=\lfloor n/2 \rfloor+1 \\ [\underbrace{0,\ldots,0}_{n-i+1}\ w_{n-i+2,i},\ldots,w_{i-1,i}\ 1\ 0,\ldots,0]^T & i=\lfloor n/2 \rfloor+2(1)n \end{cases} \tag{5.2.1}$$

$$Z_i = \begin{cases} [\underbrace{0,\ldots,0}_{i-1}\ z_{ii},\ldots,z_{i,n-i+1},\ 0,\ldots,0]^T & i=1(1)\left\lfloor \frac{n+1}{2} \right\rfloor \\ [\underbrace{0,\ldots,0}_{n-i+1}\ z_{i,n-i+2},\ldots,z_{ii},\ 0,\ldots,0]^T & i=\left\lfloor \frac{n+1}{2} \right\rfloor+1(1)n. \end{cases} \tag{5.2.2}$$

applying (5.1.6) yields the relation,

$$\left.\begin{aligned} A_1 &= A \\ A_k &= A - \sum_{i=1}^{k-1} W_i Z_i^T - \sum_{i=n-k+2}^{n} W_i Z_i^T \quad k=2(1)\left\lfloor \frac{n-1}{2} \right\rfloor \end{aligned}\right\} \tag{5.2.3}$$

as before and the elements of the kth stage in the kth and n-k+1 st column can be found by the procedure,

$$
\left.\begin{array}{ll}
\text{a)} & z_{k,j} = a^{(k)}_{kj} \quad , \; j=k(1)n-k+1 \\
\text{b)} & w_{j,k} = a^{(k)}_{jk}/z_{kk}, \; j=k+1(1)n-k+1 \\
\text{c)} & z_{n-k+1,j} = a^{(k)}_{n-k+1,j} - w_{n-k+1,k} z_{kj}, \; j=k+1(1)n-k+1 \\
\text{d)} & w_{j,n-k+1} = (a^{(k)}_{j,n-k+1} - w_{jk} z_{k,n-k+1})/z_{n-k+1,n-k+1}, \\
 & \qquad\qquad\qquad\qquad\qquad\qquad j=k+1(1)n-k \\
\text{and} \quad \text{e)} & A_{k+1} = A_k - W_k Z_k^T - W_{n-k+1} Z_{n-k+1}^T \; .
\end{array}\right\} \quad (5.2.4)
$$

Now applying the QI permutation produces locally placed elements for simplified dataflow and yields the following interesting 2×2 block partitioning.

For n=6

$$
\bar{W} = \begin{array}{c|cc|cc|cc}
 & 1 & 6 & 2 & 5 & 3 & 4 \\
\hline
1 & 1 & 0 & & & & \\
6 & w_{61} & 1 & & & & \\
\hline
2 & w_{21} & w_{26} & 1 & 0 & & \\
5 & w_{51} & w_{56} & w_{52} & 1 & 0 & \\
\hline
3 & w_{31} & w_{36} & w_{32} & w_{35} & 1 & 0 \\
4 & w_{41} & w_{46} & w_{42} & w_{45} & w_{43} & 1
\end{array}
\qquad
\bar{Z} = \begin{array}{cc|cc|cc}
1 & 6 & 2 & 5 & 3 & 4 \\
\hline
z_{11} & z_{16} & z_{12} & z_{15} & z_{13} & z_{14} \\
0 & z_{66} & z_{62} & z_{65} & z_{63} & z_{64} \\
\hline
 & & z_{22} & z_{25} & z_{23} & z_{24} \\
 & & 0 & z_{55} & z_{53} & z_{54} \\
\hline
 & & & & z_{33} & z_{34} \\
 & & & & 0 & z_{44}
\end{array}
$$

Notice that the diagonal blocks contain extra sparsity. The above block partitioning can be termed implicit because without the partition lines the permuted W and Z appear to be simple L and U factors. As the point LU factorisation is unique, it follows that the hexagonal (point) array of H.T. Kung and Leiserson (see Fig.3.2.2.3) can be applied to find the modified QIF of a matrix by applying simple pre- and post-permutations on the input and output. Likewise, the form of the permuted coupled systems,

a) $Wy = b$ b) $Zx = y$, (5.2.5)

corresponding to the point systems in (2.3.3.2) allows normal point triangular solver arrays like Fig.(3.2.2.1) to be employed to solve (5.1.1). Comparing Theorems (3.2.2.3) and (4.2.2.1) it appears that the modified QIF has no advantages in speed or efficiency over the ordinary QIF (or explicit block computation) regarding the application of systolic arrays. However this argument is based on a comparison of point and block methods using point-(ips) on block-(ips) structured arrays. The essential feature of the modified QIF is its implicit block structured nature which allows a block structured array to perform point-like computations on implicitly block structured data; thereby retaining the improved efficiency and reduced computation time of the explicit block schemes but producing point (or implicit block) structured outputs. This resolves the difficulties associated with the ordinary QIF scheme, which requires an explicitly block structured triangular solver, or the more complex LDU factorisation (requiring the solution of three coupled systems) using implicit block (point) structured solvers.

The implicit block structured array for the modified QIF uses the same principles as the 2×2 block array in Robert [85] but utilises the structure of $\overline{W}$ and $\overline{Z}$ diagonal blocks to adjust hardware and computation within each block ips cell. For the development of the array we make the following simple assumptions:

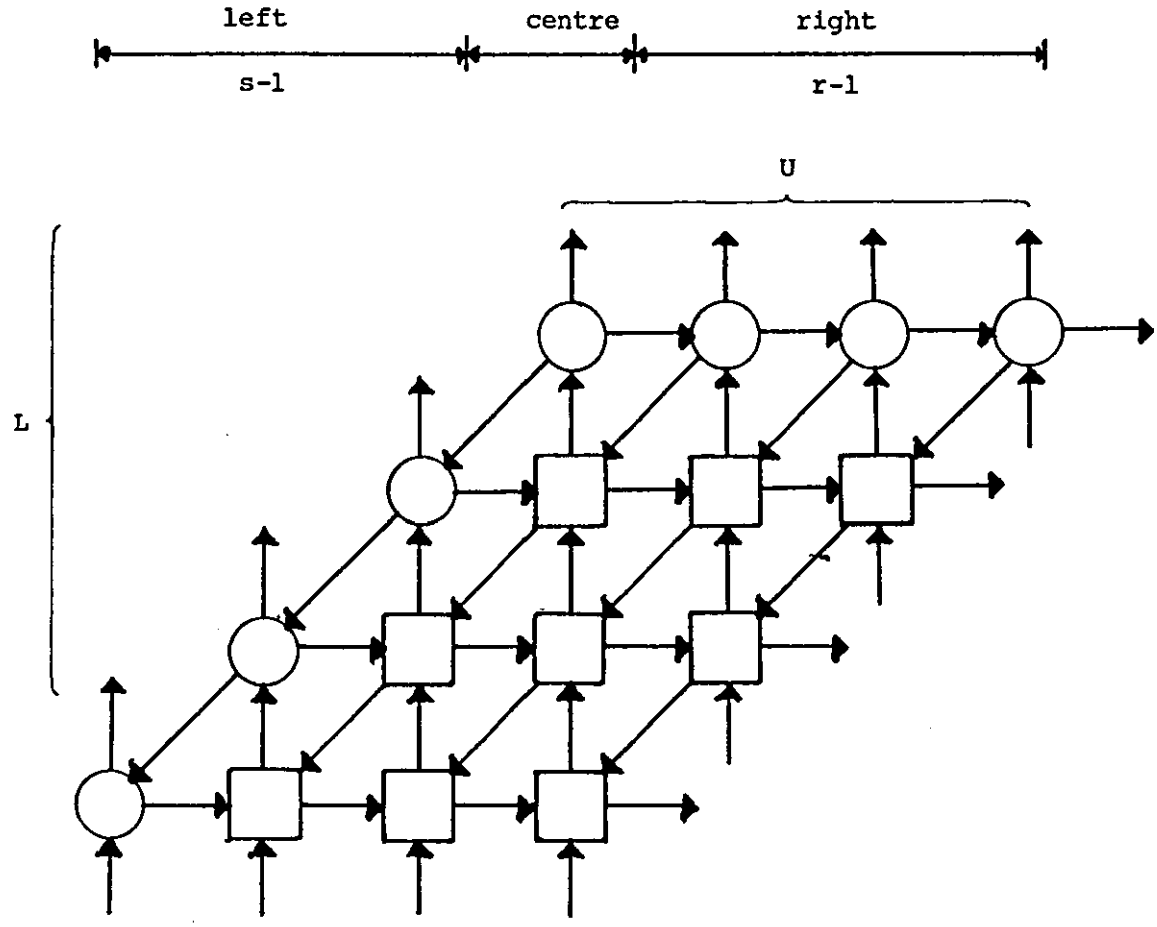
(i) The n×n permuted input matrix $\overline{A}$ does not cause a breakdown in the factorisation process (i.e. diagonally dominant or positive definite).

(ii) $n=2m$ for $m<n$ (i.e. ensures an even partitioning).

(iii) $\overline{A}$ is banded with bandwidth $w=p+q-1$.

The global structure of the array is shown in Fig.(5.2.1) and contains rs 2×2 implicit block ips cells where,

$$p = \begin{cases} 2r-1 \\ 2r-2 \end{cases}, \qquad q = \begin{cases} 2s-1 \\ 2s-2 \end{cases} . \tag{5.2.6}$$



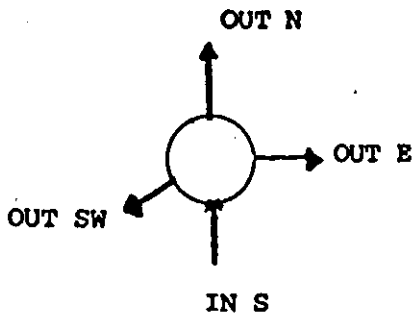
(Input format same as Fig.(4.2.2.1)

FIGURE 5.2.1: Implicit block array for modified QIF method

Each implicit block cell contains the equivalent of 2×2=4 point ips cells and computes as follows:-

Cell definitions:

Centre

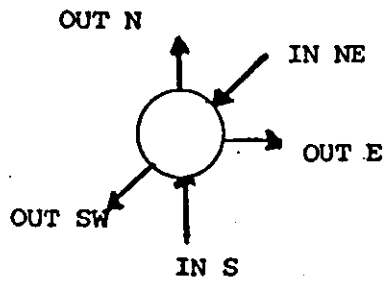


t: IN S a,b,c,d
w=c/a

t+1: OUT N a,b
OUT E w
OUT SW a,b
d=d-wb

t+2: OUT N w,d
OUT E d
OUT SW d

Left



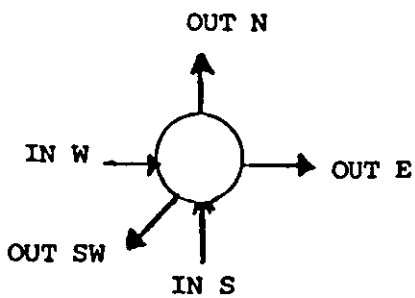
t: IN S a,b,c,d
IN NE z_1,z_2
$w_1=a/z_1$, $w_2=c/z_1$

t+1: OUT E w_1,w_2
OUT SW z_1,z_2
IN NE z_3
$w_3=b-w_1z_2$
$w_4=d-w_2z_2$

t+2: OUT E w_3,w_4
OUT SW z_3
OUT N w_1,w_2
$w_3=w_3/z_3$, $w_4=w_4/z_3$

t+3: OUT N w_3,w_4

Right



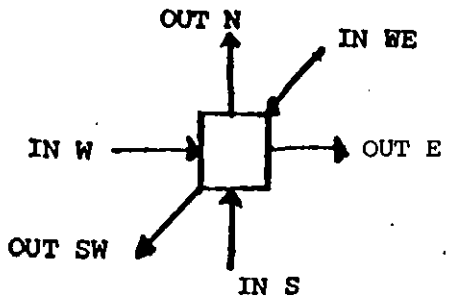
t: INS a,b,c,d
INW w
c=c-wa
d=d-wb

t+1: INW D
OUT E w
OUT SW a,b
$\overline{c}=c/D$
$\overline{d}=d/D$

t+2: OUT N a,b
OUT SW $\overline{c},\overline{d}$
OUT E E,D

t+3: OUT N c,d

Implicit Block Cell



t: IN S a,b,c,d
OUT N $\bar{a},\bar{b},\bar{c},\bar{d}$

t+1: -

t+2: IN W w_1,w_2
IN NE z_1,z_2

$$\begin{bmatrix} \bar{a} & \bar{b} \\ \bar{c} & \bar{d} \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} - \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} [z_1 z_2]$$

t+3: OUT E w_1,w_2
OUT SW z_1,z_2
IN NE z_3,z_4
IN W w_3,w_4

$$\begin{bmatrix} \bar{a} & \bar{b} \\ \bar{c} & \bar{d} \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} - \begin{bmatrix} w_3 \\ w_4 \end{bmatrix} [z_3,z_4]$$

Each cell receives four inputs corresponding to an implicit block every four point ips cycles, this being the longest period of any block cell. As there are m=n/2 implicit blocks of input data the total output time for data is 4m=2n point ips cycles. Output/computation starts after the first implicit block has reached the center processor. As each implicit block cell requires two point ips cycles for computation data is shifted into its initial starting position in at most $2\min(r,s) \leq \min(p,q)$. Hence we have the following theorem.

Theorem (5.2.1): The modified QIF of an $n \times n$ matrix A whose QI permutation matrix $\bar{A}$ has bandwidth w=p+q-1 can be computed in T=2n+min(p,q) ips cycles using approximately pq point ips cells.

This is identical to the result in Robert [85], and arises from the fact that the global dataflow of the two arrays are the same.

Examining the computation of the implicit cells, reveals that calculations occur two cycles in four and the efficiency e=½ achieved in Robert [85] is also preserved.

5.3 RESTRICTED FORMS OF SYSTOLIC QI SCHEMES

If the matrix A in (5.1.1) is a real symmetric positive definite matrix a better factorisation of the form,

$$A = WDW^T \tag{5.3.1}$$

exists, and by use of symmetry the modified SQIF method is reduced to the following procedure:

$$\left.\begin{aligned}
&\text{a)}\quad d_k = a_{kk}^{(k)} \\
&\text{b)}\quad w_{jk} = a_{jk}^{(k)}/d_k\ , \quad j=(k+1)(1)n-k+1 \\
&\text{c)}\quad d_{n-k+1} = a_{n-k+1,n-k+1}^{(k)} - w_{n-k+1,k}\, a_{n-k+1,k}^{(k)} \\
&\text{d)}\quad w_{j,n-k+1} = (a_{j,n-k+1}^{(k)} - w_{jk}\, a_{n-k+1,k}^{(k)})/d_{n-k+1}\ , \quad j=k+1(1)n-k \\
&\text{e)}\quad A_{k+1} = A_k - d_k W_k W_k^T - d_{n-k+1} W_{n-k+1} W_{n-k+1}^T
\end{aligned}\right\} \tag{5.3.2}$$

substituting (5.3.1) for A in (5.1.1) produces the three coupled systems,

$$\text{a)}\quad Wy = b, \quad \text{b)}\quad Du = y, \quad \text{c)}\quad W^T x = u\ , \tag{5.3.3}$$

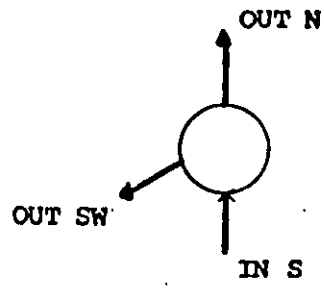
where W has the form in (5.2.1) and D is a diagonal matrix. After applying the QI permutation (5.3.1) can be written as,

$$\bar{A} = L\bar{D}L^T\ , \tag{5.3.4}$$

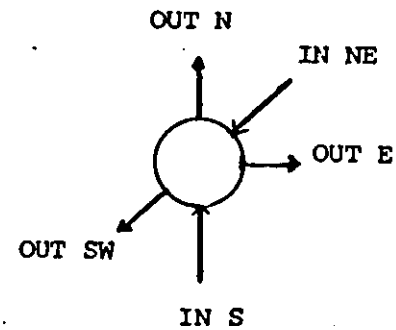
where $L=\bar{W}$ and $\bar{D}=D$ the permuted forms of W and D respectively. The associated coupled systems,

$$\text{a)}\quad L\bar{y} = \bar{b}, \quad \text{b)}\quad \bar{D}\bar{u} = \bar{y}, \quad \text{c)}\quad \bar{L}^T\bar{x} = \bar{u}\ , \tag{5.3.5}$$

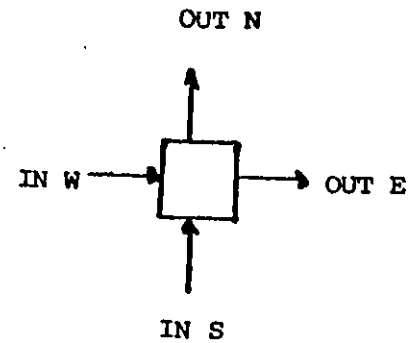
complete the conversion indicating an implicit block form for the root-free Choleski factorisation. The global structure of the array is shown in Fig.(5.3.1) and computes according to the following cell definitions.



t: IN S a,b,c,d
w=c/a
t+1: OUT SW a,c
OUT N a,w
D=d-wc, t=t*a
t+2: OUT N D
OUT SW D
t=t*d
t+3: OUT N d



t: IN S a,b,c,d
IN E d_1,x
$w_1=a/d$, $w_2=c/d_1$
t+1: OUT SW d_1,x
OUT E a,c
IN NE D
$\bar{w}_3=(b-w_1x)$
$\bar{w}_4=(d-w_2x)$
t+2: OUT N w_1,w_2
OUT SW D
OUT E w_1,w_2,$\bar{w}_3$,$\bar{w}_4$
$w_3=\bar{w}_3/D$
$w_4=\bar{w}_4/D$
t+3: OUT E w_3,w_4
OUT N $\bar{w}_3$,$\bar{w}_4$

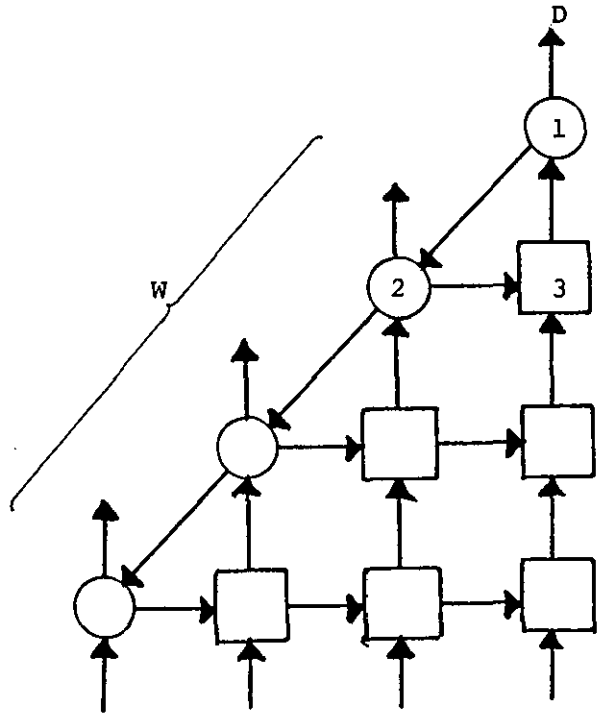


t: IN S a,b,c,d
OUT N $\bar{a}$,$\bar{b}$,$\bar{c}$,$\bar{d}$
t+1: -
t+2: IN W e,f
t+3: IN W w_1,w_2,$\bar{w}_3$,$\bar{w}_4$

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} - \begin{bmatrix} e \\ f \end{bmatrix} [w_1, w_2]$$

t+4: IN W w_3,w_4
OUT E w_1,w_2

$$\begin{bmatrix} \bar{a} & \bar{b} \\ \bar{c} & \bar{d} \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} - \begin{bmatrix} \bar{w}_3 \\ \bar{w}_4 \end{bmatrix} [w_3, w_4]$$



$$
\begin{array}{cccc}
 & & & a_{11}a_{18}a_{81}a_{88} \\
 & & & - \\
 & & a_{21}a_{28}a_{71}a_{78} & - \\
 & & - & - \\
 & a_{31}a_{38}a_{61}a_{68} & - & - \\
 & - & - & a_{22}a_{27}a_{72}a_{77} \\
a_{41}a_{48}a_{51}a_{58} & - & - & - \\
 & - & a_{32}a_{37}a_{62}a_{67} & - \\
 & - & - & - \\
 & a_{42}a_{47}a_{52}a_{57} & - & - \\
 & & - & a_{33}a_{36}a_{63}a_{66} \\
 & & - & - \\
 & & a_{43}a_{46}a_{53}a_{56} & - \\
 & & & - \\
 & & & - \\
 & & & a_{44}a_{45}a_{54}a_{55}
\end{array}
$$

$T = \frac{5n}{2} + 2s$ $s=n/2$ for a full matrix

NOTE: the orthogonal connections

FIGURE 5.3.1: Implicit block array for symmetric positive definite factorisation

The array input format is easily derived from the minimum path length of the feedback loop passing through cells 1,2 and 3, which implies that successive data blocks are separated by five cycles. Adopting the same assumptions as the modified SQIF and using the symmetry of A gives $\bar{A}$ a bandwidth of w=2q-1, and from (5.2.6) the array contains $\frac{s}{2}(s+1)$ implicit block cells. Finally, using analogous reasoning to that producing Theorem (5.2.1) we derive,

Theorem (5.3.1): The modified QIF of a real symmetric positive definite n×n matrix A can be formed by computing a root-free Choleski factorisation of the QI permuted matrix $\bar{A}$ of bandwidth w=2q-1 on a 2×2 implicit block array with $\frac{1}{2}(q+u)(q+u+2)$ ips cells in T=2.5n+(q+u) point ips cycles where $3>u\geq 0$.

Proof:

From the assumptions in Section (5.2) m=n/2, giving a total block data length of 5m=2.5n. Computation begins when the first block reaches cell 1 in Fig.(5.3.1) adding an additional delay of 2s ips cycles. Summing these times and employing (5.2.6) yields T=2.5n+(q+u), where u=0 when q=2s, u=1 for q=2s-1 and u=2 when q=2s-2.

Each implicit block cell has four point ips equivalents giving 2s(s+1) point ips cells in all. Now when q=2s we require $\frac{1}{2}q(q+2)$ point cells, with q=2s-1, $\frac{(q+1)}{2}(q+3)$ cells and for q=2s-2 $\frac{1}{2}(q+2)(q+4)$ cells. Thus $\frac{1}{2}(q+u)(q+u+2)$ is a generalised area estimate.

We are now in a position to justify our initial assumptions in array construction. First of all we assumed no breakdown during factorisations. Evans & Hadjidimos [80] derive these conditions from the QIF methods and support the results for point LU factorisations given in Theorems (2.3.1)-(2.3.2). Next we assumed n=2m m>0 this was convenient

because it avoided partitionings which produce isolated rows and columns, for example when n is odd. The single elements associated with these special rows and columns requires separate treatment and consequently requires cell modifications. For instance, cells must be able to detect the special elements and switch to point orientated calculations using extra switching hardware (or program code). Fortunately, the task is simplified by the QI permutation which pushes the special elements to the end of the input stream where they can be represented by implicit blocks of the form,

$$\begin{bmatrix} x & x \\ \delta & \delta \end{bmatrix}, \quad \begin{bmatrix} x & \delta \\ \delta & \delta \end{bmatrix}, \quad \begin{bmatrix} x & \delta \\ x & \delta \end{bmatrix}, \quad x=\text{matrix element}$$

for row, diagonal and column elements respectively. Neutral elements can then be used to good effect in simplifying internal cell changes, and a control tag adopted to mark the blocks.

Finally consider the effect of the QI permutation on the matrix structure. We assumed $\overline{A}$ was banded and the examples below show this is a reasonable assumption.

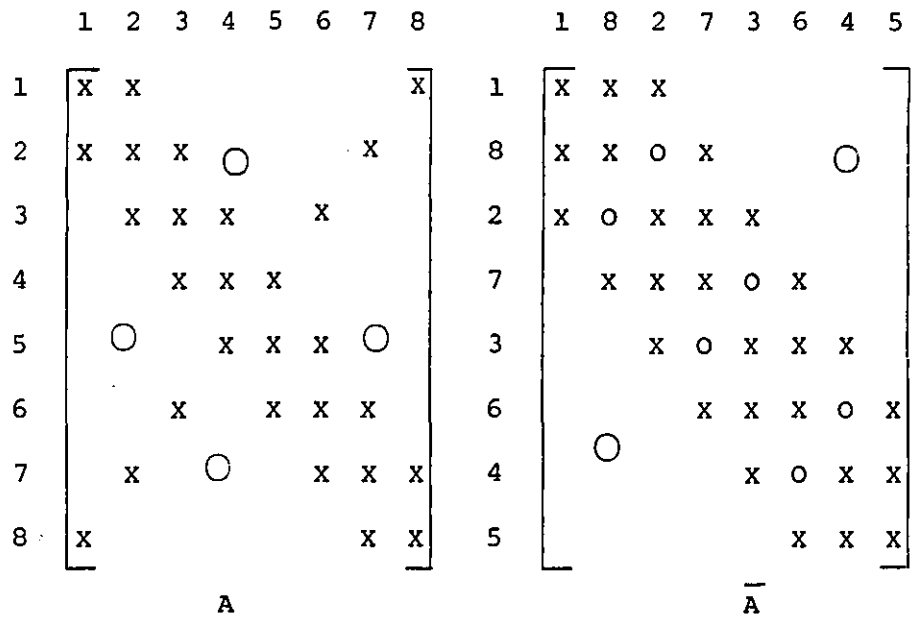
Case (i): When A is banded with bandwidth w, $\overline{A}$ has bandwidth 2w+1, e.g.

$$\begin{array}{c|cccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ \hline 1 & X & X & X & & & & & \\ 2 & X & X & X & X & & & & \\ 3 & X & X & X & X & X & & & \\ 4 & & X & X & X & X & X & & \\ 5 & & & X & X & X & X & X & \\ 6 & & & & X & X & X & X & X \\ 7 & & & & & X & X & X & X \\ 8 & & & & & & X & X & X \end{array} \quad A$$

$$\Rightarrow \quad \begin{array}{c|cccccccc} & 1 & 8 & 2 & 7 & 3 & 6 & 4 & 5 \\ \hline 1 & X & O & X & O & X & O & O & O \\ 8 & O & X & O & X & O & X & O & O \\ 2 & X & O & X & O & X & O & X & O \\ 7 & O & X & O & X & O & X & O & X \\ 3 & X & O & X & O & X & O & X & X \\ 6 & O & X & O & X & O & X & X & X \\ 4 & O & O & X & O & X & X & X & X \\ 5 & O & O & O & X & X & X & X & X \end{array} \quad \overline{A}$$

Case (ii): When A is full. $\bar{A}$ is full and the permutation is a special case of banded systems where the dense bottom right corner extends over the whole matrix.

Case (iii): If A is an X-band matrix of forward bandwidth W_1 and backward bandwidth W_2, $\bar{A}$ has bandwidth $W=W_1+W_2+1$, e.g.,



REMARK: Observe that in each case if A is symmetric so is $\bar{A}$ which colloborates the results of Theorem (5.3.1).

It is trivial to see that any matrix form fits into one of these categories by allowing bands to contain some zero entries.

Case (i) indicates that applying the QI permutation to an already banded system extends the bandwidth which increases the number of block cells in the SQIF array (by a factor of 4). Optimising internal cell requirements using the extra sparsity of permuted blocks does not compensate. However the relation between QI and LU factorisations indicates that A can be passed through the SQIF arrays to yield the 2×2 or point LU factors directly; thus retaining minimum bandwidth and array dimensions. By extension when A is full, both permuted and

unpermuted schemes can be employed on the same array. Finally in case (iii) the factorisation of A requires an effective bandwidth of w=2n-1 to allow for the fill-in of the matrix factors, producing a large SQIF array. Applying the QI permutation folds the fill-in into a reduced effective bandwidth of $w=w_1+w_2+1$ allowing a compact SQIF array to be employed. We conclude that it is necessary to form the QI permutation only when A has a X-band form. Thus, the QIF methods are factorised counterparts of the double pipe methods in Chapter 4.

5.4 INTERLUDE: THE BATS CELL REVISITED

At the expense of a slight digression recall the Audish & Evans [85] factorisation in (4.4.1.6) and the associated coupled systems of (4.4.1.8), for solving a circulant matrix. Each coupled system required the solution of a matrix A or A^T where,

$$A = \begin{bmatrix} 1 & & & \alpha \\ \alpha & & O & \\ & & & \\ & O & & \alpha \\ & & & 1 \end{bmatrix} \qquad (5.4.1)$$

In Section(4.4) two constrasting cell designs were developed; the O(n) and p-cyclic cells respectively, and incorporated into a pipelined linear array solver (BATS). Clearly (5.4.1) has the X-band structure indicating that the QI permutation can be applied effectively. In this section we suggest improvements to the O(n) and p-cyclic BATS cells to improve throughput and stability using the QI methods.

5.4.1 Improvements to the O(n) BATS Cell

Principally there are two main drawbacks with the existing O(n) BATS cell:

(i) All the data is entered on only one input line, even though two are available (see Fig.(4.4.2.1a)).

(ii) The combinations of LIFO and FIFO storage reduces throughput and restricts problem size.

In the first case data and parameters defining the coupled system travel on one input, while cell addresses for loading parameters travel on the other. As computation is mainly sequential the right-hand side vector of the circulant system (4.4.2.2) cannot be shared between the two lines. In the second case L/F storage is LIFO or FIFO depending on a hardwired control, and the size of storage (say n) places an upper bound on the maximum problem size solvable as follows. If A in (5.4.1) was an $\bar{n}\times\bar{n}$ matrix with $\bar{n}>n$ and a particular BATS cell L/F stores operated in LIFO mode at the end of input $\bar{n}-n$ stored elements would have been overwritten. The LIFO operation also determines throughput as the store requires at least 2n cycles to fill up and then empty. The pipelined loading of parameters on the other hand determines a lower bound on problem size. For example when the L/F stores and problem are both of size n the parameters are piped through the store and output before a LIFO or FIFO mode begins. But, when the problem size is $\bar{n}<n$ and a LIFO or FIFO operation begins some parameters are caught in the stores. Consequently the cell generates erroneous results and destroys parameters, causing subsequent cell computations to be invalid. It follows that the O(n) BATS cell can compute most efficiently for only one problem size (i.e., n). For problems with $\bar{n}<n$ input data must be padded with dummy parameters, placed between the real parameters and right hand side data, to achieve the correct synchronisation. In addition, the dummy parameters must have invalid

addresses associated with them to prevent erroneous loading of cells.

These problems can be overcome by converting all FIFO type data movements into LIFO operations, and throughput is increased by reducing the individual LIFO sizes. To achieve this we apply a pseudo-QI formulation on the system,

$$Au = d \ , \tag{5.4.1.1}$$

by interpreting the factorisation (4.4.2.3) as the system,

$$PWu = d \ , \tag{5.4.1.2}$$

and solving the coupled systems,

$$\text{a)} \quad Pv = d \ , \quad \text{b)} \quad Wu = v \ , \tag{5.4.1.3}$$

where P is unchanged but W is interpreted as a sparse 'butterfly' matrix of the form,

$$W = \begin{bmatrix} 1 & & & & \alpha \\ & 1 & & \circ & -\alpha^2 \\ & & \circ & & \alpha^3 \\ & \circ & & \circ & \\ & & \circ & & \\ \circ & & & & 1+(-\alpha)^{n-1} \end{bmatrix} \tag{5.4.1.4}$$

(5.4.1.3a) is then solved by the forward recurrence,

$$\left.\begin{aligned} &\text{a)} \quad v_1 = d_1 \\ &\text{b)} \quad v_i = d_i - \alpha v_{i-1} \quad i=2(1)n \end{aligned}\right\} \tag{5.4.1.5}$$

and (5.4.1.3b) by the form,

$$\left.\begin{aligned} &\text{a)} \quad \begin{bmatrix} 1 & \alpha \\ 0 & 1+(-\alpha)^{n-1} \end{bmatrix} \begin{bmatrix} u_1 \\ u_n \end{bmatrix} = \begin{bmatrix} v_1 \\ v_n \end{bmatrix} \\ &\text{b)} \quad \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} u_i \\ u_{n-i+1} \end{bmatrix} = \begin{bmatrix} v_i - (-1)^{i+1} \alpha^i u_n \\ v_{n-i+1} - (-1)^{n-i+2} \alpha^{n-i+1} u_n \end{bmatrix}, \quad i=2(1)\lfloor n/2 \rfloor \end{aligned}\right\} \tag{5.4.1.6}$$

(5.4.1.6) retains the desirable QI property of producing two results on each step, which are the inputs of the next cell. By a careful use of LIFO storage the cell can accept data in this format and sequentialise it for (5.4.1.5).

Fig.(5.4.1.1) illustrates the structure of the new O(n) cell, which consists of seven LIFO stores each of size $\frac{1}{2}\hat{n}$, combinational cells for (5.4.1.5), a 2×2 solver arrangement for (5.4.1.6), and a simple logic controller to orchestrate LIFO switching. Now assuming n is even we define two data streams of length $\hat{n}=\frac{1}{2}(n+2r)$,

$$\left.\begin{aligned} sd_1 &= \alpha_1\alpha_2 \cdots\alpha_r\ d_1\ d_2 \cdots d_{\frac{1}{2}n} \\ \text{and} \qquad & \\ \tilde{sd}_2 &= \gamma_1\gamma_2 \cdots\gamma_r\ d_n\ d_{n-1}\cdots d_{\frac{1}{2}n+1} \end{aligned}\right\} \qquad (5.4.1.7)$$

where α_i and γ_i i=1(1)r are the parameters of the circulant factorisation and associated addresses for loading. The solution to (5.4.1.1) is then found in three stages:

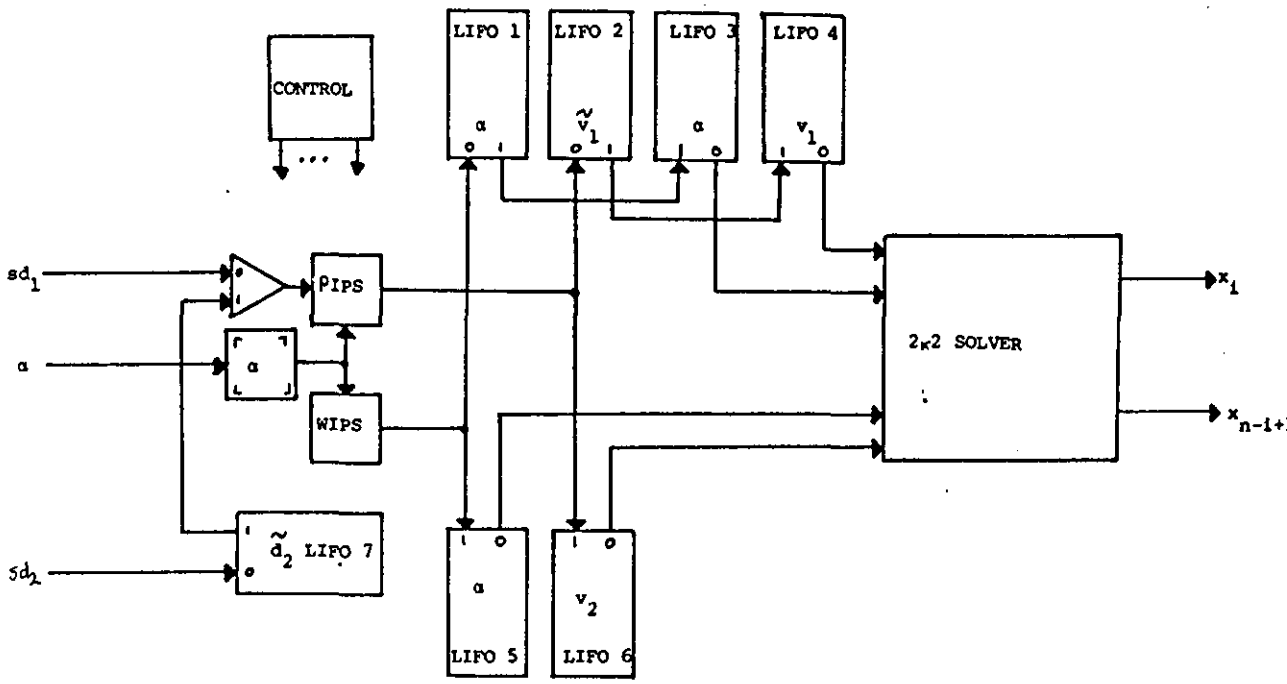
STAGE 1:

(i) The parameters of sd_1 pass through the forward solver without modification into LIFO 2, while LIFO 7 collects addresses from sd_2, and the correct cell parameter is selected.

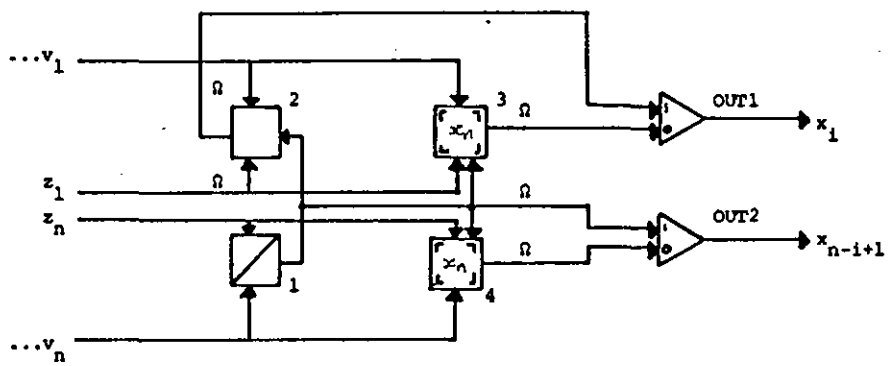
(ii) Computation starts when d_1 reaches the forward solver, with the already selected parameter loaded into the α cell, and the term (5.4.1.5a) produced. On subsequent cycles the PIPS cell generates terms in (5.4.1.5b) depositing them in LIFO 2, while the WIPS cell generates the last column of W inserting it into LIFO 1.

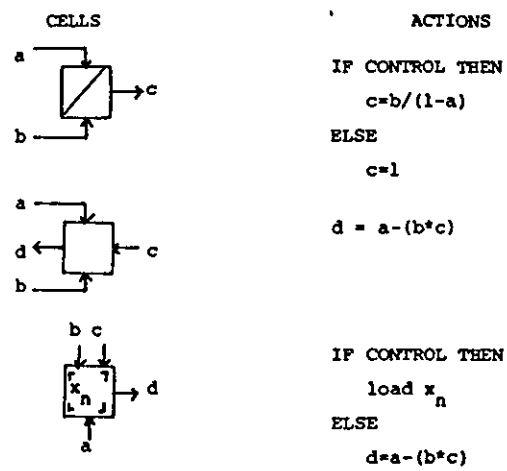
STAGE 2:

(i) Stage 2 begins when all the elements of sd_1 and sd_2 have been input. LIFO's 1,2,7 are switched to output and LIFO's 3,4,5



a) Global cell layout





b) 2×2 solver

FIGURE 5.4.1.1: Improved O(n) BATS cell

and 6 to input. The last ½n terms of (5.4.1.5) and W are computed by PIPS and WIPS and deposited into LIFO's 5 and 6 respectively.

(ii) The forward solver is switched off when the addresses begin to leave LIFO 7 and pass through PIPS into LIFO 6 unchanged.

STAGE 3:

(i) At the start of stage 3, LIFO's 3,4,5 and 6 are full, and switch to output. LIFO's 1,2 and 7 are empty and switch to input. The parameters and addresses then filter out through the 2×2 solver unchanged to the next cell.

(ii) 2×2 solver starts up computing u_1 and u_n in (5.4.1.6a) sequentially.

(iii) Finally the recurrence in (5.4.1.6b) is evaluated using the 2×2 solver.

The operation of the 2×2 solver requires some further explanation, and its structure is shown in Fig. (5.4.1.1b). Cells 1 and 2 compute u_n and u_1 while cells 3 and 4 evaluate u_i and u_{n-i+1} $i=2(1)\lfloor n/2 \rfloor$. Notice that the inputs of cell 2 are delayed by one cycle so that u_n is available when they arrive. Consequently the output line for u_n is also delayed to synchronise the final parallel output of u_1 and u_n. While u_1 is calculated, cells 3 and 4 load u_n and compute u_2 and u_{n-1}. Thus, outputs of cell 3 and cell 4 are delayed a single cycle to avoid a clash of u_1 and u_n with u_2 and u_{n-1} on output. Successive cycles then sees u_i and u_{n-i+1} generated by cells 3 and 4 with cells 1 and 2 redundant.

Now consider the signals required to control the new cell. These can be generated by the controller using triggers from a pair of tag

bits associated with the input data summarized below.

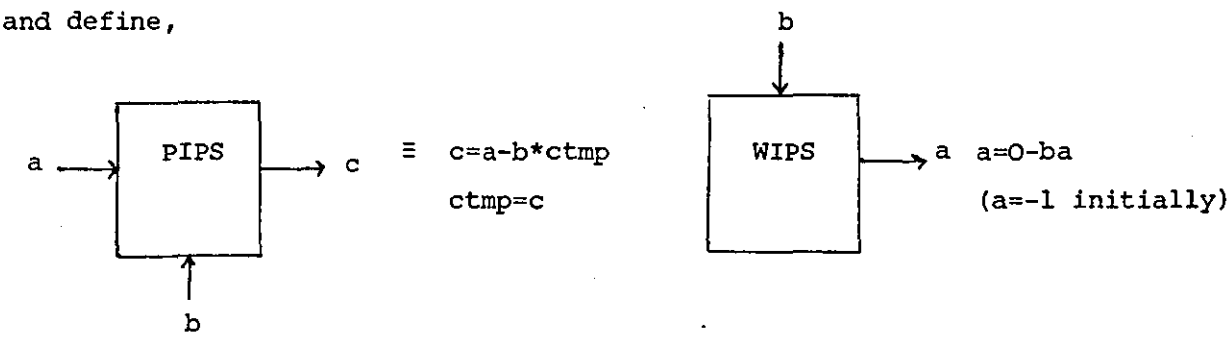
t1	t2	Action
0	0	Normal computation
0	1	Reset solvers
1	0	LIFO toggle
1	1	Disable solver to pass parameters and addresses

The sequence of tags for (5.4.1.7) is given by

$$\text{control} = \begin{cases} 1\ 1 \cdots 1 & 1\ 0\ 0 \cdots 0 & 0\ 1\ 0 \cdots 1 \\ 1\ 1 \cdots 1 & 1\ 1\ 0 \cdots 0 & 0\ 0\ 0 \cdots 0 \end{cases} \qquad (5.4.1.8)$$

$$\underbrace{\qquad\qquad}_{r} \quad \underbrace{\qquad\qquad}_{n/2} \quad \underbrace{\qquad\qquad}_{r+\frac{n}{2}}$$

and define,



with the α cell a simple register. To control the cell tag bits are interpreted differently at each stage in the computation by using the tags stored in the LIFOs. Denote the tags associated with LIFO i output as t(i) and the tags associated with cell input as t_1 and t_2. Also define $L_1=\{1,2,7\}$ and $L_2=\{3,4,5,6\}$ and notice that any two LIFOs $i \in L_1$ and $j \in L_2$ are mutually exclusive because if LIFO i is inputting LIFO j must be outputting and vice versa. Thus,

$$c_1 = t_1 \wedge \bar{t}_2 = \begin{cases} 0 & L_1 \text{ input} \\ 1 & L_2 \text{ input} \end{cases} \qquad (5.4.1.9)$$

is sufficient to control all LIFOs and,

$$c_2 = (t_1 \wedge t_2) \vee (t(2) \wedge t(7)) \qquad (5.4.1.10)$$

disables the forward cell to pass parameters and addresses (this is achieved by zeroing the α register), while,

$$c_3 = t(4) \wedge t(6) \ , \qquad (5.4.1.11)$$

is used to disable the 2×2 solver, this is a simple task as the output of LIFOs 3 and 5 associated with parameters and address are zeroes due to disabling of the forward solver. So using cells 3 and 4 of the 2×2 solver is sufficient. The 2×2 solver itself can be started with the code,

$$c_4 = \overline{t(4)} \wedge t(6) \qquad (5.4.1.12)$$

and it follows that only simple combination logic is necessary for cell control.

Next consider the case when n is odd sd1 and sd2 in (5.4.1.7) are of different lengths. Nominating sd2 as the shortest sequence unit delays must be added to the input of LIFO 7 and the outputs of LIFOs 1 and 2. These delays allow a smooth switch over from sd1 input to the input from LIFO 7 to the forward solver and re-align data for input to the 2×2 solver. A third tag bit t_3 must be added to the input controls to mark data streams when n is odd, and to control switching in and out of these additional delays.

Finally by tracing the longest path through the cell we derive the following result,

Theorem 5.4.1: The solution of the system $A_c x=b$ where A_c is an n×n matrix of semi-bandwidth r and x and b are n×1 vectors can be found by the Audish & Evans factorisation using 2r improved O(n) BATS cells in,

$$T = \tfrac{1}{2}\hat{n}+2r(\hat{n}+5) \text{ ips cycles where } \hat{n}=n+2r.$$

Proof:

The timing T is given by $T=\ell+2rc$ where ℓ=total input length and c is the cell latency. From (5.4.1.7) $\ell=\frac{1}{2}n+r$, and by summing delays through the cell $c=(n+2r)+5$, where we allow two delays for the forward solver and three delays in the 2×2 solver, and assume n is odd. As there are 2r cells, output is delayed by 2rc cycles in total.

This improved timing results over that of Theorem(4.4.2.1) from the increased throughput of the array, analysing LIFO operations shows that the next problem instance can be input after $\hat{n}$ cycles, rather than the 2n required previously. The new design requires 6 ips equivalents and at most $\frac{7}{2}\hat{n}$ FIFO register, compared with 4 ips and 2n L/F registers in the old O(n) cell. Thus speed increase and throughput are offset by an increase of 1.5-2 times the hardware, which for large n is significant. Notice however, that the new cell is capable of solving any problem of size $\bar{n}<n$ without modification giving added flexibility.

5.4.2 A Stable p-cyclic Cell

The p-cyclic cell was developed in Section (4.4.3) and solved the alternative n×n system,

$$A^t u = d \ , \qquad (5.4.2.1)$$

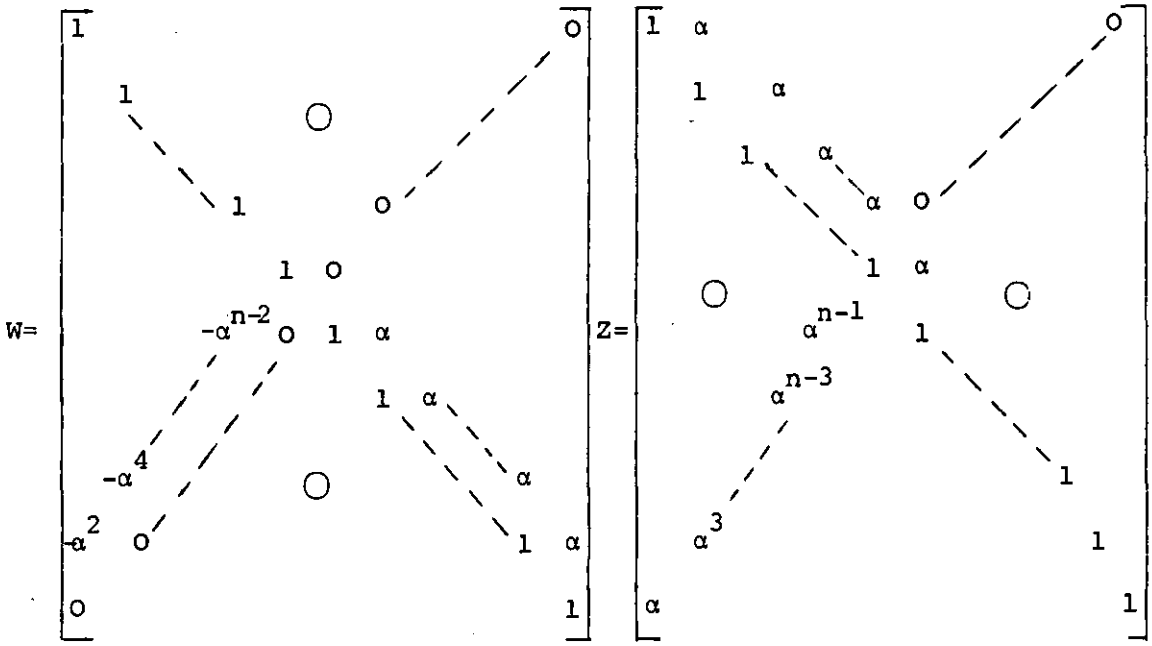
using the p-cyclic properties of A^t and the fact that $\alpha^{s+1}=0$ for $s<n$ and $0<|\alpha|<1$. The resulting procedure consisted of evaluating a polynomial of s terms to generate u_1, a forward recursive scheme to calculate u_i $i=1(1)\frac{1}{2}n$ and a backward recursion for u_{n-i+1} $i=1(1)\frac{1}{2}n$. This latter recursive scheme was unstable against rounding error, involving division by α.

In the new cell we factorise A^t using the QIF method. The special

form of A^t allows factors to be constructed by inspection and generated 'on-the-fly' by the cell, consequently,

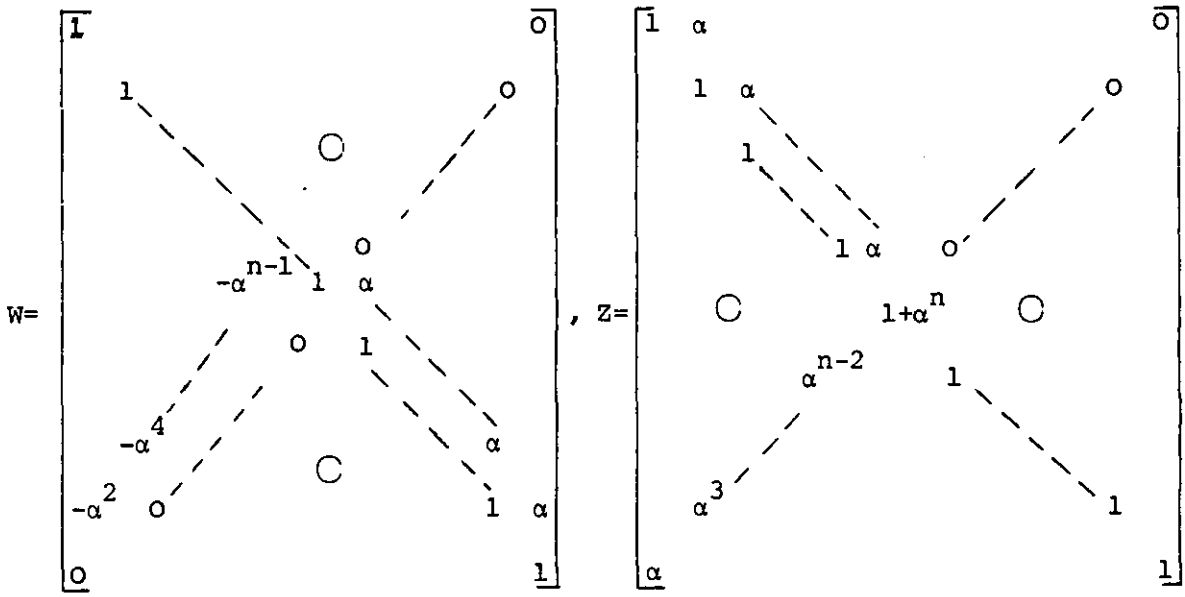
$$A^t = WZ \ , \qquad (5.4.2.2)$$

where for n=even,



(5.4.2.3)

and n=odd,



(5.4.2.4)

which after substitution in (5.4.2.1) yields the coupled systems,

$$\text{a)}\quad Wv = d \text{ and } \quad \text{b)}\quad Zu = v\ , \tag{5.4.2.5}$$

Applying the QI permutation produces the simple 2×2 block LU form which for n=8 has the form,

$$L = \begin{array}{c|cc|cc|cc|cc}
 & 1 & 8 & 2 & 7 & 3 & 6 & 4 & 5 \\ \hline
1 & 1 & 0 & & & & & & \\
8 & 0 & 1 & & & & & & \\ \hline
2 & 0 & 0 & 1 & 0 & & & & \\
7 & -\alpha^2 & \alpha & 0 & 1 & & & & \\ \hline
3 & & & 0 & 0 & 1 & 0 & & \\
6 & & & -\alpha^4 & \alpha & 0 & 1 & & \\ \hline
4 & & & & & 0 & 0 & 1 & 0 \\
5 & & & & & -\alpha^6 & \alpha & 0 & 1
\end{array}\ ,\ U = \begin{array}{c|cc|cc|cc|cc}
 & 1 & 8 & 2 & 7 & 3 & 6 & 4 & 5 \\ \hline
1 & 1 & 0 & \alpha & 0 & & & & \\
8 & \alpha & 1 & 0 & 0 & & & & \\ \hline
2 & & & 1 & 0 & \alpha & 0 & & \\
7 & & & \alpha^3 & 1 & 0 & 0 & & \\ \hline
3 & & & & & 1 & 0 & \alpha & 0 \\
6 & & & & & \alpha^5 & 1 & 0 & 0 \\ \hline
4 & & & & & & & 1 & \alpha \\
5 & & & & & & & \alpha^7 & 1
\end{array}$$

(5.4.2.6a)

and for n=7,

$$L = \begin{array}{c|cc|cc|cc|c}
 & 1 & 7 & 2 & 6 & 3 & 5 & 4 \\ \hline
1 & 1 & 0 & & & & & \\
7 & 0 & 1 & & & & & \\ \hline
2 & 0 & 0 & 1 & 0 & & & \\
6 & -\alpha^2 & \alpha & 0 & 1 & & & \\ \hline
3 & & & 0 & 0 & 1 & 0 & \\
5 & & & -\alpha^4 & \alpha & 0 & 1 & \\ \hline
4 & & & & & -\alpha^6 & \alpha & 1
\end{array}\ ,\ U = \begin{array}{c|cc|cc|cc|c}
 & 1 & 7 & 2 & 6 & 3 & 5 & 4 \\ \hline
1 & 1 & 0 & \alpha & 0 & & & \\
7 & \alpha & 1 & 0 & 0 & & & \\ \hline
2 & & & 1 & 0 & \alpha & 0 & \\
6 & & & \alpha^3 & 1 & 0 & 0 & \\ \hline
3 & & & & & 1 & 0 & \alpha \\
5 & & & & & \alpha^5 & 1 & 0 \\ \hline
4 & & & & & & & 1+\alpha^7
\end{array}$$

(5.4.2.6b)

now for n large enough $\alpha^n=0$ thus $\alpha^7=0$ and in general the last diagonal block of U has the form,

$$\begin{bmatrix} 1 & \alpha \\ 0 & 1 \end{bmatrix}\ ,\ \text{n=even, 1 n=odd.}$$

Now as $0<|\alpha|<1$ and Z is diagonally dominant, it follows that the block co-diagonal of U can be eliminated using implicit block calculations without pivoting and with the modifications restricted to the permuted rhs of (5.4.2.5b). This leads to the elimination recurrences,

$$\left.\begin{aligned} e_q &= v_q - \alpha v_{q-1} \\ e_{q-1} &= v_{q-1} + \alpha e_q \end{aligned}\right\} \qquad (5.4.2.7)$$

and generally,

$$e_{q-i} = v_{q-i} - \alpha e_{q-i+1} \quad , \quad i=0(1)q-1$$

which produce the modified vector e in polynomial form,

$$\left.\begin{aligned} e_1 &= v_1 - \alpha v_2 + \alpha^2 v_3 - \alpha^3 v_4 + \ldots + (-\alpha)^q v_{q+1} \\ e_2 &= v_2 - \alpha v_3 + \alpha^2 v_4 - \alpha^3 v_5 + \ldots + (-\alpha)^{q-1} v_{q+1} \\ &\vdots \\ e_i &= v_i - \alpha v_{i+1} + \alpha^2 v_{i+2} + \ldots + (-\alpha)^{q-i+1} v_{q+1} \\ &\vdots \\ e_q &= v_q - \alpha v_{q+1} \end{aligned}\right\} \qquad (5.4.2.8)$$

and

$$e_{q+j} = v_{q+j}, \quad j=1(1)q \text{ where } q=\lfloor n/2 \rfloor .$$

After elimination the permuted form of (5.4.2.5b) is a 2×2 block diagonal matrix. Hence the solution of (5.4.2.1) can be found using a stable three stage process:

STAGE 1: SOLVE (5.4.2.5a)

STAGE 2: CONVERT (5.4.2.5b) into the system $\tilde{Z}u=e$ by elimination where $\tilde{Z}$ is a matrix with a W type format.

STAGE 3: SOLVE $\tilde{Z}u=e$ for u.

It follows, from the fact that W and $\tilde{Z}$ fit the same global format that the three stages can be pipelined to achieve high throughput yielding the new BATS cell form,

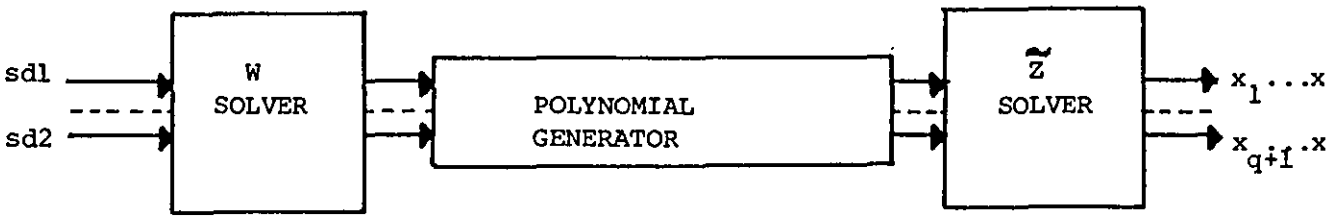
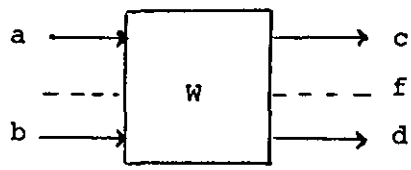


FIGURE 5.4.2.1: Stable p-cyclic cell

where W and $\tilde{Z}$ have the following definitions:



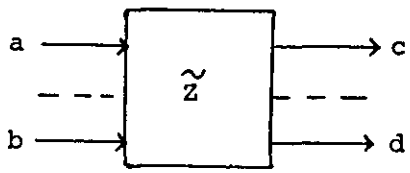
on start up $\beta=1$, cold=0, dold=0

cold=c, dold=d

c=a, d=b+(β*cold)-(α*dold)

$\beta=\beta*(\alpha^2)$

f=α



on start up $\beta=\alpha$

c=a, d=b-(β*a)

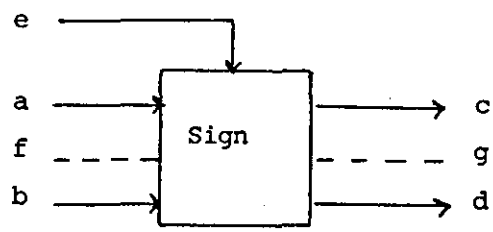
$\beta=\beta*(\alpha^2)$

The polynomial generator is a linear array of q cells which produces the polynomials of (5.4.2.8) according to the general recurrence,

$$\left.\begin{aligned} e_i^{(0)} &= v_i \\ e_i^{(k+1)} &= e_i^{(k)} + (-\alpha)^k v_{i+k} \quad , \; k=1(1)m \\ e_i &= e_i^{(q-i+2)} \end{aligned}\right\} \qquad (5.4.2.9)$$

where m=q-i+1 and i=1(1)q,

and uses cells of the form,



c=a$\pm$(e*f) (sign dictates + or -)

g=αf

d=b

The basic idea is to pump the $e_i^{(0)}$ values from left to right through the array as they emerge from the W solver. From (5.4.2.9) it follows that at the end of the qth cycle of array operation the (q-i+1)th generator cell contains the completed e_i polynomial. Thus, on the (q+1)th cycle the generator must be disabled to preserve the results and cells reduced to simple FIFO type operations, pipelining data into the $\tilde{Z}$ solver. Snap-shots of generator operation are given in Fig. (5.4.2.2), notice that the values v_{q+j}, j=1(1)q are also queued by the cells to maintain synchronisation.

Control of the new BATS cell and in particular the generator sub-array is achieved by adopting the tag bit method used in the O(n) cell improvement. First we revise the t_1 and t_2 tag bit definitions as follows:

t_1	t_2	Action
0	0	Normal computation
0	1	Data switch
1	0	Start up the cell
1	1	Disable cells

and define the tag bits entering different cells of Fig.(5.4.2.1), as $t_1(i)$ and $t_2(i)$ for i=0(1)q+1, such that the input tags of cells W and $\tilde{Z}$ are $t_1(0)$, $t_2(0)$ and $t_1(q+1)$,$t_2(q+1)$ respectively. Also let polynomial generator cell i receive tag bits $t_1(i)$, $t_2(i)$, i=1(1)q and $t_1(b)$,$t_2(b)$ which define the tag bits associated with the broadcast line. The generator array is then easily switched from polynomial generator to FIFO mode and vice versa using two mutually exclusive states defined by,

$$\left.\begin{aligned} OFF_i &= (t_1(b) \wedge t_2(b)) \vee \overline{ON_i} \\ ON_i &= (t_1(i) \wedge \overline{t_2(i)}) \vee \overline{OFF_i} \end{aligned}\right\} \qquad (5.4.2.10)$$

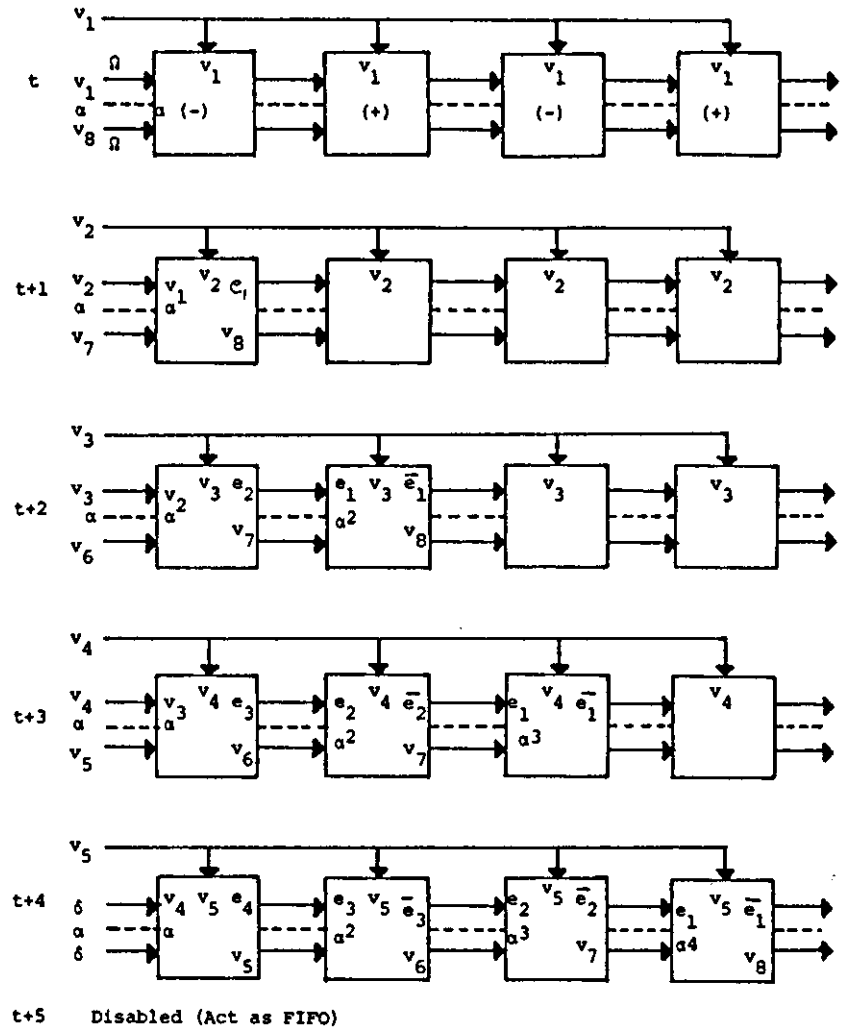


FIGURE 5.4.2.2: Snapshots of polynomial generator (for n=8)

for cell i, with OFF and ON indicating FIFO and polynomial modes respectively. Clearly the requirement that all cells should be switched OFF simultaneously on completion of the polynomials is satisfied by the use of broadcast tags. Notice however that cells can only be switched ON again in a strict left to right order, which preserves data already in the array, but also achieves high throughput by allowing different problem instances to be pipelined. Thus preserving the attributes of the original but unstable BATS cell.

A further area of concern is the passing of parameters and addresses (for setup) through the array unharmed, and is solved by adopting the input format,

$$sd_1 = \left\{\begin{array}{c|c} \alpha_1\ \alpha_2 \ldots \alpha_r\ d_1\ d_2\ \ldots d_q & \overline{\alpha_1}\ \overline{\alpha_2} \ldots \text{data} \\ 1\ 1 \ldots 1\ 1\ 0\ \ldots 0 & 1\ 1 \ldots t_1 \end{array}\right.$$

1st problem 2nd problem

and,

$$sd_2 = \left\{\begin{array}{c|c} \gamma_1\ \gamma_2 \ldots \gamma_r\ d_{2q} \ldots d_{q+1} & \overline{\gamma_1}\ \overline{\gamma_2} \ldots \quad \text{data} \\ 1\ 1 \ldots 1\ 0\ 0 \ldots 1 & 1\ 1 \ldots \quad t_2 \end{array}\right.$$

where q=n/2 from above. Tracing through the array operation shows that this sequence also controls the polynomial cells using (5.4.2.10) and that the w and $\tilde{z}$ cells are easily disabled by setting and holding cold=dold=0 and β=0 respectively.

Next consider the data switch command $(\overline{t_1} \wedge t_2)$. This is a special control which provides a neat solution to the construction of e_q. From (5.4.2.8) e_q is the smallest polynomial and enters the generator last and accumulates only one term using v_q and v_{q+1} occurring on different input lines whereas the values used in evaluating all other e_i polynomials are derived from a single input line. Consequently to preserve computation the broadcast value is selected from the correct line using a multiplexor controlled by the data switch tag code. A further complication arises when n is odd and the WZ form of (5.4.2.4) must be used. Here $q=\lfloor n/2 \rfloor$ producing data sequences of different lengths which must be padded as follows:

$$sd_1 = \left\{\begin{array}{c|c} \alpha_1\ \alpha_2 \ldots \alpha_r\ d_1\ d_2\ \ldots d_q\ \delta & \overline{\alpha_1} \ldots \\ 1\ 1 \ldots 1\ 1\ 0 \ldots 0\ 0 & 1 \ldots \end{array}\right.$$

and

$$sd_2 = \left\{ \begin{array}{cccccccc|c|c} \gamma_1 & \gamma_2 & \cdots & \gamma_r & d_n & d_{n-1} & \cdots & d_{q+2} & d_{q+1} & \bar{\gamma}_1 \cdots \\ 1 & 1 & \ldots & 1 & 0 & 0 & \ldots & 0 & 1 & 1 \ldots \end{array} \right. .$$

From (5.4.2.6) it follows that the W cell computes correctly with no further control, as does the $\tilde{Z}$ cell under the assumptions that $\alpha^n=0$ and the dummy element $\delta=0$. The only significant problem is the correct synchronisation of elements for the calculation of e_q; this is solved by further use of the data switch and an additional tag bit t_3 to differentiate between problems with odd and even n. Thus the interface between the W cell and polynomial generator which produces the broadcast signal has the form,

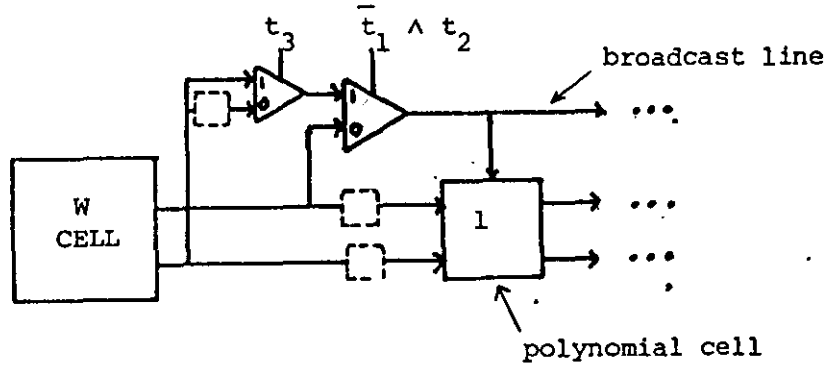


FIGURE 5.4.2.3: W-cell/generator interface

Finally, when $\alpha^{s+1}=0$ where $s<q$ the value m of (5.4.2.9) can be redefined as,

$$m = \min(q-i+1, s) \ , \qquad (5.4.2.11)$$

from which it follows that each e_i, i=1(1)q polynomial contains at most s terms. Consequently the polynomial generating array can be truncated to s cells reducing area and cell latency. We can now prove the following timing theorem for the solution of the circulant problem (4.4.1.8) using the improved cell.

Theorem 5.4.2.1: The solution of the linear system $A_c x=b$ where A_c is a $n \times n$ circulant matrix of semi-bandwidth r, and x and b are $n \times 1$ component vectors, can be found using the Audish & Evans factorisation and 2r stable p-cyclic BATS cells in at most $T=n+r(4s+14)+2$ ips cycles, where $\alpha^{s+1}=0$ for $\alpha=\max_{1 \leq i \leq r} (\alpha_i)$.

Proof:

The timing of the BATS pipeline using the new p-cyclic cell is given by,

$$T = k(\hat{n}+2rc) \quad , \tag{5.4.2.12}$$

where $\hat{n}=\lceil n/2 \rceil + r \leq \frac{1}{2}(n+2)+r$ and is the maximum data input (hence output) length, c is the new cell latency and k is the number of ips cycles equivalent to the maximum cycle of the cells in Fig.(5.4.2.1). Observation of the above cell definitions shows that the W-cell is the most computationally complex cell, and a lower and upper bound for T can be derived according to whether the value α^2 is known or not.

Case (i): Lower bound

When α^2 is known the W-cell computation can be pipelined, using the two stage calculation:

$$\begin{aligned} t: &\quad d_o = b+\beta c_o, c_o = a, \beta = \beta * \alpha^2 \quad , \\ t+1: &\quad d_{old} = d, \quad d = d_o - \alpha d_{old}, \quad c = c_o \quad . \end{aligned}$$

Thus by introducing an extra cycle to the cell latency a single ips cycle is the cycle time of the internal p-cyclic cell hence k=1. The latency is c=s+4, allowing 2 cycles for W-cell, s+1 for polynomial array (and interface) and 1 for Z-cell. Hence, we have,

$$T = \tfrac{1}{2}n+r(2s+9) + 1 \quad . \tag{5.4.2.13}$$

Case (ii): Upper bound

When α^2 is unknown the term $\beta=\beta*\alpha^2$, must be calculated by

two sequential operations,

$$\beta = \beta*\alpha, \quad \beta = \beta*\alpha;$$

the pipelining of W-cell computations is no longer possible and k=2 results. The latency is determined as c=s+3 hence,

$$T = n+r(4s+14)+2 \qquad (5.4.2.14)$$

giving the theorem time.

The area estimate of the design in ips equivalents is determined as follows. Assign 3 (or 4) 2 (or 3) and 2 ips equivalents for the $W,\tilde{Z}$ and polynomial cells respectively depending on whether α^2 is known or not. The area is then at most (2s+7) ips cells per BATS cell and 2r(2s+7) for the full pipe. By comparison with the original p-cyclic cell and the intuitive assumption that α^2 is unknown (i.e. simpler set up procedure), we conclude that the former scheme was faster and more area efficient. However the new design would be more desirable in practice due to its inherent stability which outweighs the lower speed and area estimates of the unstable cell.

5.5 SYSTOLIC QUADRANT INTERLOCKING ELIMINATIONS (SQIE)

The close relationship between Gaussian Elimination and the factorisation of matrices embodied in Theorems (2.3.1)-(2.3.2), implies that if a QI factorisation exists intuitively there should also be a corresponding QI form of Gaussian elimination. The essential idea of a QI Elimination (QIE) is to replace (5.1.1) by the more easily solved system,

$$Zx = \bar{b}\ , \qquad (5.5.1)$$

where, $Z^T = [z_1, z_2, \ldots, z_n]$

and

$$Z_i = \begin{cases} [\underbrace{0 \ldots 0}_{i-1} \; z_{ii} \; \cdots \; z_{i,n-i+1} \; 0 \; \cdots \; 0]^T & i=1(1)\left\lceil \frac{n+1}{2} \right\rceil \\ [\underbrace{0 \ldots 0}_{n-i} \; z_{i,n-i+1} \; \cdots \; z_{ii} \; 0 \; \cdots \; 0]^T , & i=\left\lceil \frac{n+1}{2} \right\rceil +1(1)n \end{cases} \tag{5.5.2}$$

and permuted form of the $n \times 1$ vector b in (5.1.1) is b.

Now by employing the QI permutation the permuted matriz $\bar{Z}$ of Z has a 2×2 block upper triangular form. Consequently by extending (2.3.2.1) to the 2×2 block case, $\bar{Z}$ is constructed from a sequence of modified matrices $A^{(1)}, A^{(2)}, \ldots, A^{(k)}$ for $k=1(1)\lceil n/2 \rceil$ as follows:-

$$\left.\begin{array}{l} \text{when } k=1 \qquad A^{(1)} = \bar{A} \text{ (the QI permuted form of A)} \\ k>1 \quad A_{ij}^{(k)} = \begin{cases} A_{ij}^{(k-1)} & , \; i=1(1)k-1, \; j=1(1)\lceil n/2 \rceil \\ 0 & , \; i=k(1)\lceil n/2 \rceil, \; j=1(1)k-1 \\ A_{ij}^{(k-1)} - M_{i,k-1} A_{k-1,j}^{(k-1)} & , \; i=k(1)\lceil n/2 \rceil, \; j=k(1)\lceil n/2 \rceil \end{cases} \end{array}\right\} \tag{5.5.3}$$

where,

$$A_{ij}^{(k)} = \begin{bmatrix} a_{ij}^{(k)} & a_{i,n-j+1}^{(k)} \\ a_{n-i+1,j}^{(k)} & a_{n-i+1,n-j+1}^{(k)} \end{bmatrix} \tag{5.5.4a}$$

are 2×2 submatrices and the $M_{i,k-1}$ are multiplier matrices with the form,

$$M_{i,k-1} = A_{i,k-1}^{(k-1)} \left[A_{k-1,k-1}^{(k-1)}\right]^{-1} , \tag{5.5.4b}$$

finally, $\qquad Z = A^{(k)}$, when $k=\lceil n/2 \rceil$.

The corresponding rhs modification sequence follows trivially as,

$$\left.\begin{array}{l} b^{(1)} = \bar{b} \text{ (QI permuted form of b)} \\ \hat{b}_i^{(k)} = \hat{b}_i^{(k-1)} - M_{i,k-1} \hat{b}_{k-1} \quad i=k(1)\lceil n/2 \rceil \end{array}\right\} \tag{5.5.4c}$$

where,

$$\hat{b}_j = [b_j, b_{n-j+1}]^T .$$

Tracing the elimination sequence shows that the $M_{i,k-1}$ matrices form a 2×2 block lower triangular matrix which produces a W format matrix M and (5.5.1) becomes,

$$Zx = Mb \ . \tag{5.5.5}$$

The systolic QIE (SQIE) therefore reduces to a 2×2 explicit block structured Gaussian elimination array. The new array is easily derived using the global structure in Fig.(3.2.2.2), by re-defining the cells of (3.2.2.4) and (3.2.2.5) and reformating data inputs, as follows: first assume that no pivoting is necessary, (i.e. A is diagonally dominant or positive definite) in order to simplify block 2×2 cell definitions. Next from (5.5.4b) with k=2

$$M_{i,1} = A_{i1}^{(1)} [A_{11}^{(1)}]^{-1} \tag{5.5.6}$$

and for i=2 (the first modification) it follows that,

$$A_{21}^{(1)} - M_{21} A_{11}^{(1)} = 0 \ . \tag{5.5.7}$$

The first 2×2 block row update in (5.5.3) has the form,

$$A_{2j}^{(2)} = A_{2j}^{(1)} - M_{21} A_{1j}^{(1)} \ , \quad j=2(1)\lceil n/2 \rceil \tag{5.5.8}$$

and can be computed in pipelined fashion as follows. For purposes of illustration let j=2 then (5.5.8) becomes,

$$A_{22}^{(2)} = A_{22}^{(1)} - M_{21} A_{12}^{(1)} \ . \tag{5.5.9}$$

Now letting,

$$M_{21} = \frac{1}{D} . A_{21}^{(1)} * X \ , \tag{5.5.10}$$

where $X = D.A_{11}^{-1}$,

and substituting for (5.5.10) in (5.5.9) yields,

$$A_{22}^{(2)} = A_{22}^{(1)} - (\frac{1}{D}.A_{21}^{(1)})(XA_{12}^{(1)}) \ . \tag{5.5.11}$$

Notice that X is easily constructed by simply swapping diagonal elements and negating anti-diagonal elements. Likewise $\frac{1}{D}.A_{21}^{(1)}$ is simple to compute requiring the formation of $D=a_{11}a_{nn}-a_{1n}a_{n1}$ and

simple divisions. With these values known (5.5.11) is constructed by assigning $E=\frac{1}{D}A_{21}^{(1)}$ and

$$\left.\begin{aligned} Y &= XA_{12}^{(1)} \\ A_{22}^{(2)} &= A_{22}^{(1)}-(E*Y) \end{aligned}\right\} \qquad (5.5.12)$$

producing simple block ips calculations of the form,

$$\begin{bmatrix} y_1 & y_2 \\ y_3 & y_4 \end{bmatrix}^{(\frac{1}{2})} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}^{(0)} + \begin{bmatrix} x_1 & 0 \\ x_3 & 0 \end{bmatrix} \begin{bmatrix} a_{12} & a_{17} \\ a_{82} & a_{87} \end{bmatrix}^{(1)} \qquad (5.5.12a)$$

$$\begin{bmatrix} y_1 & y_2 \\ y_3 & y_4 \end{bmatrix}^{(1)} = \begin{bmatrix} y_1 & y_2 \\ y_3 & y_4 \end{bmatrix}^{(\frac{1}{2})} + \begin{bmatrix} 0 & x_2 \\ 0 & x_4 \end{bmatrix} \begin{bmatrix} a_{12} & a_{17} \\ a_{82} & a_{87} \end{bmatrix}^{(1)} \qquad (5.5.12b)$$

$$\begin{bmatrix} a_{22} & a_{27} \\ a_{72} & a_{77} \end{bmatrix}^{(1+\frac{1}{2})} = \begin{bmatrix} a_{22} & a_{27} \\ a_{72} & a_{77} \end{bmatrix}^{(1)} - \begin{bmatrix} e_1 & 0 \\ e_3 & 0 \end{bmatrix} \begin{bmatrix} y_1 & y_2 \\ y_3 & y_4 \end{bmatrix}^{(1)} \qquad (5.5.12c)$$

$$\begin{bmatrix} a_{22} & a_{27} \\ a_{72} & a_{77} \end{bmatrix}^{(2)} = \begin{bmatrix} a_{22} & a_{27} \\ a_{72} & a_{77} \end{bmatrix}^{(1+\frac{1}{2})} - \begin{bmatrix} 0 & e_2 \\ 0 & e_4 \end{bmatrix} \begin{bmatrix} y_1 & y_2 \\ y_3 & y_4 \end{bmatrix}^{(1)} \qquad (5.5.12d)$$

which yields the pipeline procedure,

t: compute X

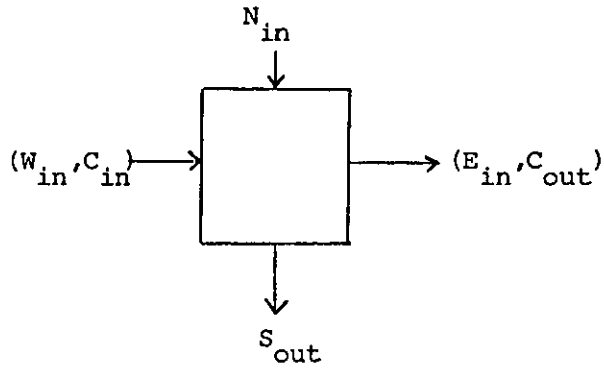
t+1: evaluate D, and $Y^{(\frac{1}{2})}$ using (5.5.12a)

t+2: evaluate 1st column of E and $Y^{(1)}$ using (5.5.12b)

t+3: evaluate 2nd column of E and $A_{22}^{(1+\frac{1}{2})}$ using (5.5.12c)

t+4: evaluate $A_{22}^{(2)}$ using (5.5.12d)

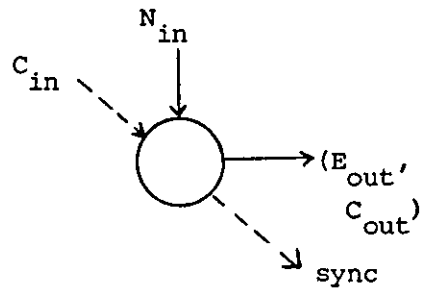
From which the new cell definitions follow immediately.



IF c_{in} THEN
{PHASE1:STARTUP

t:N_{in} r_1,r_2,r_3,r_4
W_{in} x_1,x_2,x_3,x_4
$y_1=0+x_1r_1$, $y_2=0+x_1r_2$
$y_3=0+x_3r_1$, $y_4=0+x_4r_2$

```
                t+1:  E_out x1,x2,x3,x4
                      y1=y1+x2r3, y2=y2+x2r4
                      y3=y3+x4r3, y4=y4+x4r4
        }
    ELSE
        {PHASE2: COMPUTATION
                t:  N_in r1,r2,r3,r4
                    W_in e1,e3
                    E_out e2,e4
                    S_out r̄1,r̄2,r̄3,r̄4
                    r̄1=r1-e1y1, r̄2=r2-e1y2
                    r̄3=r3-e3y1, r̄4=r4-e3y2
                t+1: W_in e2,e4
                    E_out e1,e3
                    r̄1=r̄1-e2y3, r̄2=r̄2-e2y4
                    r̄3=r̄3-e4y3, r̄4=r̄4-e4y4
        }
```




```
    IF C_in THEN
        {PHASE1: SETUP
                t:  N_in r1,r2,r3,r4
                    x2=0-r2, x3=0-r3, x1=r4,x4=r1
                t+1: E_out x1,x2,x3,x4
                    D=r1r4-r2r3
        }
    ELSE
        {PHASE2: COMPUTATION
                t:  N_in e1,e2,e3,e4
                    E_out ē2,ē4
                    ē1=e1/D, ē3=e3/D
                t+1: E_out ē1,ē3
                    ē2=e2/D, ē4=e4/D
        }
```

Notice that each cycle of a phase is a single ips cycle, and that the control value C_{in} moves independently of computation. For instance,

the horizontal signal (C_{out}/C_{in}) moves one cell every cycle starting up cells left to right, while the diagonal signal sync initiates row computation and moves from boundary cell to boundary cell every 5 ips cycles. This is clarified by tracing array operations shown in Fig.(5.5.1) which also indicates the data input format. Each input

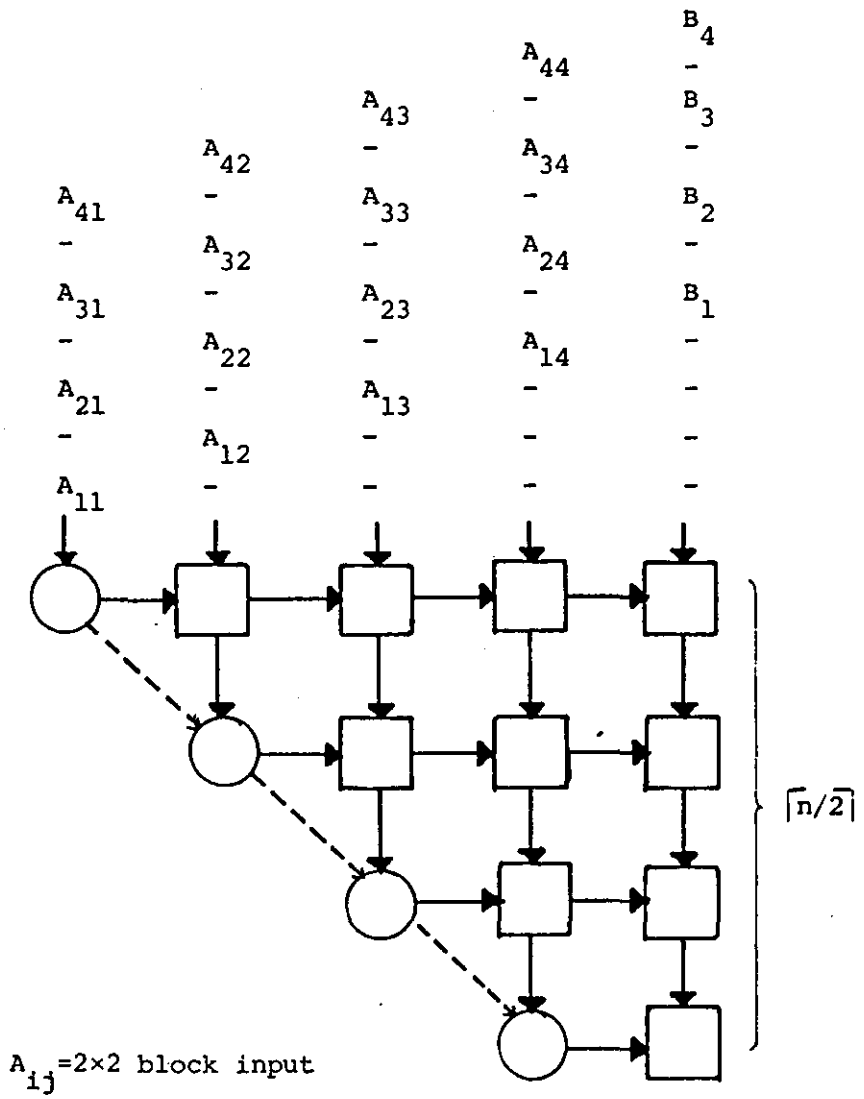


FIGURE 5.5.1: 2×2 Block eliminator

is a 2×2 block of four inputs, and the block B_i have the special form,

$$B_i = \begin{bmatrix} b_i & 0 \\ b_{n-i+1} & 0 \end{bmatrix}$$

derived from (5.5.6) which preserves computation at the expense of extra hardware in the last column of cells. The corresponding result to Theorem (3.2.2.2) now follows immediately.

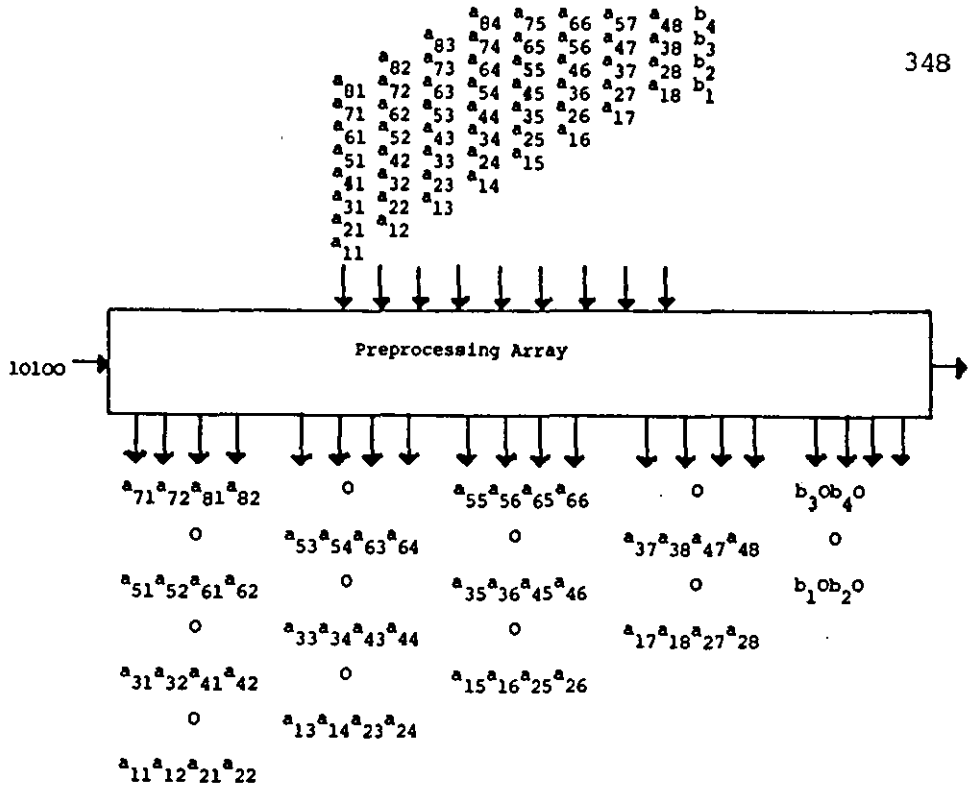
Theorem (5.5.1): The 2×2 block Gaussian elimination (or QIE) of a matrix A can be found using a block structured triangular array in $T=5\lceil n/2\rceil$ ips cycles and uses $O(n^2)$ point ips cells.

Proof:

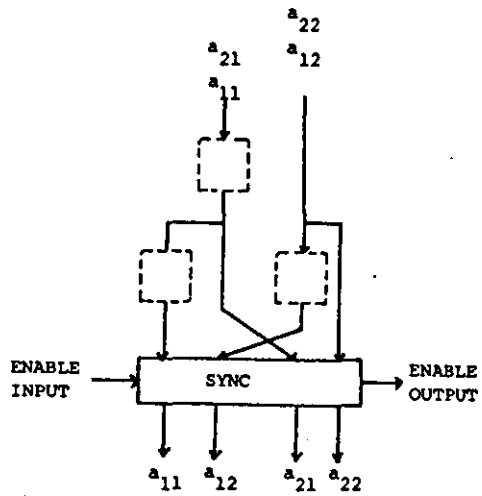
The array contains $\lceil n/2\rceil$ rows of block ips cells with each cell equivalent to 4 point ips cells. The synchronisation signal starts a new row computing every 5 point ips cycles and when it leaves the array all modifications must be complete. It follows that $T=5\lceil n/2\rceil$ is the total computation time.

The array requires $\sum_{i=1}^{\lceil n/2\rceil} i=\frac{1}{2}\lceil n/2\rceil(\lceil n/2\rceil+1)$ block ips cells, and $2\lceil n/2\rceil(\lceil n/2\rceil+1)$ point ips cell equivalents, compared with $\frac{1}{2}n(n+1)$ in the ordinary point elimination case. We conclude that the block elimination method is faster but requires more hardware than the point case. Efficiency is also improved in the block case but this results from effectively starting up two adjacent rows of point cells for each block row at a time in the new design when compared with a single row in the old method.

Finally, Fig.(5.5.2) indicates that the input format for the new array can be derived from the input of the original point version of the array. The idea is to employ a reformatting preprocessor consisting of a linear array of $\lceil n/2\rceil+1$ reformating cells. Each cell of the processor consists of delay registers and whose output is controlled by a simple bit signal travelling left to right. The preprocessor also provides a suitable method for expanding a small host interface to a large enough bandwidth for array input.



a) Input/output format of the preprocessing array (n=8)



b) Basic preprocessing cell

FIGURE 5.5.2: Reformatting of data for 2×2 block eliminator

5.6 SYSTOLIC QUADRANT INTERLOCKING ITERATION (SQII)

To complete this chapter we now briefly consider Quadrant Interlocking Iterative (QII) schemes as introduced by Evans & Sojoodi-Haghighi [82]. The idea here is to take the linear system,

$$Ax = b \ , \tag{5.6.1}$$

and apply a QI splitting of the form,

$$A = X-W-Z \ , \tag{5.6.2}$$

where,

$$X = [x_1, x_2, \ldots, x_n]$$

$$-W = [w_1, w_2, \ldots, w_n] \text{ and } -z^T = [z_1, z_2, \ldots, z_n] \tag{5.6.3}$$

such that,

$$X_i = \begin{cases} [\underbrace{0 \ldots 0}_{i-1}\ a_{ii}\ 0, \ldots 0,\ a_{n-i+1,i}\ 0, \ldots, 0]^T & i = 1(1)\left\lceil \frac{n+1}{2} \right\rceil \\ [0, \ldots, 0\ a_{n-i+1,i}\ 0, \ldots, 0\ a_{ii}\ 0, \ldots, 0]^T & i = \left\lceil \frac{n+3}{2} \right\rceil (1) n \end{cases} \tag{5.6.4}$$

and for n odd

$$W_i = \begin{cases} [\underbrace{0, \ldots, 0}_{i}\ a_{i+1,i}, \ldots, a_{n-i,i}\ 0 \ldots 0]^T & i = 1(1)\frac{n-1}{2} \\ [0, \ldots, 0]^T & i = \frac{1}{2}(n+1) \\ [\underbrace{0, \ldots, 0}_{n-i+1}\ a_{n-i+2,i} \cdots a_{i-1,i}\ 0, \ldots, 0]^T & i = \frac{n+3}{2}(1) n \end{cases} \tag{5.6.5a}$$

$$Z_i = \begin{cases} [\underbrace{0, \ldots, 0}_{i}\ a_{i,i+1}, \ldots, a_{i,n-i},\ 0, \ldots, 0]^T & i = 1(1)\frac{n-1}{2} \\ [0, \ldots, 0]^T & i = \frac{n+1}{2} \\ [\underbrace{0, \ldots, 0}_{n-i+1}\ a_{i,n-i+2}, \ldots, a_{i,i-1},\ 0, \ldots, 0]^T & i = \frac{n+3}{2}(1) n \end{cases} \tag{5.6.5b}$$

and for n even,

$$W_i = \begin{cases} [\underbrace{0, \ldots, 0}_{i}\ a_{i+1,i}, \ldots, a_{n-i,i},\ 0, \ldots, 0]^T & i = 1(1)\frac{n}{2} - 1 \\ [0, \ldots, 0]^T & i = n/2,\ \frac{n}{2} + 1 \\ [\underbrace{0, \ldots, 0}_{n-i+1}\ a_{n-i+2,i}, \ldots, a_{i-1,i},\ 0, \ldots, 0]^T & i = \frac{n}{2} + 2(1) n \end{cases} \tag{5.6.6a}$$

$$Z_i = \begin{cases} [\underbrace{0, \ldots, 0}_{i}\ a_{i,i+1}, \ldots, a_{i,n-i},\ 0, \ldots, 0]^T & i = 1(1)\frac{n}{2} - 1 \\ [0, \ldots, 0]^T & i = \frac{n}{2}, \frac{n}{2} + 1 \\ [\underbrace{0, \ldots, 0}_{n-i+1}\ a_{i,n-i+2}, \ldots, a_{i,i-1},\ 0, \ldots, 0]^T & i = \frac{n}{2} + 2(1) n. \end{cases} \tag{5.6.6b}$$

Substituting for A in (5.6.1) yields the various QI iterative schemes, by an analogous reasoning to that given in Section (2.4). First we replace the system (5.6.1) by the equivalent system,

$$Ex^{(k+1)} = Fx^{(k)} + b \ . \tag{5.6.8}$$

Then the simultaneous (or Jacobi) QI iterative method is then defined by,

$$E = X \text{ and } F = W + Z \ , \tag{5.6.9}$$

and the vector $x^{(k+1)}$ can be determined by solving the 2×2 systems,

$$\begin{bmatrix} a_{ii} & a_{i,n-i+1} \\ a_{n-i+1,i} & a_{n-i+1,n-i+1} \end{bmatrix} \begin{bmatrix} x_i \\ x_{n-i+1} \end{bmatrix}^{(k+1)} = \begin{bmatrix} c_i \\ c_{n-i+1} \end{bmatrix}^{(k)} \qquad i=1(1)\left\lceil \frac{n+1}{2} \right\rceil \tag{5.6.10}$$

where,

$$c_i^{(k)} = -\sum_{j=1}^{n} a_{ij}x_j^{(k)} + b_i \ , \qquad , \ j\neq i \text{ and } n-i+1 \tag{5.6.11}$$

$$c_{n-i+1}^{(k)} = -\sum_{j=1}^{n} a_{n-i+1,j}x_j^{(k)} + b_{n-i+1}$$

By introducing an acceleration parameter ω the simultaneous over-relaxation version of the Jacobi QI iterative method is given when,

$$\begin{bmatrix} a_{ii} & a_{i,n-i+1} \\ a_{n-i+1,i} & a_{n-i+1,n-i+1} \end{bmatrix} \begin{bmatrix} x_i \\ x_{n-i+1} \end{bmatrix}^{(k+1)} = (1-\omega) \begin{bmatrix} a_{ii} & a_{i,n-i+1} \\ a_{n-i+1,i} & a_{n-i+1,n-i+1} \end{bmatrix}^{(k)} * \begin{bmatrix} x_i \\ x_{n-i+1} \end{bmatrix}^{(k)} + \omega \begin{bmatrix} c_i \\ c_{n-i+1} \end{bmatrix}^{(k)} \tag{5.6.12}$$

replaces (5.6.10) and,

where $c_i^{(k)}$ and $c_{n-i+1}^{(k)}$ are defined in (5.6.11).

The successive (or Gauss-Seidel) QI iterative method is defined by

$$E = (X-W) \text{ and } F = Z \ , \tag{5.6.13}$$

and the $x^{(k+1)}$ values are given by (5.6.10) with,

$$c_i^{(k)} = -\sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=n-i+2}^{n} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^{n-i} a_{ij}x_j^{(k)} + b_i$$

$$c_{n-i+1}^{(k)} = -\sum_{j=1}^{i-1} a_{n-i+1,j}x_j^{(k+1)} - \sum_{j=n-i+2}^{n} a_{n-i+1,j}x_j^{(k)} -$$

$$\sum_{j=i+1}^{n-i} a_{n-i+1,j}\, x_j^{(k)} + b_{n-i+1}, \quad j\neq i,\ j\neq n-i+1 \qquad (5.6.14)$$

By introducing the acceleration parameter ω produces the successive overrelaxation (extrapolated Gauss Seidel) QI iterative method results when (5.6.14) is substituted in (5.6.12). Applying the QI permutation to the above iterative algorithms illustrates that they are simply 2×2 block forms of the more familiar point Jacobi and Gauss Seidel methods. It follows that the normal conditions of diagonal dominance or positive definiteness of A ensures that the iteration matrix derived from (5.6.9) and (5.6.13) converges. Thus, for any initial starting vector $x^{(0)}$ the QI methods converge to the solution x of (5.6.1) and also implies,

$$\det\left(\begin{bmatrix} a_{ii} & a_{i,n-i+1} \\ a_{n-i+1,i} & a_{n-i+1,n-i+1} \end{bmatrix}\right) \neq 0\ , \quad i=1(1)\lceil n/2\rceil$$

to ensure that X^{-1} exists. The corresponding overrelaxation QI forms are convergent when $0<\omega\leq 1$ and $0<\omega<2$ for the Jacobi and Gauss-Seidel respectively provided the original method converges. Proofs of these statements can be found in Evans & Sojoodi-Haghighi [84], but are special cases of the more general group iterative results.

The systolic implementation of the QI schemes adopts the same cascaded form Fig.(3.2.3.1) and uses 2×2 block calculations in the linear arrays. If we represent the general QI scheme by,

$$\bar{x}^{(k+1)} = \bar{B}\bar{x}^{(k)} + d \ , \qquad (5.6.15)$$

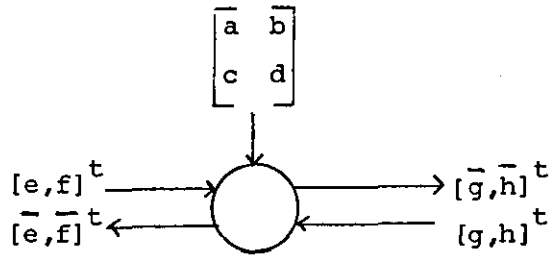
where $\bar{B}=\bar{E}^{-1}\bar{F}$ is the QI permuted form of the iteration matrix such that $B=X^{-1}(W+Z)$ for the Jacobi form and $B=(X-W)^{-1}Z$ for the Gauss-Seidel scheme and $d=E^{-1}b$. The 2×2 block form of Theorem (3.2.3.1) is easily derived.

Theorem 5.6.1: For r iterations of the form $x^{(k+1)}=Bx^{(k)}+d$ where B is an n×n matrix with block bandwidth $\bar{w}=\bar{p}+\bar{q}-1$, and d is an n×1 vector requires a time of $T\leq 2\{2\lceil n/2\rceil+r(2\bar{p}-1)+MAX(\bar{p}-1,\bar{q}-1)-\bar{p}+1\}$ ips cycles and takes 4rw equivalent ips cells.

Proof:

The basic linear array in the cascaded form Fig.(3.2.3.1) is a 2×2 block matrix vector array, thus using similar reasoning to that given in Theorem (5.1.1), gives the input length $2\lceil n/2\rceil$ and the number of block ips cells as $\bar{w}$, where $\bar{p}=\lceil p/2\rceil$ and $\bar{q}=\lceil q/2\rceil$. Each block ips requires 4 point ips cells for its implementation and 2 point ips cycles for calculation. Applying this information to Theorem (3.2.3.1) produces the required T and cell count.

The QI Jacobi array is derived directly from Fig.(3.2.3.2), by simply substituting 2×2 blocks for the input data elements and adopting the 2×2 block ips definition of (5.1.20b) and the revised form of (5.1.20c) below.



$$\begin{aligned} &D=ad-bc \\ &a=a/D,\ b=b/D,\ c=c/D,\ d=d/D \\ &\bar{g}=e,\ \bar{h}=f \\ &\bar{e}=ag-hb \\ &\bar{f}=cg-hd \end{aligned} \qquad (5.6.16)$$

Using pipelining to overlap the 2×2 inversion produces a boundary cell with the same cycle as a block ips and Fig.(3.2.3.2) implies that the Jacobi scheme has the same timing as a 2×2 block matrix vector problem of bandwidth $\bar{w}+1$. Thus, substituting $\bar{p}+1$ for $\bar{p}$ in Theorem (5.6.1) yields the time of the Jacobi form. Likewise, the QI Gauss Seidel form is constructed from a 2×2 block lower triangular solver, and an upper triangular array of bandwidths $\bar{q}$ and $\bar{p}-1$ respectively. As the lower triangular part introduces a delay of a single block ips cycle, the timing is derived from Theorem (5.6.1) with bandwidth $\bar{w}=\bar{p}$. Notice however that the cell counts produced by the Theorem are invalid due to the overhead of extra ips equivalents in the boundary cell, and that the bandwidth in the Gauss Seidel form represents only timing not the true bandwidth of the matrix (hence array). In our original estimate (5.1.20c) contained at most 10 ips equivalents, consequently, for r iterations the Jacobi scheme requires r(4w+10) ips equivalents and the Gauss Seidel method r(4w+6) ips equivalents.

Now the overrelaxation forms of these QI arrays are easily derived by writing (5.6.12) in the form,

$$\begin{bmatrix} \bar{x}_i \\ x_{n-i+1} \end{bmatrix}^{(k+1)} = \begin{bmatrix} \bar{a}_{ii} & a_{i,n-i+1} \\ a_{n-i+1,i} & a_{n-i+1,n-i+1} \end{bmatrix}^{-1} \left\{ (1-\omega) \begin{bmatrix} \bar{a}_{ii} & a_{i,n-i+1} \\ a_{n-i+1,i} & a_{n-i+1,n-i+1} \end{bmatrix} * \begin{bmatrix} \bar{x}_i \\ x_{n-i+1} \end{bmatrix}^{(k)} + \omega \begin{bmatrix} \bar{c}_i \\ c_{n-i+1} \end{bmatrix}^{(k)} \right\}$$

which reduces to, (5.6.17)

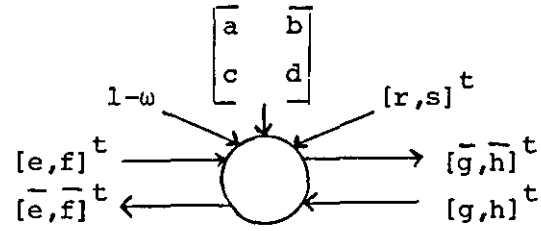
$$\begin{bmatrix} \bar{x}_i \\ x_{n-i+1} \end{bmatrix}^{(k+1)} = (1-\omega) \begin{bmatrix} \bar{x}_i \\ x_{n-i+1} \end{bmatrix}^{(k)} + \omega \begin{bmatrix} \bar{a}_{ii} & a_{i,n-i+1} \\ a_{n-i+1,i} & a_{n-i+1,n-i+1} \end{bmatrix}^{-1} \begin{bmatrix} \bar{c}_i \\ c_{n-i+1} \end{bmatrix}^{(k)}$$

(5.6.18)

From which we notice that the term ,

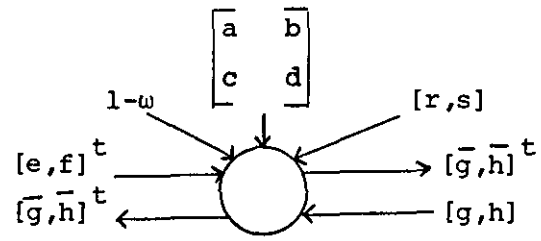
$$\omega \begin{bmatrix} \overline{a}_{ii} & a_{i,n-i+1} \\ a_{n-i+1,i} & a_{n-i+1,n-i+1} \end{bmatrix}^{-1} \quad i=1(1)\lceil n/2 \rceil \qquad (5.6.19)$$

can be precomputed before entering the cascaded iteration array. Definitions (5.6.16) and (5.1.20c) can then be revised to give more general block Jacobi and Gauss-Seidel arrays. For instance (5.6.16) becomes,



$$\begin{aligned} &r=(1-\omega)r,\ s=(1-\omega)s \\ &\bar{g}=e,\ \bar{h}=f \\ &\bar{e}=r+(ag-hb) \\ &\bar{f}=s+(cg-hd) \end{aligned} \qquad (5.6.20)$$

and (5.1.20c)



$$\begin{aligned} &r=(1-\omega)r,\ s=(1-\omega)s \\ &k=(e-g),p=(f-h) \\ &\bar{g}=r+(ak+bp) \\ &\bar{h}=s+(ck+dp) \end{aligned} \qquad (5.6.21)$$

where the 2×2 input is equivalent to (5.6.19). The calculation of the r and s values requires the $x^{(k)}$ vector to be delayed while the $c_i^{(k)}$ and $c_{n-i+1}^{(k)}$ values are computed. Consequently r,s and $c_i^{(k)}$, $c_{n-i+1}^{(k)}$ calculations can be overlapped. It follows that the most complex computation is the evaluation of $\bar{g}$ and $\bar{h}$ in (5.6.21) which is performed in 2 ips cycles, that is,

$t+\frac{1}{2}$: $k=(e-g)$, $p=(f-h)$

$t+1$: $t_0=a*k$, $t_1=c*k$

$t+\frac{3}{2}$: $t_2=r+t_0$, $b_3=s+t_1$, $t_0=b*p$, $t_1=d*p$

$t+2$: $\bar{g}=t_2+t_0$, $\bar{h}=t_3+t_1$

Consequently the above array timings are unchanged. This is an intuitive result, because when ω=1 overrelaxation methods become the standard Jacobi and Gauss Seidel schemes. As (1-ω) is known we require at most six point ips cells for the new boundary cells, this produces revised cell counts of $r(4\bar{w}+6)+10$ and $r(4\bar{w}+2)+10$ for over-relaxation Jacobi and Gauss-Seidel. The additional 10 ips account for the preprocessor evaluating (5.6.19).

These results on block iterative (or SQII) arrays are disappointing because we require approximately the same time and cover twice as much hardware as the point-orientated schemes. Some improvements for the Jacobi scheme is obtained if we adopt the pipelining strategy for (5.1.20b) which was used to reduce a block-ips cell requirement to two point ips cells. Recall that the cell count of Theorem (5.1.1) was roughly halved due to this pipelining effect, but that computation time remained unchanged, even though the pipelined block ips had an apparent cycle time of a single ips. The problem was the feedback loop associated with the boundary cell, which forced a two ips cycle time. In the Jacobi scheme the boundary avoids these feedback problems. Thus, by virtue of the fact that each Jacobi iteration appears like an extended matrix vector computation which utilises both speed up and reduced cell count to give a factor of 2 improvement in performance over the point schemes in Berzins, Buckley and Dew [83]. Closer analysis indicates that this optimised 2×2 explicit block matrix vector calculation (with two point ips cells per block cell) is a variation of the D-pipe (of Section (4.1)). We conclude that the D-pipe is simply an implicit block version of explicit block calculations. Furthermore from the simple observation that F in (5.6.9) is an X band matrix with null forward and backward

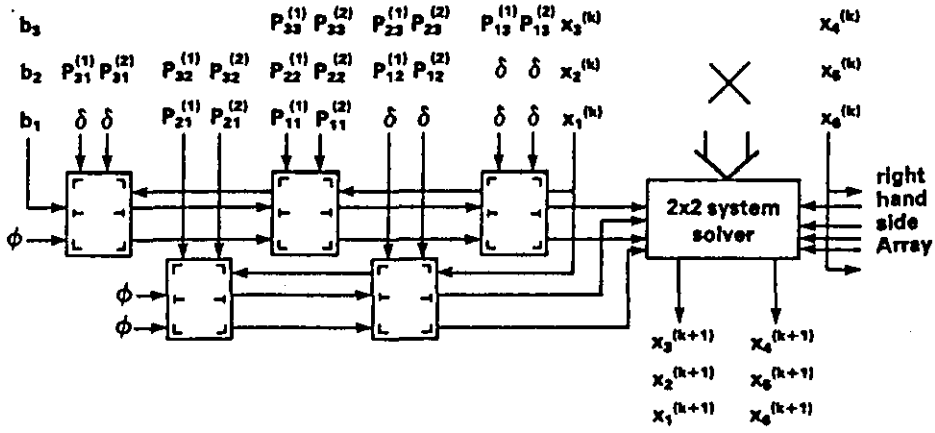
diagonals. It follows that a factor of 4 speedup can be achieved using the D^2-pipe arrangement for the Jacobi iteration as shown in Fig.(5.6.1), where the p_{ij} inputs are derived from the partitioning of (4.1.10)-(4.1.15) applied to F. Tracing the operation of the D^2-pipe Jacobi iteration we can derive the following theorem.

Theorem 5.6.2: For r iterations of the QI Jacobi method applied to the n×n system Ax=b where A has forward and backward bandwidths w_1 and w_2 can be computed in $T=\frac{1}{2}n+(r-1)(p+1)+MAX(\lfloor p/2\rfloor,\lfloor q/2\rfloor)+\lceil p/2\rceil+1$, where $w=p+q-1=MAX(w_1,w_2)$ and uses $r(4w+10)$ ips cells.

Proof:

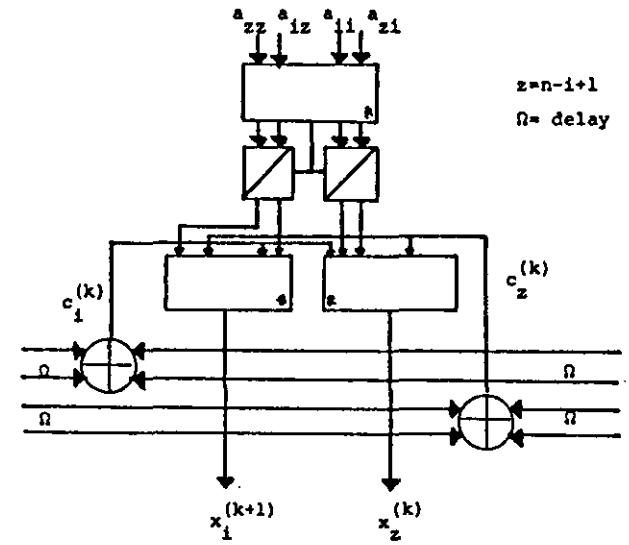
From Theorem (4.1.3) the timing of a single Jacobi iteration is given by $T=n/2+w/2+c$ where c=3 is the combined latency of the D^2-pipe adders and the 2×2 solver in Fig.(5.6.1). It follows from the D^2-pipe arrangement that the latency of each iteration is p+1 ips cycles. Allowing $MAX(\lfloor p/2\rfloor,\lfloor q/2\rfloor)$ cycles of synchronisation on the first iteration, the rth iteration starts computing after $(r-1)(p+1)+MAX(\lfloor p/2\rfloor,\lfloor q/2\rfloor)$ cycles. The total output length is $\frac{1}{2}n$ (see 2×2 solver input) and the first output is delayed by $\lceil p/2\rceil+1$ cycles while $c_i^{(k)}$ and $c_{n-i+1}^{(k)}$ accumulate their terms. The timing follows. The area bound is simply found by observing that the left and right D^2-pipes of a single iteration require 2w ips cells each, and that the 2×2 solver uses at most 10 ips equivalents, giving a total of 4w+10 ips cells per iteration.

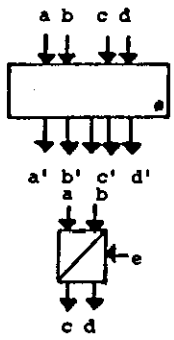
Notice that the D^2-pipe symmetry not only reduces the input output data length but minimises the latency of each iteration. For example, when $w_1=p_1+q_1-1$ and $w_2=p_2+q_2-1$ applying the QI permutation produces a 2×2 block form with $w=(p_1+p_2)+(q_1+q_2)-1$ and iteration latency $(p_1+p_2)+1$, whereas the D^2-pipe has latency $MAX(p_1,p_2)+1$.



0($n/2$) Single Iteration of XWZ Double IPS Pipe

FIGURE 5.6.1a





```
time
 t       read a,b,c,d
 t+1     output e=ac-bd
                 a'=a, b'=b, c'=c, d'=d
 t       read a,b,e
 t+1     output c=a/e, d=b/e
```

FIGURE 5.6.1b: 2×2 system solver

The D^2-pipe iteration is easily extended to Jacobi overrelaxation methods, but cannot be applied to Gauss-Seidel forms. The reason is simple, the feedback loop associated with the triangular solver portion of an iteration prevents the application of the D^2-pipe partitioning. Consequently, an improvement can only be made if the permuted form of (X+W) has a null first subdiagonal block, so that a block version of Fig.(4.2.1) can be constructed.

Finally we consider the input format of the last solution of (5.6.10) when n is odd and the system reduces to a single equation. The computation involving $c_i^{(k)}$ or $c_{n-i+1}^{(k)}$ is preserved easily with block cell inputs of the form,

$$\begin{bmatrix} y \\ 0 \end{bmatrix} = \begin{bmatrix} y \\ 0 \end{bmatrix} + \begin{bmatrix} a & b \\ 0 & 0 \end{bmatrix} \begin{bmatrix} c \\ d \end{bmatrix}$$

and

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} + \begin{bmatrix} a & 0 \\ b & 0 \end{bmatrix} \begin{bmatrix} c \\ d \end{bmatrix}$$

for block sub- and super-diagonal blocks respectively. If we also represent the last equation of (5.6.10) by the system,

$$\begin{bmatrix} a & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ 0 \end{bmatrix} = \begin{bmatrix} c \\ 0 \end{bmatrix}$$

it follows that the boundary cells of each iteration produce,

$$\begin{bmatrix} x \\ 0 \end{bmatrix} = \frac{1}{a} \begin{bmatrix} 1 & 0 \\ 0 & a \end{bmatrix} \begin{bmatrix} c \\ 0 \end{bmatrix} = \begin{bmatrix} c/a \\ 0 \end{bmatrix}$$

which also preserve computation. Consequently only changes to input format not the cell operation are required when n is odd.

5.7 SUMMARY

In this chapter we examined the suitability of Quadrant Interlocking methods for the systolic solution of linear systems. The QI permutation which can be formalised as a matrix P with elements p_{ij} given by,

$$p_{ij} = \begin{cases} p_{i,2i-1} = 1 & i=1(1)\frac{(n+1)}{2} \\ p_{\frac{n+1}{2}+i,n-2i+1} = 1 & i=1(1)\frac{(n-1)}{2} \end{cases} \qquad (5.7.1)$$

for n odd and

$$p_{ij} = \begin{cases} p_{i,2i-1} = 1 & \\ p_{\frac{n}{2}+i,n-2i+2} = 1 & i=1(1)n/2 \end{cases} \qquad (5.7.2)$$

for n even,

was utilised to solve data localisation problems in array dataflow. A side effect of this indicated that,

$$X = PDP^T, \quad W = PLP^T, \quad \text{and } Z = PUP^T \qquad (5.7.3)$$

where L,D and U were 2×2 block lower, diagonal and upper triangular matrices respectively. Consequently all QI arrays were special block forms of traditional systolic arrays which utilised pre- and post-permutations according to (5.7.3) to maintain sparsity patterns.

For QIF methods the ordinary scheme reduced immediately to the 2×2 block scheme of Robert [85] with computation time T=2n+min(p,q) for a matrix of bandwidth w=p+q-1 and efficiency e=½. The idea of implicit block calculations was introduced with the modified QIF form which allowed the ordinary point oriented calculations to be performed on block structured arrays, retaining their improved efficiency and computation time without introducing the problems of solving block structured coupled systems after factorisation. This concept was

extended to produce an orthogonally connected triangular array with computation time T=2.5n+s (where s was the block bandwidth) and a reduced number of processors for the root free Choleski factorisation.

From these experiments it was established that the QI permutation could be used most effectively to minimise the bandwidth hence cell count of factorisation arrays for the X-band matrix. Thus the QI methods are most suitable for solving periodic and circulant type matrices systolically. With this knowledge the O(n) and p-cyclic BATS cells in Chapter 4 were redesigned to improve throughput and in the latter case stability, producing a faster circulant system solver.

Block Gaussian Elimination was considered next, but proved less successful. The 2×2 block form gave rise to an elimination array with T=2.5n and improved efficiency which resulted from the fact that the equivalent of two rows of point ips cells were started in parallel rather than sequentially. A preprocessor was also defined which re-formatted the standard elimination input 'on the fly' to neatly expand the bandwidth of the host to the size of the block array.

Finally 2×2 block forms of the Jacobi and Gauss-Seidel iterative schemes were considered and shown to use twice the number of cells of the ordinary point schemes without improving computation time. The use of a pipelining trick produced the same cell count as the point arrays and established a relationship between the 2×2 block matrix vector arrays and double pipes. Using this correspondence a fast D^2-pipe Jacobi iteration was defined which minimised iteration latency and produced a factor of four speed-up over the point version while using only twice the number of cells.

CHAPTER 6

SYSTOLIC PRECONDITIONING AND INCOMPLETE ARRAYS

"Making Workable choices occurs in a crucible of Informative mistakes. Thus intelligence accepts fallibility. And when absolute (fallible) choices are not known, Intelligence takes chances with limited data in an arena where mistakes not only are possible but necessary".

- Darwi Odrade
extract from *"CHAPTER HOUSE DUNE"*, by Frank Herbert.

In this chapter we are concerned with systolic algorithms and arrays for preconditioned iterative procedures used in the solution of systems of linear finite difference equations derived from partial differential equations. For boundary value problems the resulting coefficient matrix is large and sparse (i.e. narrow bandwidth). The so-called preconditioned methods are primarily aimed at increasing the convergence rate of iterative techniques used to solve these linear systems by pre(post)-multiplication of the system by a suitable additional preconditioning matrix. Hence the use of preconditioning increases the arithmetic work in the solution process, which must then be offset against the greatly improved convergence rate to produce a faster algorithm.

With respect to systolic arrays we are mainly interested in the representation of this additional computation associated with preconditioning which may offer overall reductions in cell count and computation time due to a reduced number of iterations for existing systolic designs. Relevant reading on preconditioning can be found in Evans & Lipitakis [79], Evans [83c], Evans & Lipitakis [83], Lipitakis & Evans [80], Lipitakis [78].

6.1 BASIC PRECONDITIONING METHODS

Let

$$Au = d \ , \tag{6.1.1}$$

be a system of n linear equations, where A is positive definite, non-singular, banded, and of large order. Such problems arise from the application of finite difference techniques to the solution of partial differential equations. Now, clearly if A is easily invertible (6.1.1) is trivially solved by,

$$u = A^{-1}d \ . \tag{6.1.2}$$

However, it is often the case that difficulties arise in the construction of A^{-1}, especially when A results from the discrete approximation of P.D.E.'s on a grid. The key concept of preconditioning is to select some nonsingular matrix R, where R^{-1} approximates the inverse of A but whose construction is much simpler than forming A^{-1} itself, (6.1.1) is then easily preconditioned by the premultiplication of R^{-1} to give,

$$R^{-1}Au = R^{-1}d \ , \tag{6.1.3}$$

where R is termed the conditioning matrix. From (6.1.3) the general iterative procedure,

$$u^{(i+1)} = u^{(i)} + \alpha R^{-1}(d-Au^{(i)}) \ , \tag{6.1.4}$$

follows naturally and is consistent with (6.1.1) provided $\alpha \neq 0$ and R is nonsingular. Now if $\alpha=1$ and $R^{-1}=D^{-1}$ (where D is the diagonal matrix of A) (6.1.4) corresponds to (2.4.1.3), or the Jacobi method. Consequently, a careful choice of α and R^{-1} can produce many of the basic iterative methods. For $\alpha>1$ we produce the simultaneous displacement method of (2.4.1.4) and application of Theorems (2.4.1) and (2.4.4) indicates that the asymptotic rate of convergence is given by,

$$R_\infty = -\log_n|\rho(I-\alpha A)| \ , \tag{6.1.5}$$

which after some further manipulation produces,

$$R_\infty = \frac{2}{P} \ , \tag{6.1.6}$$

where P is the P-condition number from definition (2.2.8). Equation (6.1.6) indicates that the rate of convergence is inversely proportional to the condition number of A, a result which generalises to many other iterative techniques, see Evans [83c]. Consequently an improved convergence rate is achieved if the condition matrix R approaches A^{-1}

such that the condition number of $R^{-1}A$ is much less than the corresponding value of A. Furthermore the parameter α can be chosen according to Theorem (2.4.4) to minimise P and the number of iterations (i.e. maximise the convergence rate).

Using this basic preconditioning principle we can define two main classes of preconditioned iterative methods for the solution of (6.1.1) termed implicit and explicit preconditioning respectively. The main difference between the two schemes being the choice of the preconditioning matrix. If R is known or easily found (6.1.4) must be re-arranged to produce the implicit scheme,

$$R(u^{(i+1)}-u^{(i)}) = \alpha(d-Au^{(i)}) \quad , \tag{6.1.7}$$

If we denote $\Delta u=u^{(i+1)}-u^{(i)}$ and $e=(d-Au^{(i)})$ (6.1.7) is then equivalent to the solution of the linear system,

$$R\Delta u = \alpha e. \tag{6.1.8}$$

Using (6.1.8) implicit schemes can be further subdivided into compact and sparse forms.

In a compact form R in (6.1.8) is based on a factorisation of A into easily invertible (i.e. solvable) matrices such as lower and upper triangular type factors. However a direct factorisation of A especially with sparsely banded matrices leads to fill-in increasing memory requirements of conventional algorithms. Compact preconditioning controls the fill-in problem by the production of incomplete factors L_s and U_s, where $R=L_sU_s \tilde{} A$ such that the error norm (A-R) is kept to a minimum. Table (6.1.1) lists some of the factorisation strategies which have appeared in the literature to date, and are not discussed in detail here, except for the following remarks. First it must be clear that each method (in Table (6.1.1)) adopts its own strategy for retaining the

	Exact Method	R conditioning matrix	Iterative Method
1	Gaussian Elimination $A=\mathcal{L}^{-1}\mathcal{U}$	$\mathcal{L}_s^{-1}\mathcal{U}_s$	Evans (1974)
2	Triangular Factorisation $A=\mathcal{L}\mathcal{U}$	$\mathcal{L}_s\mathcal{U}_s$	Stone (1968), Evans & Lipitakis, (1979)
3	Choleski Square Root $A=QQ^T$	$Q_sQ_s^T$	Dupont (1968), Meijerink & Van der Vorst (1977), Gustafsson (1978)
4	Root Free Factorisation $A=\mathcal{L}\ D\ \mathcal{U}$	$\mathcal{L}_sD_s\mathcal{U}_s$	Evans & Lipitakis (1982)
5	Root Free Choleski $A=\mathcal{L}\ D\ \mathcal{L}^T$	$\mathcal{L}_sD_s\mathcal{L}_s^T$	Kershaw (1978)
6	Normalised Symmetric Factorisation $A=DTT^TD$	$D_sT_sT_s^TD_s$	Varga (1960), Evans & Lipitakis (1980)

TABLE (6.1.1): Approximate factorisation methods

α	R	Iterative Method
1	I	Jacobi (J)
$2/(\bar{a}+\bar{b})$	I	Simultaneous Displacement (SD)
1	(I-L)	Gauss-Seidel (GS)
ω	(I-ωL)	Successive Overrelaxation (SOR
ω(2-ω)	(I-ωL)(I-ωU)	Symmetric SOR (SSOR)
1	"	Preconditioned Jacobi(PJ)
$2/(\bar{a}+\bar{b})$	"	Preconditioned Simultaneous Displacement (PSD)
r	(I+rH)(I+rV)	Douglas-Rachford ADI (DR-ADI)
2r	(I+rH)(I+rV)	Peaceman-Rachford ADI(PR-ADI)
$2/(\bar{a}+\bar{b})$	(I+rH)(I+rV)	Alternating Direction Preconditioning (ADP), (or EADI)

TABLE (6.1.2): Iterative methods

where $\bar{a},\bar{b}$ are the smallest and largest eigenvalues of $R^{-1}A$.

sparsity of its L_s, U_s factors subject to the type of coefficient matrix. Second, the amount of work saved by producing the incomplete factors may not be large, consequently a valid criticism of compact preconditioning is that it may be better to produce a direct factorisation to solve (6.1.1) and avoid the iterative form (6.1.4) altogether.

In contrast, sparse preconditioning avoids the large amount of work of compact preconditioning by adopting factors based on a simple splitting of A. If we assume without loss of generality that the system (6.1.1) is normalized (6.1.4) represents all known first degree linear iterative schemes for suitable choices of α and R, indicated in Table (6.1.2) where A is assumed to have the form,

$$\text{a)}\quad A = I-L-U \text{ or} \qquad \text{b)}\quad A = H + V\ , \tag{6.1.9}$$

with L and U strictly lower and upper triangular matrices respectively and H,V are symmetric, positive definite matrices that commute (see Varga [62]). Thus if A can be suitably split into two matrices, the general form of R can be expressed as the product of these two matrices. That is,

$$R = (I-\omega L)(I-\omega U) = L_s U_s\ , \tag{6.1.10}$$

for (6.1.9a) and,

$$R = (I+rH)(I+rV) = L_s U_s\ , \tag{6.1.11}$$

for (6.1.9b) where ω and r are acceleration parameters associated with the method. We conclude that all known convergent iterative methods can be interpreted as improvements on the "condition" of the system (6.1.1) by a different choice of condition matrix R.

Explicit preconditioning provides a further alternative form to (6.1.3) where (6.1.1) is premultiplied by a matrix Q to yield,

$$QAu = Qd\ , \tag{6.1.12}$$

such that QA is a matrix with a simple splitting which produces a

novel iterative method. For instance, the Jacobi form of the explicit preconditioned method is given by the following formulation where (6.1.1) is assumed normalized. We set,

$$A = I - B \ , \tag{6.1.13}$$

and precondition with Q=(I+B) to obtain,

$$(I-B^2)u = (I+B)d \ , \tag{6.1.14}$$

which yields the iterative formula,

$$u^{(i+1)} = B^2u^{(i)} + (I+B)d \ , \tag{6.1.15}$$

Likewise the Gauss-Seidel iteration with the splitting (6.1.9a) is,

$$(I-L)u = Uu + d \ , \tag{6.1.16}$$

and using the preconditioning matrix Q=(I+L) yields the iterative formula,

$$u^{(i+1)} = L^2u^{(i)} + (I+L)(Uu^{(i)} + d) \ . \tag{6.1.17}$$

Notice that both these methods lack a parameter α which appears in (6.1.4) and in conjunction with the choice of R is used to control the condition number. Furthermore if we denote the eigenvalues λ_i, i=1(1)n of B in (6.1.13) such that,

$$a \leq |\lambda_i| \leq b < 1 \ ,$$

where a and b are the smallest and largest eigenvalues respectively, and B is convergent. The iteration matrix B^2 of (6.1.15) has eigenvalues $\mu_i=\lambda_i^2$, i=1(1)n satisfying,

$$a^2 \leq |\mu_i| \leq b^2 < 1$$

implying that the P-condition number is increased! From (6.1.6) this indicates that the rate of convergence of the preconditioned method is slower than that of the un-preconditioned form. This apparent contradiction in improving convergence rates by preconditioning methods can be resolved by considering the norm of the iteration matrix and its relation with the Neumann expansion. Let the linear system,

$$Cu = b \ , \tag{6.1.18}$$

have the matrix splitting C=G-H and the generalised iteration form,

$$u^{(i+1)} = Mu^{(i)} + z \ , \tag{6.1.19}$$

where $M=G^{-1}H$, and $z=G^{-1}b$. When (6.1.19) converges $||M||<1$ and from (2.4.3.10),

$$u^{(r)} = M^r u^{(0)} + (M^{r-1}+M^r+\ldots+M+I)z \tag{6.1.20}$$

with $u^{(0)}$ the initial starting vector. On convergence $M^r u^{(0)}=0$ and

$$u = (M^{r-1} + M^r + \ldots + M+I)z \tag{6.1.21}$$

thus

$$C^{-1} = (M^{r-1} + M^r + \ldots + M+I)G^{-1} \ . \tag{6.1.22}$$

Indicating that convergence is accelerated if more terms in the Neumann expansion of the iteration matrix are constructed on each iteration. For example, repeated substitution in (6.1.15) yields,

$$\left.\begin{aligned} u^{(1)} &= (I+B)d \\ u^{(2)} &= (B^3+B^2+B+I)d \\ u^{(3)} &= (B^5+B^4+B^3+B^2+B+I)d \end{aligned}\right\} \tag{6.1.23}$$

compared with,

$$\left.\begin{aligned} u^{(1)} &= d \\ u^{(2)} &= (B+I)d \\ u^{(3)} &= (B^2+B+I)d \end{aligned}\right\} \tag{6.1.24}$$

in the unpreconditioned Jacobi method. Notice that (6.1.23) collects two terms of the expansion every iteration compared with only one term per iteration for (6.1.24). Alternatively, the explicit nature of the preconditioning is exposed if we substitute $R=L_sU_s$ and (6.1.18) into (6.1.8) to yield,

$$L_sU_s\Delta u = b-Cu^{(0)} \ , \tag{6.1.25}$$

with $\Delta u=u^{(1)}-u^{(0)}$ and $\alpha=1$, which on re-arrangement gives,

$$\Delta u = U_s^{-1}L_s^{-1}b - U_s^{-1}L_s^{-1}Cu^{(0)} \ , \tag{6.1.26}$$

defining $L_s^{-1}=G^{-1}$ and $U_s^{-1}=(M^{r-1}+\ldots M+I)$ yields,

$$\Delta u = C^{-1}b - u^{(0)} \Rightarrow C^{-1}b = u^{(0)}+\Delta u \qquad (6.1.27)$$

indicating that if C^{-1} is known the solution u can be found directly, but more importantly that if some powers of M are known the explicit preconditioning method is accelerated by choosing $U_s^{-1}L_s^{-1}=R^{-1}$, such that it forms a closer approximation to C^{-1} (or A^{-1} in (6.1.1)).

In the following sections systolic arrays for the above preconditioned iterative schemes are developed. The arrays themselves correspond to a global array structure (see Fig.(6.1.1)), consisting of two stages, a pre-processing stage for preconditioning and an iteration stage for solution. Now, from a computational viewpoint the basic aim of preconditioning is to offset the increased arithmetic work introduced to

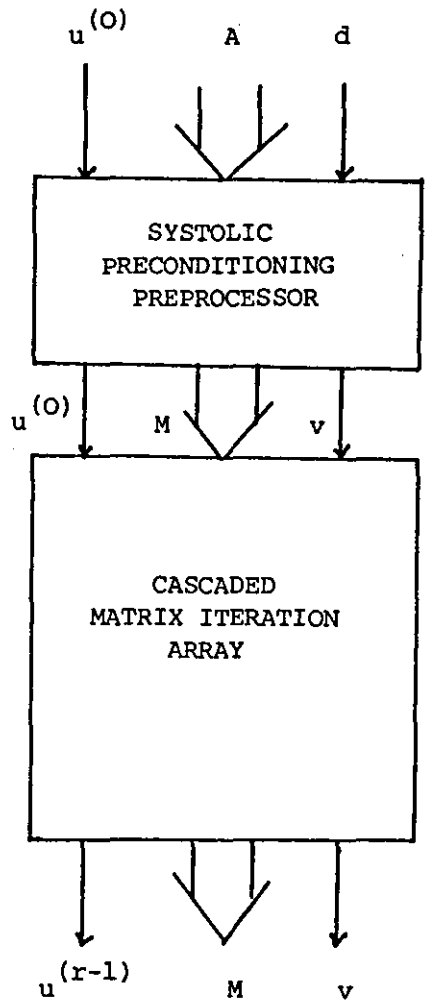


FIGURE 6.1.1: Global structure of a preconditioned solver

the solution process (due to ill-conditioning) against the greatly improved convergence rate of the method to produce a faster design. Thus the essential motivation of systolic preconditioning is a tradeoff of cells introduced to the preconditioning preprocessor against those removed from the iteration section of the array. To this aim appropriate pre-processing arrays must be developed which maximise overall cell reduction while retaining convergence rate improvements. Hence systolic preconditioners place an additional constraint on the choice of R.

6.2 HEXAGONAL MATRIX POWER GENERATION

The first preprocessor we consider is a pipelined array for generating the powers of a matrix. This problem has a certain intuitive appeal because in addition to facilitating the construction of (part or all) the Neumann expansion and performing explicit preconditioning, it also has much wider applications. For example the exp(A), sin(A), cos(A) and log(A) of a matrix A can all be expressed as matrix power series. The power generator also incorporates a recurrence type formulation which makes it attractive for systolic implementation, because the traditional matrix product array of Fig.(3.2.1.6) can be applied directly using a multipass formulation to yield,

$$
\begin{aligned}
&C_0 = M \\
&\text{for } i=1 \text{ TO } k \text{ DO} \\
&\quad \{A=C_{i-1}, B=C_{i-1} \\
&\quad C_i = Z + (A*B) \\
&\quad \}
\end{aligned}
\qquad (6.2.1)
$$

when Z=0, (6.2.1) generates the sequence M^{2^i} of successive matrix squares, by setting B=M the sequence M^{i+1} is produced, and when in addition $C_0 = Z = I$ the Neumann expansion. The start matrix M is also

banded, so each pass produces an increase in the bandwidth of C_i, i=1(1)k, and hence the number of cells on the next pass affects both area and time. To produce a fixed sized design the array must be big enough to allow the growth of the bandwidth on each pass. Suppose M has an initial bandwidth $w_o=p_o+q_o-1$, where p_o=number of super diagonals and q_o=the number of subdiagonals (including the main diagonal). A simple analysis gives the bandwidth of M^2 as $w_1=w_o+(p_o-1)+(q_o-1)=2w_o-1$, and so generally,

$$w_i = 2w_{i-1} - 1, \quad i=1(1)k \ , \tag{6.2.2}$$

is the bandwidth of M^{2^i}. It follows by repeated substitution in (6.2.2) that,

$$\begin{aligned} w_k &= 2w_{k-1}-1 = 2(2w_{k-2}-1)-1 \\ &\vdots \\ &= 2^k w_o - (2^{k-1}+2^{k-2}+ \ \ldots \ 2^o) \end{aligned}$$

which after the summation of the geometric progression produces,

$$w_k = 2^k(w_o-1) + 1 \ . \tag{6.2.3}$$

Alternatively if the sequence M^{i+1} is computed the bandwidth grows much more slowly according to,

$$\bar{w}_i = (i+1)[\bar{w}_o-1] + 1 \ , \tag{6.2.4}$$

which follows from induction on i with $\bar{w}_o=w_o$, i.e., for i=1,

$$\bar{w}_1 = 2\bar{w}_o-1 = (p_o+q_o-1) + (p_o+q_o-1) - 1$$

and generally,

$$\begin{aligned} \bar{w}_i &= \bar{w}_{i-1}+(p_o-1)+(q_o-1) \\ &= i(\bar{w}_o-1)+1+(\bar{w}_o-1) = (i+1)(\bar{w}_o-1)+1 \end{aligned}$$

and when $(i+1)=2^j$ substitution in (6.2.2) yields the result,

$$w_j = 2^j(w_o-1)+1 = (i+1)(w_o-1)+1$$

directly. Now as $\bar{w}_i<w_i$ the maximum number of hex cells required to compute all the powers up to and including M^{2^k} is w_{k-1}^2, as the bandwidths

of A and B are both at most w_{i-1} on each pass of (6.2.1). When the sequence $M^{(i+1)}$ i=1(1)k is computed A can have bandwidth $\overline{w}_i$ and B has constant bandwidth w_o on each pass, so that only $[(k+1)(w_o-1)+1]w_o < w_{k-1}^2$ cells are required and produce an increasingly skewed hex array as the bandwidth expands. Clearly these cell bounds hold only for small k which ensures that the last matrix in the sequence is also banded. However, if k is increased $\overline{w}_k$ and w_k approach 2n-1 and the matrix fills. Since the bandwidth of successive powers cannot exceed that of a full matrix an upper bound of $w^2=(2n-1)^2$ hex cells is established. It follows that this dense array can compute an infinite sequence of powers by multipass and from Theorem (3.2.1.6) k passes require T=k(5n-1) ips cycles.

REMARK: An orthogonal array for multipass powering has recently appeared in Quinton, Joinnault and Gachet [86] and indicates that Corollary (3.2.1.3) carries over to matrix power generation.

Finally, we can determine the maximum matrix power that can be computed for a given matrix before we use more hardware than the dense case, by the relations,

$$\begin{aligned} w_{k-1}^2 &\leq (2n-1)^2 \\ 2^{k-1}(w_o-1) &\leq 2(n-1) \\ k &\leq \lceil \log(n-1) - \log(w_o-1) + 2 \rceil \end{aligned} \tag{6.2.5}$$

for the sequence M^{2^i}, i=1(1)k and,

$$\begin{aligned} (k+1)(w_o-1)w_o + w_o &\leq (2n-1)^2 \\ k &\leq \frac{(2n-1)^2 - w_o - (w_o-1)w_o}{(w_o-1)w_o} \\ &\leq \frac{(2n-1)^2 - w_o^2}{(w_o-1)w_o} \end{aligned} \tag{6.2.6}$$

for the sequence M^{i+1}, i=1(1)k.

The above relations completely define the behaviour of systolic matrix power generation and a number of drawbacks can be identified which can help to characterise the type of preprocessor we should attempt to produce:

(i) The matrix square operations duplicate the matrix input bringing each element from the host memory more than once.

(ii) For the early powers of the banded matrix the bound of w_{k-1}^2 cells results in unused cells which reduces efficiency, and delays output of the array unnecessarily.

(iii) The total number of host input output lines is $3(w_1+w_2)$, where $w_1=w_2=w_{k-1}$ for repeated squaring, $w_1=\overline{w}_{k-1}$, $w_2=w_0$ for the sequence M^{i+1}, and $w_1=w_2=2n-1$ for the dense matrix, when w_1 and w_2 are the bandwidths of A and B respectively.

(iv) The array inputs from North West, North East, and South in Fig.(3.2.1.6) make pipelining difficult, and multi-pass iterations reduce throughput.

Below we describe a reduced bandwidth array for a matrix product, which can be pipelined and which incorporates optical concepts of a soft-systolic frame to avoid duplicate inputs by bringing the matrix M from the host only once. Figure (6.2.1) shows the global input structure of the reduced bandwidth array with data movement through the modified hex being shown as a ray diagram, using the principle of wavefront reflection (from optics). The main principle of the array is to add an upper boundary of cells on the north edge of the existing hex (of w_{k-1}^2 cells) which act as reflectors and mirrors for individual data sequences and wavefronts respectively. Tracing dataflow as illustrated by the snap-shots of Fig.(6.2.2) indicates that data moving south to north as it

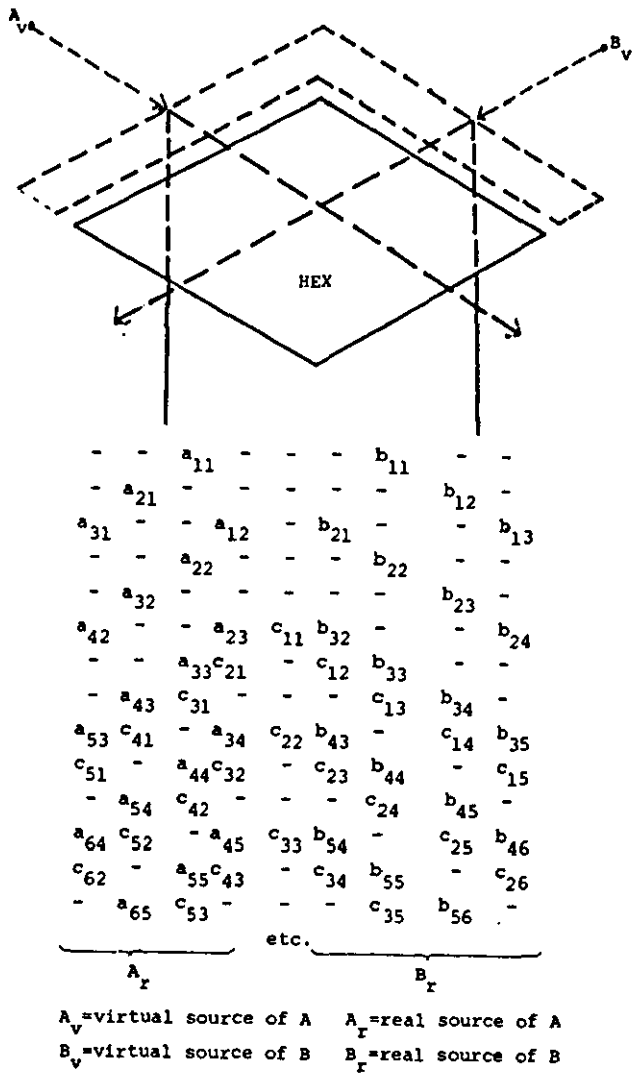


FIGURE 6.2.1: Reduced bandwidth input format

leaves the hex is incident on the mirror-like boundary and reflected back into the hex. The reflected data waves appearing on the NE and NW boundaries from the arrays viewpoint to emanate from two virtual sources A_v and B_v, mimicking the traditional hex input. This array, however, requires less host input/output connections as the NE and NW inputs are part of the array.

Now from the input format the result matrix C and real matrix

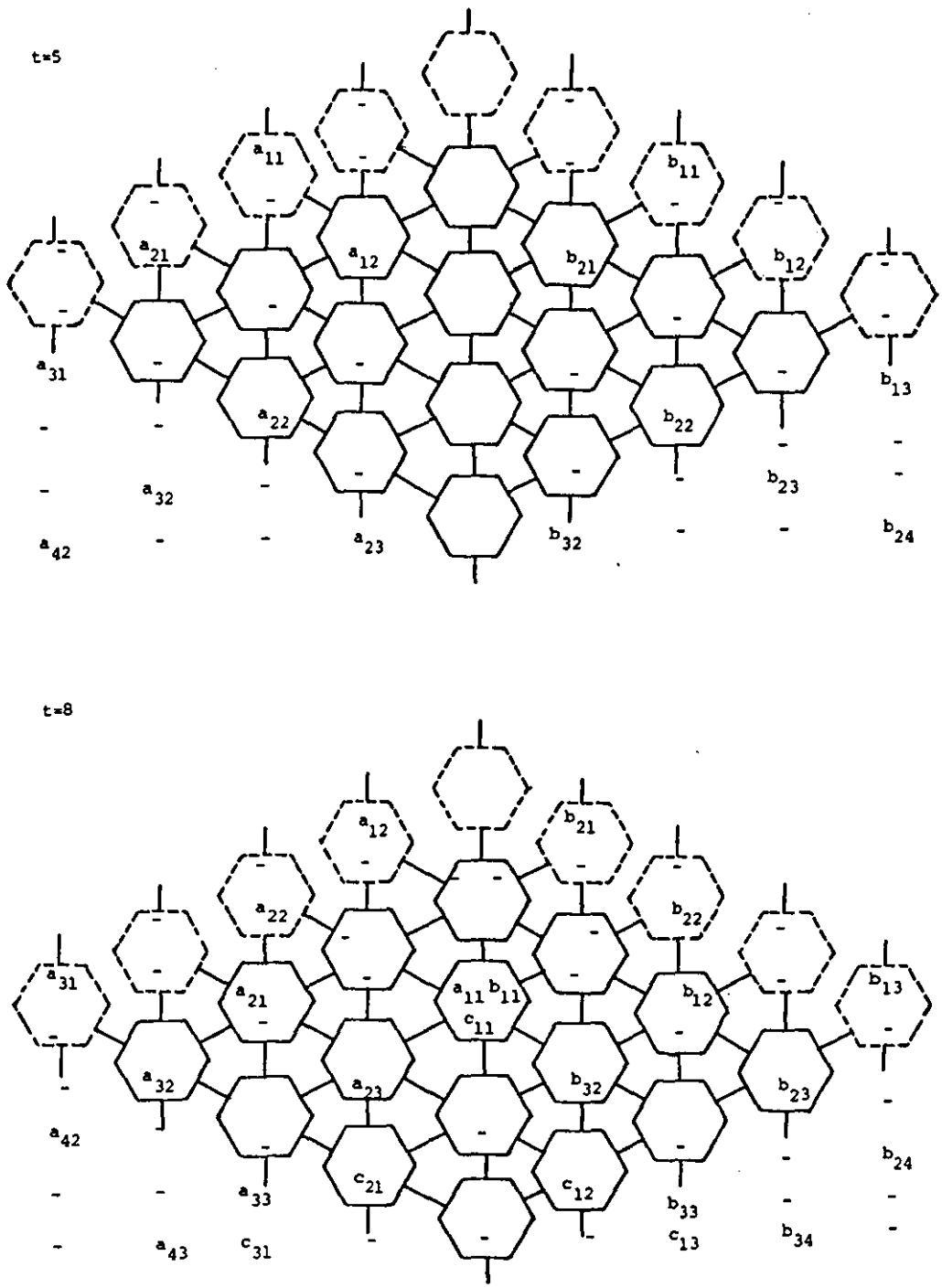


FIGURE 6.2.2: Reduced array snapshots

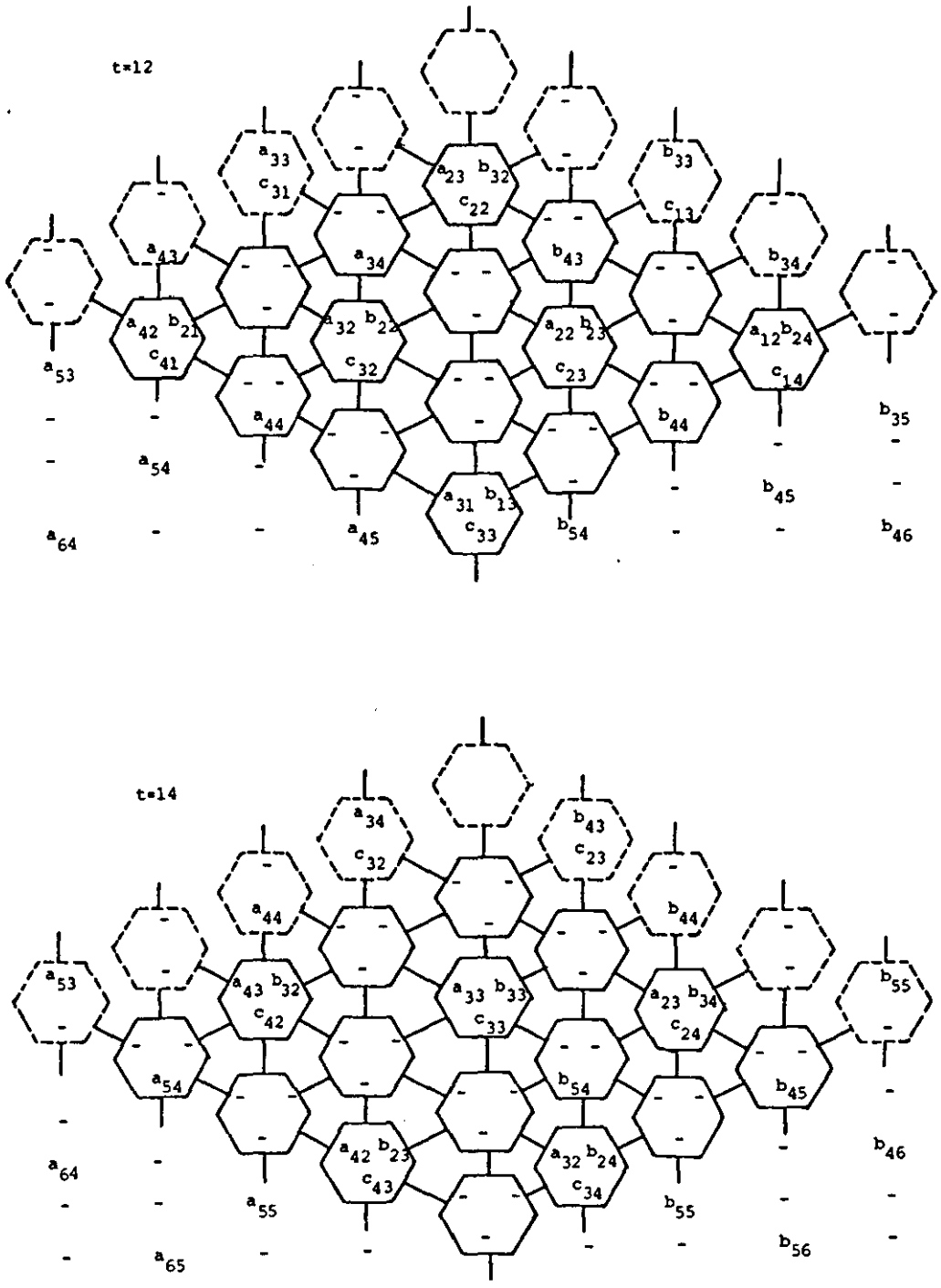
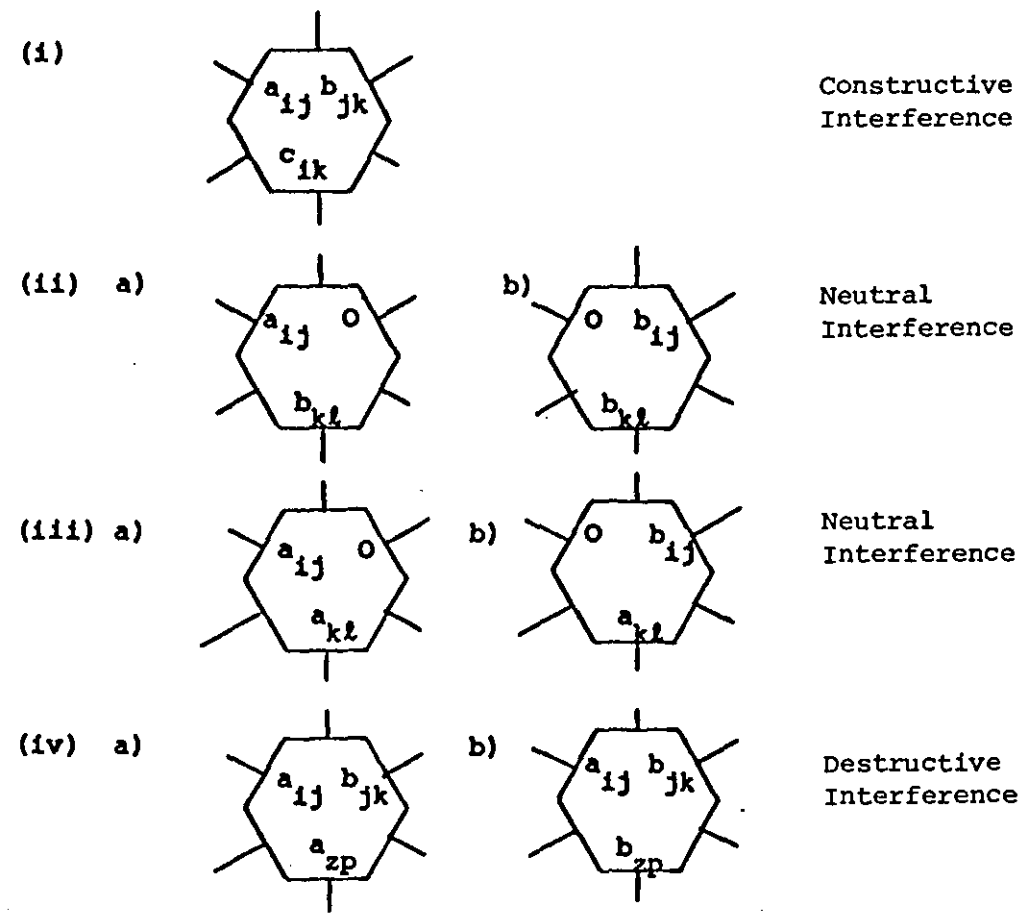


FIGURE 6.2.2: continued

inputs A_r,B_r must be multiplexed on the same south-north inputs. The key to the data multiplexing is the manner in which neutral elements or the holes of the C input are filled. To preserve computation, the data elements once inside the array must constructively interfere. In fact, we can identify just four types of interference for this particular problem, which are given below.



Case (i) is the true inner product step necessary for correct matrix production operation. Cases (ii) and (iii) represent the two possibilities of data moving SW and SE meeting an A_r or B_r element on its trip to the mirror section before reflection. This is a potentially disastrous situation; but the inner product (y=y+0*x) indicates that computation is neutral and preserves all inputs. This leaves only case (iv) where three

data elements meet in the same cell like case (i) but where no c_{ik} value is present. Clearly this type of operation would modify the A_r or B_r elements before reflection generating an incorrect product. But, the reflection in Fig.(6.2.1) is made so that virtual data inputs are the same as those in the traditional hex. It follows that only cases (i)-(iii) are possible, otherwise the original hex would miss the accumulation of at least one full partial product, and this is impossible because the traditional array is formally verified in Melhem & Rheinboldt [84]. However, closer analysis of the dataflow in Fig.(6.2.2) indicates that a situation similar to case (iv) can occur if A_r and B_r data is not placed correctly. The data elements are placed in the south input in a manner related to the bandwidths of A and B. When the matrix bandwidths are different the array becomes rectangular and skewed in the direction of the largest bandwidth, and the A,B inputs take different amounts of time to synchronise with C. Consequently, it becomes possible for a clash (where two inputs require the same slot) to occur. The south boundary of Fig.(6.2.1) has w_1+w_2+2 inputs and can be partitioned into non-overlapping regions which prevent A and B inputs from colliding, reducing the problem of clash resolution to the examples in Fig.(6.2.3). In case (i) A_{ij} elements clash with C_{ij} entries, case (ii) indicates a similar clash with the B_{ij} elements in an alternative non-clash position, and case (iii) indicates the simultaneous collision of A_{ij}, B_{ij} values with C_{ij} elements. A simple observation on dataflow shows that shifting the C_{ij} elements back along the input stream by a sufficient number of spaces resolves all clashes. Unfortunately shifting destroys the synchronisation necessary for constructive interference, and the A_{ij} and B_{ij} elements must be delayed somewhere in the array to resynchronise

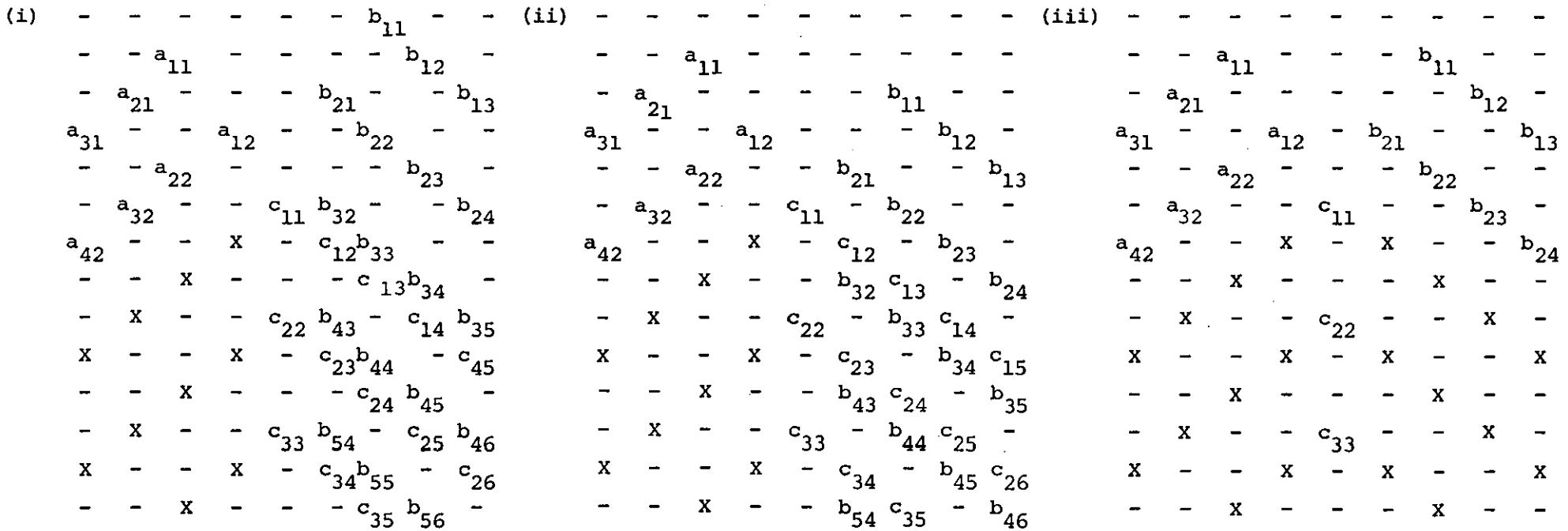
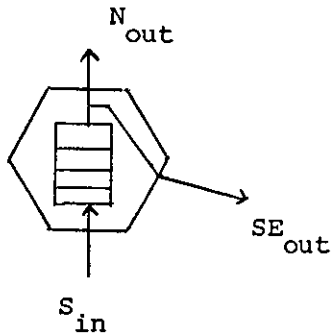


FIGURE 6.2.3: Clash resolution

data. The most likely place for delays is the mirror boundary. Reflecting cells are trivial to construct and consist of a small delay queue for resynchronisation and a switch for reflecting and non-reflecting states controlled by a single bit tagged to the south input elements which operates as follows:



IF TAG THEN SE_{out}=0
ELSE SE_{out}=Front-of-Queue
Back-of-Queue=S_{in}
N_{out}=Front-of-Queue

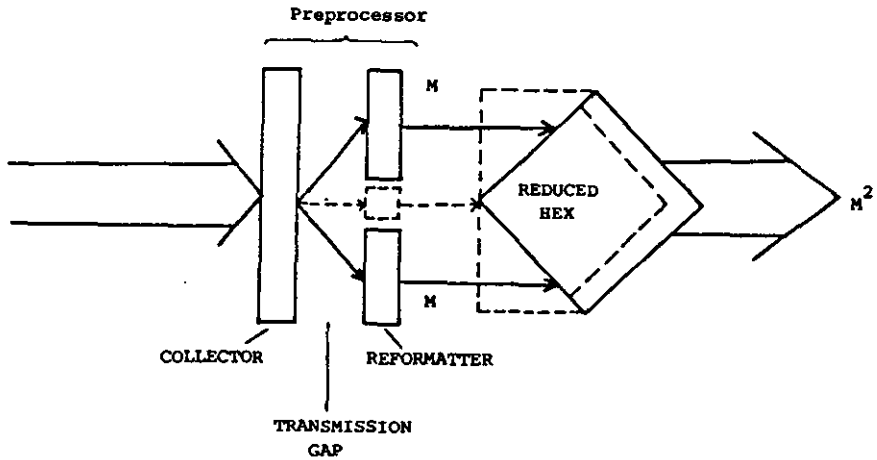
Thus, setting the tag bits of C_{ij} elements true prevents them from re-entering the array by reflection and upsetting subsequent computations.

Theorem 6.2.1: The product of two n×n matrices A and B with bandwidths w_1 and w_2 respectively can be computed on a Reduced Communication Bandwidth Hexagonal array of w_1*w_2 inner product cells and w_1+w_2-1 reflecting cells in $T=3n+(w_1+w_2)+2q$ ips cycles where $q>0$ is a small constant.

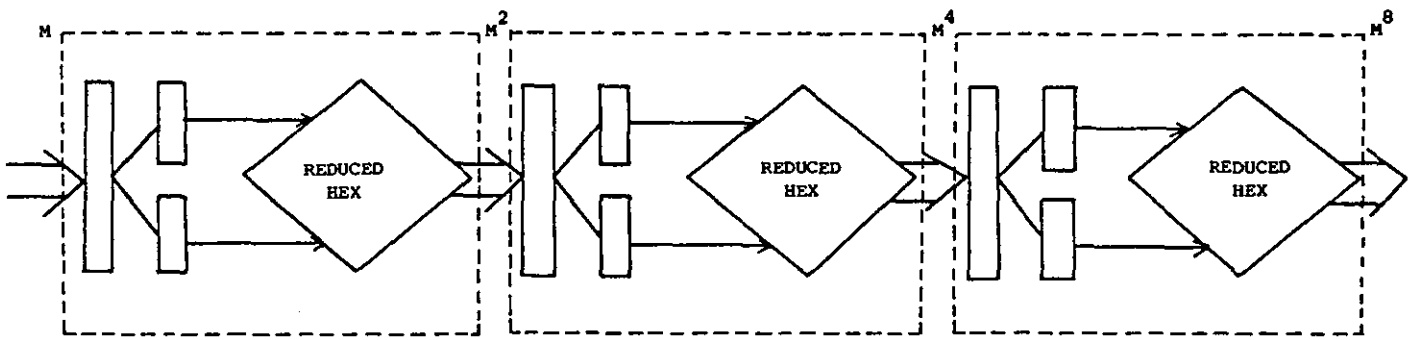
Proof: [By generalization of the snapshots in Fig.(6.2.2)].
The delay for positioning A_r or B_r depending on which has the longest distance to travel to the reflectors is $\min(w_1,w_2)$ (the longest vertical distance from south to north boundary). Reflected elements are delayed q cycles in the reflecting cell, and require at most $\max(w_1,w_2)$ for the c_{11} element to meet the reflected elements and accumulate its final value. After a further delay of q cycles through the mirror boundary c_{11} is output giving an array latency of $(w_1+w_2)+2q$ cycles. As the output length is 3n the theorem time follows immediately. The design requires an additional w_1+w_2-1 reflecting cells for the mirrored upper boundary

and the same ips cells as Theorem (3.2.1.6) producing the area bound. Finally the former hex required $3(w_1+w_2)$ input and output connections compared with $2(w_1+w_2)$ for this design saving a third of the host connections and reducing the communication bandwidth. Finally experiments with clash resolution indicates that q is small (i.e. 2,3) making the reflecting cell overhead negligable.

Next, to improve throughput and reduce input duplicates we introduce a preprocessor to convert the above general matrix product array into a pipelined power generator. The structure of a preprocessor for matrix squaring and its pipeline arrangement for the sequence M^{2^i} i=1(1)k are shown in Fig.(6.2.4), and consists of three major sections, a collector, transmission gap, and reformatter. The collector accepts a single matrix input in standard hexagonal matrix input representation and creates a duplicate. The transmission gap then separates the two representations into non-overlapping regions of the input to form a basis for the A_r and B_r inputs. Finally, a set of delay queues reformat the data to resolve any clashes, synchronisation, and tag bit alignment problems before input to the reduced hex array. Assuming a hex array of w_{k-1}^2 ips cells produces a fixed sized pipeline component and computing the matrix square operation simplifies data reformatting as the hex is more symmetrical than skewed because A_r and B_r have the same bandwidth. Furthermore, considering the detailed operation of the preprocessor indicates that with slight modifications it can also generate the sequence M^{i+1}, i=1(1)k and sum the first k+1 terms of the Neumann expansion. A detailed example of the preprocessor for some stage i in the pipelined generator is shown in Fig.(6.2.5). For simplicity the preprocessor can be partitioned into left, centre, and right sections,



a) Single pipeline stage



b) Pipelined power generator array

FIGURE 6.2.4: Pipelining of multipass power generation

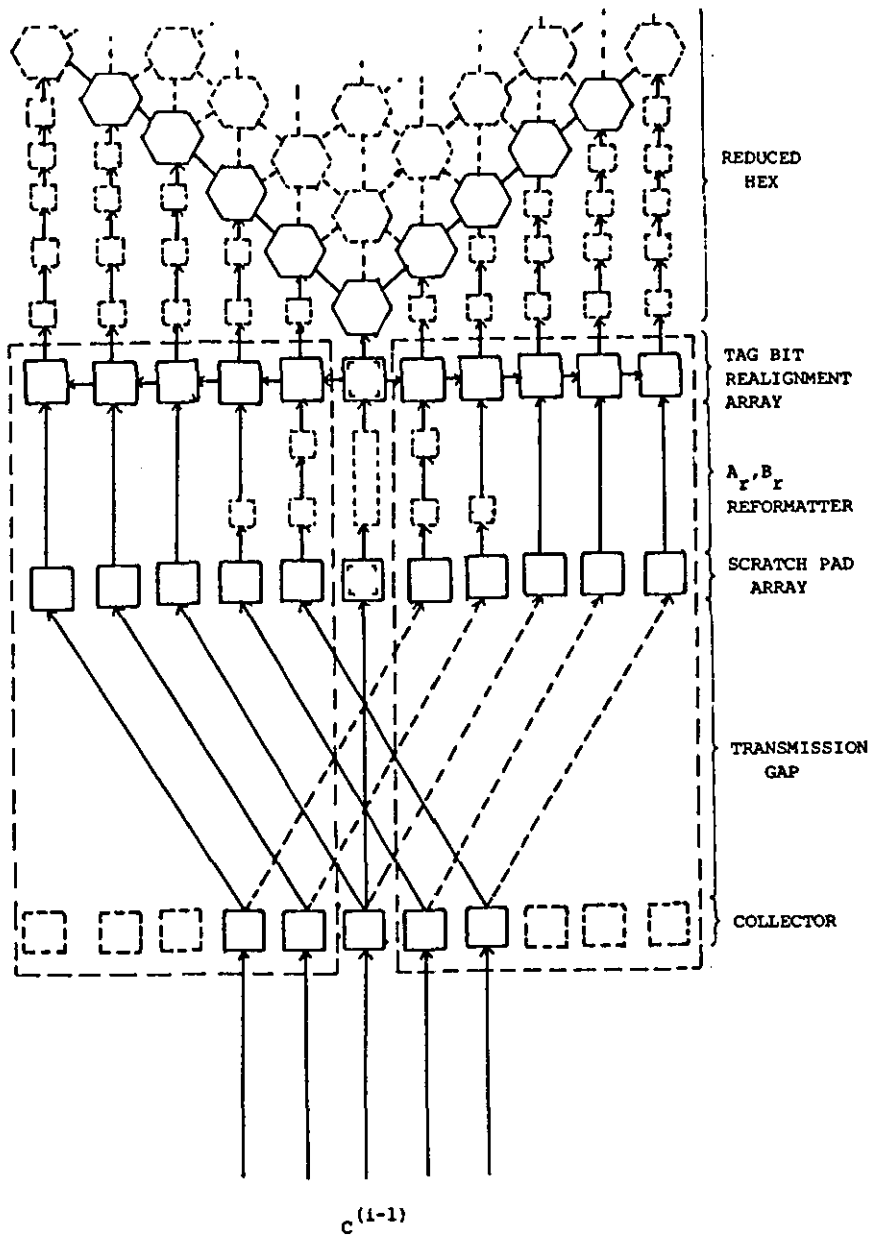


FIGURE 6.2.5: Detailed preprocessor design for pipelined power generation (using $w_{k-1}=5$)

where operation of the left and right parts is similar. It follows that by denoting the inputs and outputs of stage i by $A_r^{(i)}, B_r^{(i)}$ and $C_r^{(i)}$ respectively, that the preprocessor operation is described fully by the left and centre sections.

Now the first non-zero elements to leave stage (i-1) and enter the stage (i) preprocessor are the $A_r^{(i-1)}$ and $B_r^{(i-1)}$ values, which if allowed

to reach the stage (i) hex array could interfere with later computations and consequently must be removed. Removal of these elements is achieved by an array of scratch pad (SP) cells which checks the tag bit of each input element, resetting the data elements (to zero) for false bits, and the tag bit itself for true bits. This scrubs the stage (i-1) output to leave only $C_{pq}^{(i-1)}$ values non-zeros. The reformatting section then reformats this scrubbed input stream to generate $A_r^{(i)}$ from $C^{(i-1)}$ ($B_r^{(i)}$ in the right section) by delay queues organised to create the correct amount of skew. The next step is to re-generate the tag bit structure to identify the $C^{(i)}$ elements which will accumulate the partial products of stage (i). Fortunately assuming that each stage of the pipeline has w_{k-1}^2 cells simplifies identification of the new $C^{(i)}$ elements, because $A_r^{(i)}$ and $B_r^{(i)}$ can be given the same bandwidth (by a suitable padding of zeroes) at each stage. Hence the number of cycles between the leading elements $A_r^{(i)}(1,1)$, $B_r^{(i)}(1,1)$ and $C_{11}^{(i)}$ is always the same. Once the $C_{11}^{(i)}$ position is identified the remaining $C_{pq}^{(i)}$ elements are easily located using the regular chevron structure portrayed by Fig.(6.2.1). The tag bit alignment for stage (i) is then easily generated by a linearly connected unidirectional array of tag realignment cells, which pump control bits from right to left (left to right in the right preprocessor section) setting the tag bits of the reformatted $A_r^{(i)}$ ($B_r^{(i)}$) data as they pass vertically through the array. Deriving the control bit sequence from the suitably delayed tag bit sequence associated with $C_{pp}^{(i-1)}$ generates the correct chevron input for $C^{(i)}$. Thus, the centre section of the preprocessor consists simply of a special scratch pad cell which clears all elements but leaves tag bits unchanged, and a delay queue of size d_q, where, d_q is the number of cycles between $A_r^{(1)}(1,1)$ or $B_r^{(1)}(1,1)$

and $C_{11}^{(1)}$. The choice of measurement from $A_r^{(1)}(1,1)$ or $B_r^{(1)}(1,1)$ determined by the original input format and bandwidth of the starting matrix (M).

To produce the sequence M^{i+1} the right preprocessor section must be modified to allow the $B_r^{(i-1)}$ matrix to be resynchronised in stage (i). This is achieved by allowing data to pass straight through the collector and transmission gap (see Fig.(6.2.6)), modifying scratch pad cells to zero data elements with true tag bits and adding additional delays in

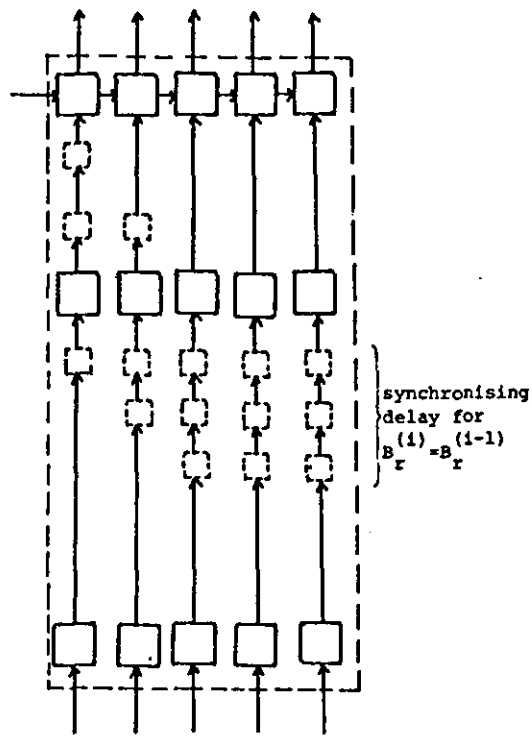


FIGURE 6.2.6: Alternate right section for producing Neumann expansion ($d_q=3$)

the reformatter delay queues so that $B_r^{(i)}=B_r^{(i-1)}$. The essential idea being to replace the $C^{(i-1)}$ values which would normally become $B_r^{(i)}$ by

$B_r^{(i-1)}$. Thus $B_r^{(1)}=B_r^{(2)}=\ldots=B_r^{(k)}=M$ and by definition, the Neumann expansion is also accumulated if the centre realignment cell is modified to set the data elements to one as well as tag bits, and with $A_r^{(1)}=I$, $B_r^{(1)}=M$ initially. We conclude that our pipeline computes all the multi-pass forms of (6.2.1) and state the following generalised theorem.

Theorem (6.2.2): k powers of an n×n matrix M of bandwidth w can be generated (and accumulated) using a k-stage pipeline of reduced bandwidth hex arrays of size w_{k-1}^2 where w_{k-1} is the bandwidth of the largest power in $T=3n+k[2w_{k-1}+2q+3]$ for q a small delay constant.

Proof:

Using Theorem (6.2.1) we note that the output delay of a single hex array is $2w_{k-1}+2q$, and from the preprocessor of Fig.(6.2.5) the delay for reformatting is at most 3 cycles. Hence for k stages the total output delay is $k(2w_{k-1}+2q+3)$, and as there are 3n results T follows directly. The area estimate follows from the assumption that each array contains w_{k-1}^2 ips cells, and that the preprocessor is made up only of switchable delay registers.

Finally we conclude this section with a few simple observations. First, the pipeline can be optimised by selecting smaller hexes in the earlier stages where the bandwidth is small. This complicates pre-processor reformatting at each stage but reduces cell count and improves pipeline latency. Second, the duplication of input matrix in the pre-processor can be implemented easily using electro-optical concepts. The idea being to convert the data from electrical to optical and back to electrical signals as it passes through the collector, across the transmission gap and back into the scratch pad array. The unpleasant non-planar part of the design in the transmission gap would then be

reduced to freespace or waveguided (via a glass wafer) transmission in which intersecting signals would not interact.

6.3 COMPACT SYSTOLIC ARRAYS FOR INCOMPLETE FACTORISATION METHODS

A preprocessor for implicit compact preconditioning relies on direct methods for the production of L_s and U_s factors, which are able to control the fill-in factor associated with large sparse system solution, so that the number of non-zeros created is small when compared with the nonzeros of the original coefficient matrix. From a numerical viewpoint this control must also produce an easily specified algorithm which uses available machine memory efficiently. A number of methods for omitting fill-in elements are available, for example:

a) restricting specified locations of the coefficient matrix to be filled in.

b) distributing storage equally among the rows of a matrix.

c) neglecting small elements - on the basis that they will not affect the result significantly.

d) fill the available storage then forget any fill-ins that cannot be accommodated.

We can translate fixed storage to fixed area and consider the implementation of an incomplete (or approximate) factorisation process on a systolic array with dataflow an added constraint on the type of fill-in strategy adopted. We shall concentrate on the Extended to the Limit (EL) factorisation of Lipitakis & Evans [80], a method of type a), which restricts fill-ins to certain diagonals of the matrix. In particular we develop the concept of EL architectures for producing compact systolic factorisation arrays. Clearly the success of the incomplete strategy

depends upon the structure of the coefficient matrix to be factorised.

In order to asses the compact (incomplete) systolic arrays produced we consider the system (6.1.1) with the coefficient matrix derived from the 2-D and 3-D parabolic and elliptic equations given by (2.5.1.17), (2.5.1.20), which for convenience we denote as problem I, and as problem II. These systems are not only large and sparse but have also been subject to a detailed study regarding EL factorisation in Lipitakis [78], providing a good assessment of new systolic EL type algorithms.

The EL factorisation procedure produces various approximate and exact factorisations of A in (6.1.1). Essentially, an approximate method is obtained if A is replaced by $(A+\overline{R})$ such that,

$$A + \overline{R} = L_s U_s \ , \tag{6.3.1}$$

with L_s and U_s sparse lower and upper triangular factors, whose product approximates A with error $\overline{R}$. The concept of a limit is introduced by generating a sequence of algorithms (denoted by FACTOR (i)),

$$A + \overline{R}_i = L_{s_i} U_{s_i} \Rightarrow \text{FACTOR(i)} \ , \tag{6.3.2}$$

which produce a corresponding sequence of decreasing residuals, such that,

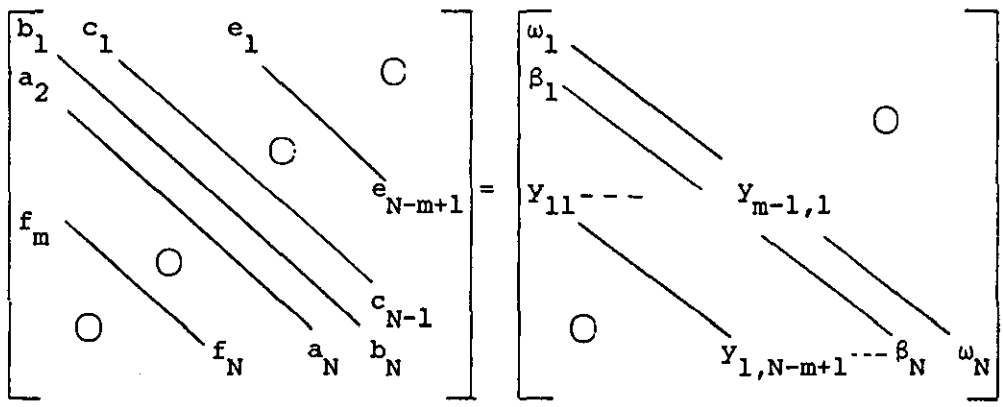
$$\lim_{k \to z} (A - L_{s_k} U_{s_k}) = \underline{O} \ , \tag{6.3.3}$$

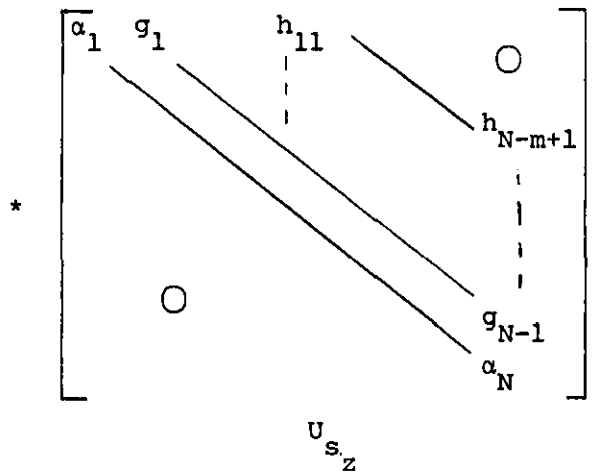
where FACTOR(z) is the complete algorithm, with $\overline{R}_z = \underline{O}$. A global algorithm which embodies a particular sequence of FACTOR(i) algorithms is the ALUBOT algorithm (see Lipitakis [78], and Lipitakis & Evans [80]), which is stated below for completeness.

```
ALUBOT(N,m,r,A){
/* Tri-diagonal Factorisation */
w_1=b_1; d_1=a_2; g_1=c_1/w_1
FOR i=2 TO m-2
   {w_i=b_i-d_{i-1}*g_{i-1}; d_i=a_{i+1}; g_i=c_i/w_i}
w_{m-1}=b_{m-1}-d_{m-2}*g_{m-2}
/* Rest of Factorisation */
FOR j=1 TO N-m+1
   {e_{1,j}=v_{j+m-1}; h_{1,j}=u_j/w_j
     g_{m+j-2}=c_{m+j-2}/w_{m+j-2}; d_{m+j-2}=a_{m+j-1}
     IF r-j+1 >=2 THEN
       {FOR i=2,r-j+1
          {e_{ij}=-g_{i+j-2}*e_{i-1,j};
           h_{ij}=-d_{i+j-2}* h_{i-1,j}/w_{i+j-1}
          }
       }
     ELSE IF NOT ((j=1) OR (r=1)) THEN
       {IF j>r THEN IP=2 ELSE IP=r-j+2;
        r1=r+1
        FOR i=IP to r
          {z=0;
           FOR k=1 TO i-1 {z=z+e_{kj}*h_{k-i+r_1, i+j-r_1};
           E_{ij}=-g_{i+j-2}*e_{i-1,j-z};
           z=0;
           FOR k=1 TO i-1 {z=z+e_{k-i+r_1,i+j-r_1}*h_{k,j}}
           h_{ij}=(-d_{i+j-2}*h_{i-1,j}-z)/w_{i+j-1}
          };
       };
     i=r; z=0
     FOR k=1 TO i {z=z+e_{kj}*h_{k,j}}
     w_{m+j-1}=b_{m+j-1}-d_{i+j-1}*h_{ij}g_{i+j-1}*e_{ij}-d_{i+j-1}*g_{i+j-1}-z
   }

REMARK    v=f, u=e, e_{ij}=y_{ij}
```

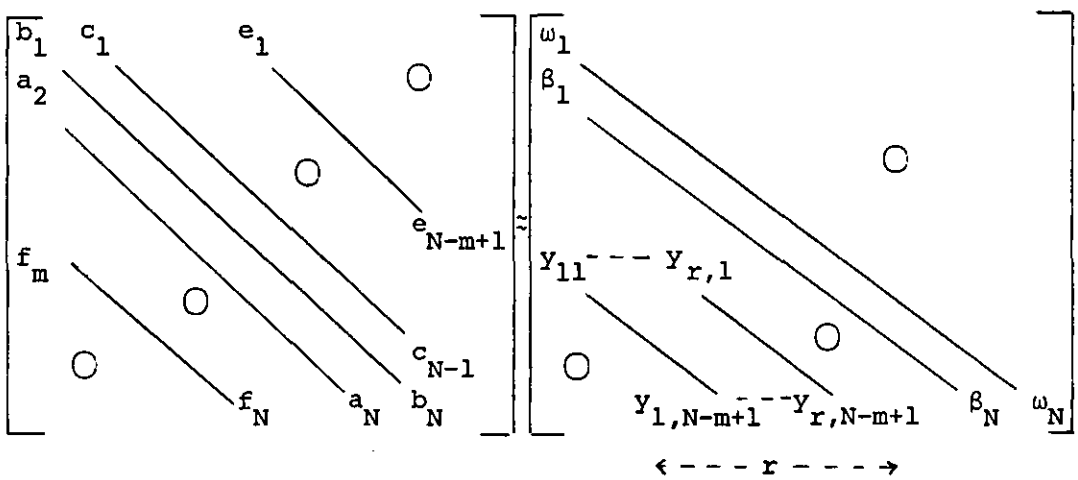
To illustrate the method a coefficient matrix A derived from problem I gives a complete factorisation of the form,

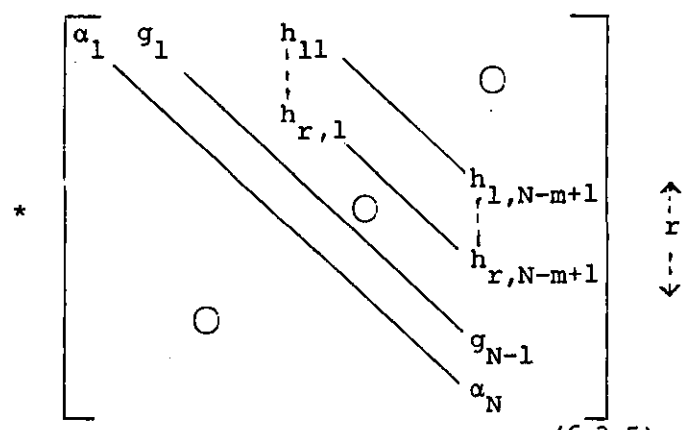




(6.3.4)

where $N = n^2$ and $m = n+1$. For the incomplete factorisation, a value r is specified which dictates the number of non-zero diagonals to be retained (for fill-in) moving left from the outermost diagonals e_i, f_i. Applying the ALUBOT produces,





(6.3.5)

and for r=1(1)m-2 the FACTOR(r) sequence defines arbitrary approximations to A. An approximate solution to (6.1.1) is then found using the coupled systems,

$$L_{s_r}\, y = d \ , \quad U_{s_r}\, u = y \ , \tag{6.3.6}$$

and by logical extension the preconditioned form (6.1.8) can be solved by repeated application of similar coupled systems. Thus for simplicity the FACTOR(i) algorithms can be encoded as the procedure calls,

```
SEQ
  ALUBOT(N,m,r,A)                                          (6.3.7)
  FBSUBS(N,m,r,L_sr,U_sr,d)
```

where the parameter r defines arbitrary approximations to the solution of (6.1.1) and is used to tradeoff storage requirements, computation time and accuracy. The method generalises easily to cases with more bands and for problem II an alternative algorithm yields,

```
SEQ
  ALUBOT-2(N,m,p,r1,r2,A)                                  (6.3.8)
  FBSUBS-2(N,m,p,r1,r2,L_sr1,r2,U_sr1,r2,d)
```

where $N=n^3$, $m=n+1$, $p=n^2+1$ and,

$$L_{s_{r_1},r_2} = \begin{bmatrix} \omega_1 & & & & \\ \beta_1 & & & & \\ y_{11} \cdots y_{r_1,1} & \ddots & & & \\ f_{11} \cdots f_{r_2,1} & & \ddots & & \\ & \ddots & & \beta_{N-1} & \omega_N \end{bmatrix} \quad U_{s_{r_1},r_2} = \begin{bmatrix} \alpha_1 & g_1 & h_{1,1} \cdots h_{r_1,1} & t_{11} \cdots t_{r_2,1} & \\ & \ddots & \ddots & \ddots & \\ & & & & g_{N-1} \\ & & & & \alpha_N \end{bmatrix} \tag{6.3.9}$$

and two parameters are available for optimising space/time/accuracy tradeoffs.

Systolic 'Extension to the Limit' (SEL) Architectures are created easily in a soft-systolic frame by replacing algorithm code in (6.3.7) and (6.3.8) by code describing the data structure of a hexagonal factorisation array. It should now be self-evident that the above incomplete factorisations admit the possibility of large cell savings, and defines a sequence of systolic architectures which trade accuracy against cell count. This view of systolic array derivation is similar to that in Thompson & Tucker [85] in which a semi-formal model of algorithm design suggests a sequence of draft designs, formalised by algorithm transformations as a result of design decisions. In the present context we view draft algorithms as systolic arrays and algorithm transformations as movements up and down the approximation sequence (6.3.2), and across soft, hybrid and hard-systolic frames, with design decisions based on the technology constraints.

A generalized incomplete array corresponding to the FACTOR(i) algorithm can be derived as a series of cell compactions as follows.

First Level Compaction:

Let w_s be the semi-bandwidth of A (for problems I and II), and w_I the interior bandwidth of the central band. From close observation of the standard hex (Fig.(3.2.2.3)) the following features are evident:

a) The matrix A is input in diagonal format

b) Accumulation of the L_s and U_s entries of a diagonal occur in cells of the same column

c) A fill-in entry for an initially empty diagonal can only occur in a cell belonging to the same column as the boundary input cell for that diagonal.

Consequently, if we prevent an initially empty diagonal from filling in

none of the cells in the associated column can produce a non-zero result. It follows that each column of cells where fill-in is prohibited can be replaced by delay cells, as depicted in Fig.(6.3.1). As delay cells consist only of registers it follows that the array must consume less area - i.e. more compact. Furthermore the delay cell columns need to pass data only SW and SE, allowing vertical connections to be removed altogether producing a sizeable reduction in the host/array interface. One further useful attribute is that increases in r (or generally r_i) minimise the number of additional hex ips cells introduced. This follows from the tapered structure of the array and the fact that fill-in diagonals are retained inwards from the outermost diagonal (column size increases as we move inward).

Now the total number of cells saved (or delay cells) is given by,

$$S = W_s^2 - C \ , \tag{6.3.10}$$

where C=number of true cells in the compacted design. For Fig.(6.3.1b)

$$C = (\text{cells in central band}) + 2*(\text{cells in retained columns})$$

$$= C_1 + 2*C_2 \ , \tag{6.3.11}$$

with

$$C_1 = W_s + 2\sum_{i=1}^{\bar{W}} (W_s - i) = W_s + 2W_s\bar{W} - \bar{W}(\bar{W}+1) \ , \tag{6.3.12}$$

when $\bar{W} = \left\lfloor \frac{W_I - 1}{2} \right\rfloor$ and,

$$C_2 = \sum_{i=1}^{r} \{(W_s - t + 1) - i + 1\} = (W_s - t + 2)r - \frac{r}{2}(r+1) \ , \tag{6.3.13}$$

where t is the column index of the first retained column (in this case t=5). Using Fig.(6.3.1c) the assessment is generalised to give,

$$C = C_1 + 2*(\text{sum of cells in retained column bands})$$

$$= C_1 + 2*C_3 \tag{6.3.14}$$

and with r_i, i=1(1)k denoting the number of retained columns in k

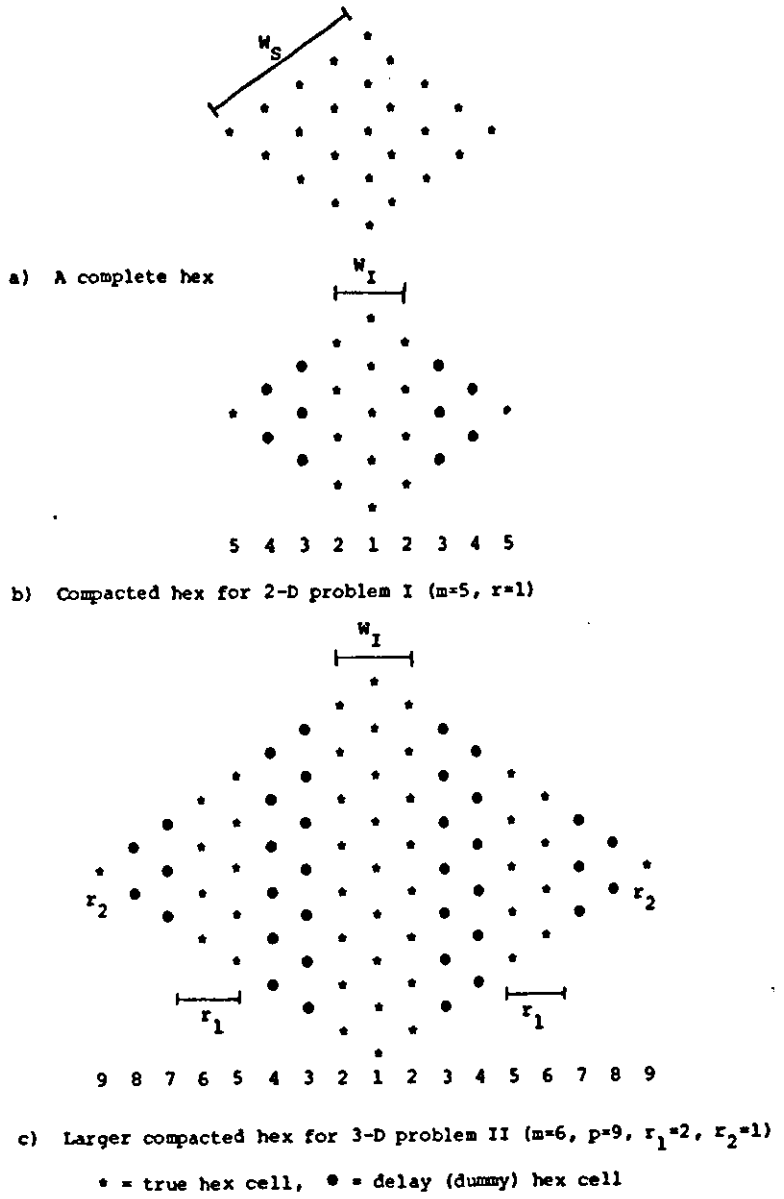


FIGURE 6.3.1: Cell replacement and compaction for EL algorithm

individual bands, and t_i, i=1(1)k the column index of the first retained column of each band from the central (or principal diagonal) column. If a single column band contributes Δr_i cells

$$C_3 = \sum_{j=1}^{k} \Delta r_j \ , \qquad (6.3.15a)$$

and,

$$\Delta r_j = \sum_{i=1}^{r_j} \{(W_s - t_j + 1) - i + 1\} = (W_s - t_j + 2) r_j - \frac{r_j}{2}(r_j + 1) \qquad (6.3.15b)$$

Hence we have,

$$C_3 = \sum_{j=1}^{k} (W_s - t_j + 2) r_j - \tfrac{1}{2} \sum_{j=1}^{k} r_j (r_j + 1) \ . \qquad (6.3.15c)$$

Thus for problem I, $W_s=n+1$, $W_I=3$, $t=n-2$, $r=4$,

$$C_1 = (n+1)+2(n+1-1) = 3n+1 ,$$

$$C_2 = ((n+1)-(n-2)+2)4-\frac{4}{2}(5) = 5*4-2*5 = 10$$

$$C = C_1+2*C_2 = 3n+1+20 = 3n+21 ,$$

producing the saving,

$$S = (n+1)^2-3n-21 = n^2-n-20 , \qquad (6.3.16)$$

over the complete array. For problem II $W_s=n^2+1$, $W_I=3$, $t_1=n-2$, $r_1=4$, $t_2=n^2-2$, $r_2=4$,

$$C_1 = (n^2+1)+2n^2 = 3n^2+1$$

$$C_3 = \Delta r_1+\Delta r_2$$

$$\Delta r_1 = ((n^2+1)-(n-2)+2)4-2(5) = (n^2-n+5)4-10$$

$$\Delta r_2 = ((n^2+1)-(n^2-2)+2)4-10 = 10$$

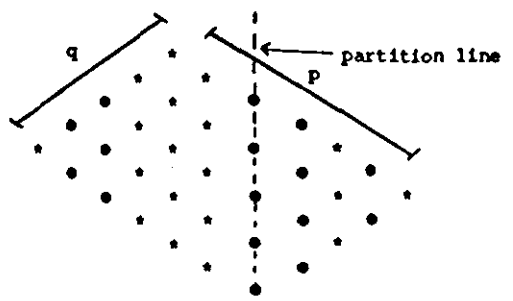
$$C = 11n^2-8n+21 ,$$

yielding a saving of,

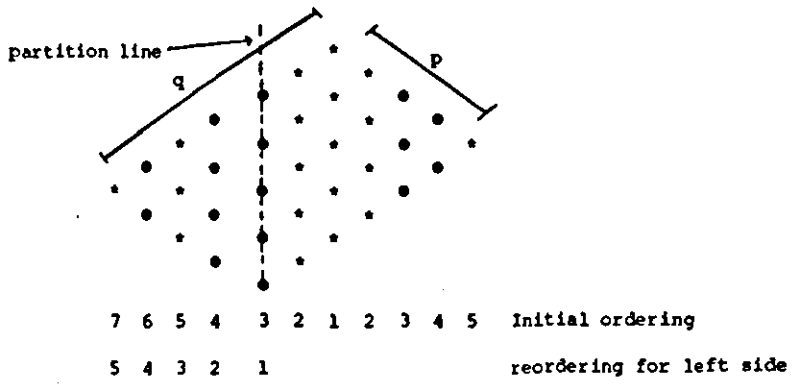
$$S = n^4-9n^2+8n-20 , \qquad (6.3.17)$$

over the cells required for the complete array.

The principle is easily extended to matrices in which the band structure is not symmetric (as opposed to symmetric elements), for example in Fig.(6.3.2) where W=p+q-1.



a) Hex skewed right



b) Hex skewed left

FIGURE 6.3.2: Non-symmetric array compaction

Now, C = (cells in bands from left of array)+(cells in central band)+ (cells in bands from right array)

$$= C_1 + C_2 + C_3 \ . \qquad (6.3.18)$$

Putting W_s=min(p,q) gives,

$$C_2 = 2W_s + \sum_{i=1}^{\overline{W}} (W_s - i) = 2W_s + W_s\overline{W} - \tfrac{1}{2}\overline{W}(\overline{W}+1) \qquad (6.3.19)$$

$$C_1 = \sum_{j=1}^{k_o} \Delta r_j = \sum_{j=1}^{k_o} (W_s - t_j + 2) r_j - \tfrac{1}{2} \sum_{j=1}^{k_o} r_j (r_j + 1) \ , \qquad (6.3.20)$$

for Fig.(6.3.2a) with k_o the number of bands on the left side. To compute C_3 a partition line is added such that columns to the right of the line are strictly descending in cell count, and cells to left between the partition and centre column contain W_s cells each.

C_3 = (total cells left of partition)+(total cell right of partition)

$$= C_4 + C_5 \ , \qquad (6.3.21)$$

where,

$$C_4 = \sum_{i=1}^{k_1} \Delta r_i \quad \text{and} \quad \Delta r_j = r_j W_s \ . \qquad (6.3.22)$$

Now renumber the columns to the right of the partition with the column

coincident with the line having index 1, and adjust the starting places from t_j to $\bar{t}_j$ and modify number column r_j accordingly. Then,

$$C_5 = \sum_{i=1}^{k_2} \Delta\bar{r}_i = \sum_{j=1}^{k_2} (W_s - \bar{t}_j + 2)\bar{r}_j - \tfrac{1}{2} \sum_{j=1}^{k_2} \bar{r}_j(\bar{r}_j + 1) \qquad (6.3.23)$$

where k_1+k_2=number of retained bands in right-hand partition of array. The formula for Fig.(6.3.2b) is derived similarly with the partition line on the left.

First level compaction shows that large area savings can be made when the number of retained diagonals is small compared with n. In the sample problems we choose r_i=4. The analysis in Lipitakis [78] suggests that a large error reduction occurs for $r_i \lesssim 4$, and that keeping more bands for $r_i > 4$ does not significantly improve this initial reduction in the approximation error further. Hence the results of (6.3.16) and (6.3.17) are quite realistic.

Second Level Compaction

If the delay cells introduced in the first level compaction are termed primary neutral cells, second level compaction identifies secondary neutral cells arising from dataflow side-effects due to the creation of primary cells. To identify these secondary cells we trace out the path of an element input to a column of primary cells, which in general consists of two stages.

Stage 1: A vertical movement to the upper hex boundary. (It is clear from first level compaction that only zero values make this journey).

Stage 2: On the hex upper boundary, cells to the left of centre compute multipliers, and cells on the right perform a simple negate (see Fig. (3.2.2.3)) and the results are reflected to travel SE and SW respectively.

Results of cells at the top of a primary cell column must be zero, hence any cells on the SE or SW paths never modify values moving vertically.

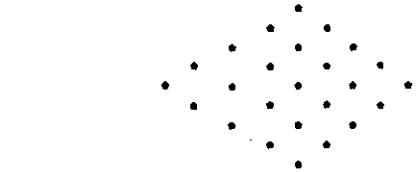
Figs.(6.3.3) and (6.3.4) illustrate the compaction of the hex array for problem I and II, the cuts will be explained shortly. As no modification occurs in secondary neutral cells, like primary cells they can be replaced by delay cells. It is then trivial to deduce that the number of true hex cells is given by $(\lceil \frac{W_I}{2} \rceil + r - 1)^2$ for problem I, and more generally $(\lceil \frac{W_I}{2} \rceil + \sum_{i=1}^{k} r_k - 1)^2$ for k bands of widths r_i, i=1(1)k respectively. More intuitively the number of true hex cells is proportional to the number of non-zero diagonals of A plus any diagonals retained for fill-in.

Third Level Compaction:

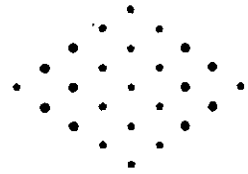
The first and second level compactions modify area but leave array computation time unchanged. In a normal numerical algorithm the decrease in calculations for incomplete compared with complete factorisations, would be expected to achieve a corresponding decrease in algorithm computation time. This attribute is transferred to our incomplete array using a third level compaction, which identifies a series of cuts to reduce array latency. A cut is identified by marking out regions of primary and secondary cells which partition the array into disjoint regions of three types:

1. A region consisting only of true hex cells
2. A region consisting only of neutral cells
3. A region with both true hex and neutral cells.

Compaction is then achieved by removing all type 2. regions, to leave tesselating disjoint regions which when compressed form a smaller hex array.

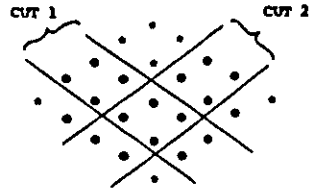


a) Standard hex array

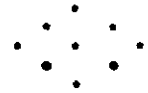


b) Non-retained diagonal cells replaced (by delay)

c) Identify secondary neutral cells (replace with delays)

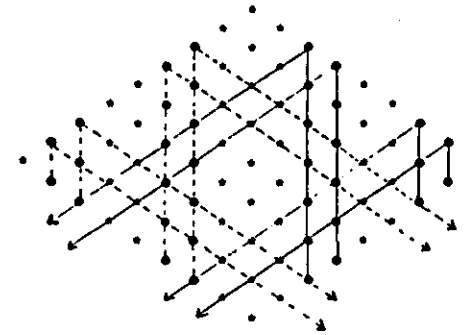


d) Define a synchronisation cuts (and remove delays)

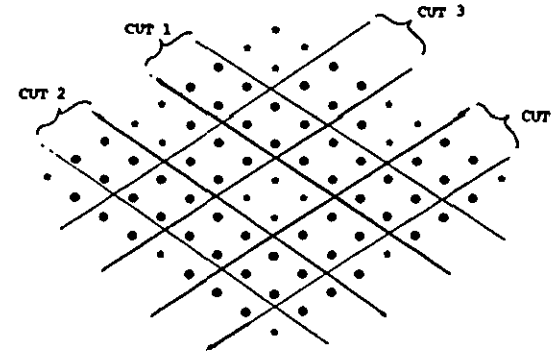


e) Compact hex array for approximate factorisation

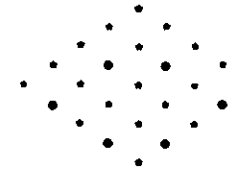
FIGURE 6.3.3: Construction of incomplete factorisation array



a) Identify secondary neutral cells



b) Make synchronisation cuts



c) Compacted array

REMARK: The original hex required 9x9=81 hex cells,
Compacted array 5x5=25 hex cells a saving of 56 cells
which is almost 75% area reduction.

FIGURE 6.3.4: Incomplete array construction for Fig.6.3.1c

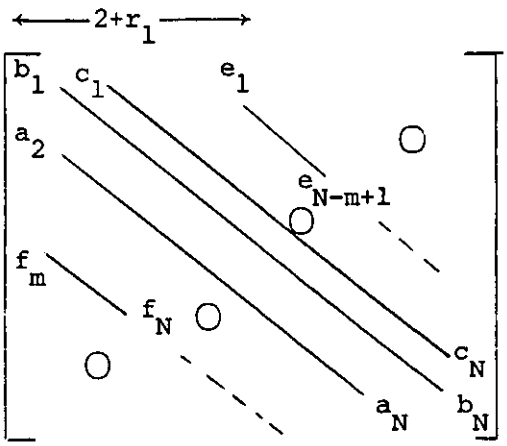
This array has W^2 cells where W is the sum of the semi-bandwidth and retained (sub)-super diagonals for fill-in. In fact after third level compaction we can generally expect a saving of over 75% of the original number of hex cells. However making cuts and removing cells in this way affects the factorisation process by destroying synchronisation between the central band and retained outer diagonals. It follows that third level cuts are weak variants of cuts used in H.T. Kung & Lam [84], which due to the approximate nature of incomplete calculations do not demand that dataflow must be retimed as delays are removed.

Now in order to utilise the cut array we must answer two questions:

1. What kind of factorisation does the cut hex compute?
2. Is this factorisation a good incomplete method for the given problem?

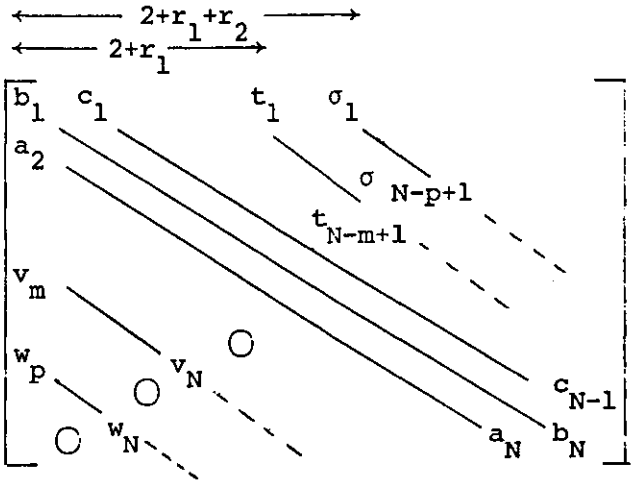
Simple observation on the cut hex dataflow answers 1. immediately

For the model problems A in (6.1.1) becomes transformed as follows:

A = 

for problem I (6.3.24)

and

A = 

for problem II (6.3.25)

But factorising these matrices and solving the associated coupled systems produces a solution to a totally different system to the original one, consequently we would expect poor approximate solutions. A further complication arises when we notice that the shifting of outer diagonals inwards creates unknown entries (dashed lines in (6.3.24) and (6.3.25)). The choice of these elements could be crucial to a good approximate factorisation.

Careful consideration of the incomplete method described by the ALUBOT algorithm allows the derivation of a new algorithm, the SYSTOLIC INCOMPLETE FACTORISATION (SIF) method. Here the incomplete method is performed as a two step process.

(i) Form a new matrix $\bar{A}$ from A by shifting outer diagonals inwards to remove diagonals not retained for fill-in (i.e. (6.3.24) and (6.3.25)). Factorise using the cut hex to produce L_s and U_s.

(ii) Derive two matrices $\bar{L}_s$ and $\bar{U}_s$ from L_s and U_s by shifting retained diagonals outwards, and inserting zero diagonals to recover the original sparsity pattern of (6.3.5) and (6.3.9) and solve the new coupled systems corresponding to (6.3.6).

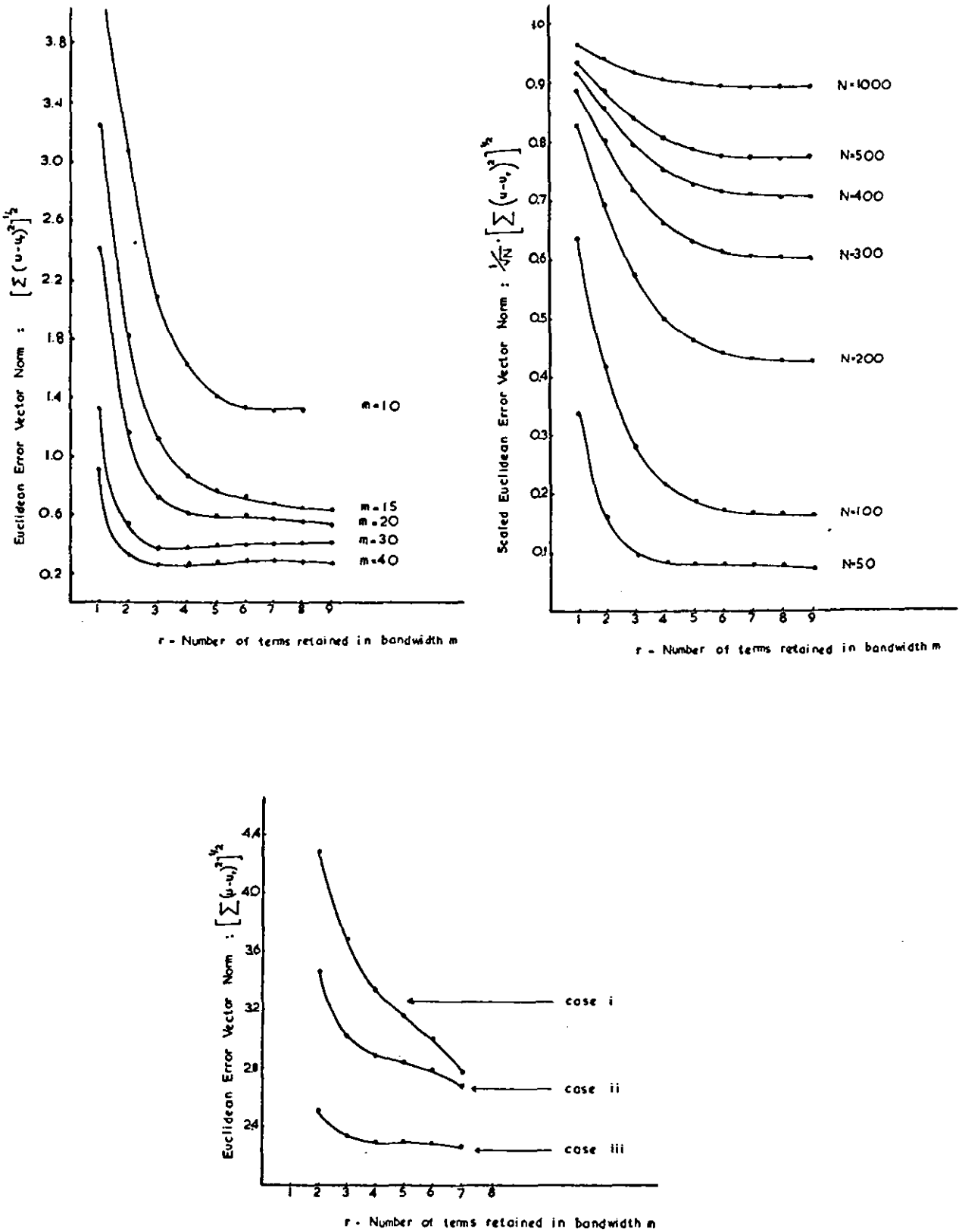


FIGURE 6.3.5: Approximate factorisation results (Lipitakis [78])

and can be summarized as,

SEQ
  ALUBOT(N,r+2,r,$\bar{A}$) (6.3.26)
  FBSUBS(N,r+2,r,$\bar{L}_{s_r}$,$\bar{U}_{s_r}$,d) .

The extension to problem II is trivial and yields,

SEQ
  ALUBOT-2(N,r_1+2,r_1+r_2+2,r_1,r_2,$\bar{A}$) (6.3.27)
  FBSUBS-2(N,r_1+2,r_1+r_2+2,r_1,r_2,$\bar{L}_{s_{r_1,r_2}}$,$\bar{U}_{s_{r_1,r_2}}$,d) .

The essential idea behind the SIF is to compensate for the initial shift in of diagonals (so the cut hex can be employed) by shift out in between factorisation and solution. This has two advantages, firstly the coupled systems solve a problem with similar structure to the original, second the fill-ins associated with the newly created elements are moved out of the matrix to reduce their effect on the final approximate solution. The method was tested for various values of m, with N fixed, and with m fixed and N varied, and the number of retained diagonals r plotted against the Euclidean error vector norm. Tests were carried out for problem I with $d=(1,1,\ldots,1)^t$ and repeated with $u=(1,1,\ldots,1)^t$ with d_i, i=1(1)N simply the sum of the coefficient from row i of A. The results are shown in Graphs (A1)-(A6) of Fig.(6.3.6) and can be compared with the approximate method results of Lipitakis [78] in Fig.(6.3.5). The label EXT-1 and EXTO indicates that values created by the shifting in of diagonals were set to -1 or 0 respectively.

The results indicate that the cut hex with the shifted solution process retains the desirable property of rapid approximation, error reduction for the retention of the first few fill-in diagonals. Results were not as good as the EL algorithm, and we have to consider retaining r=7 diagonals rather than r=4. The cases for M=40 and N=50

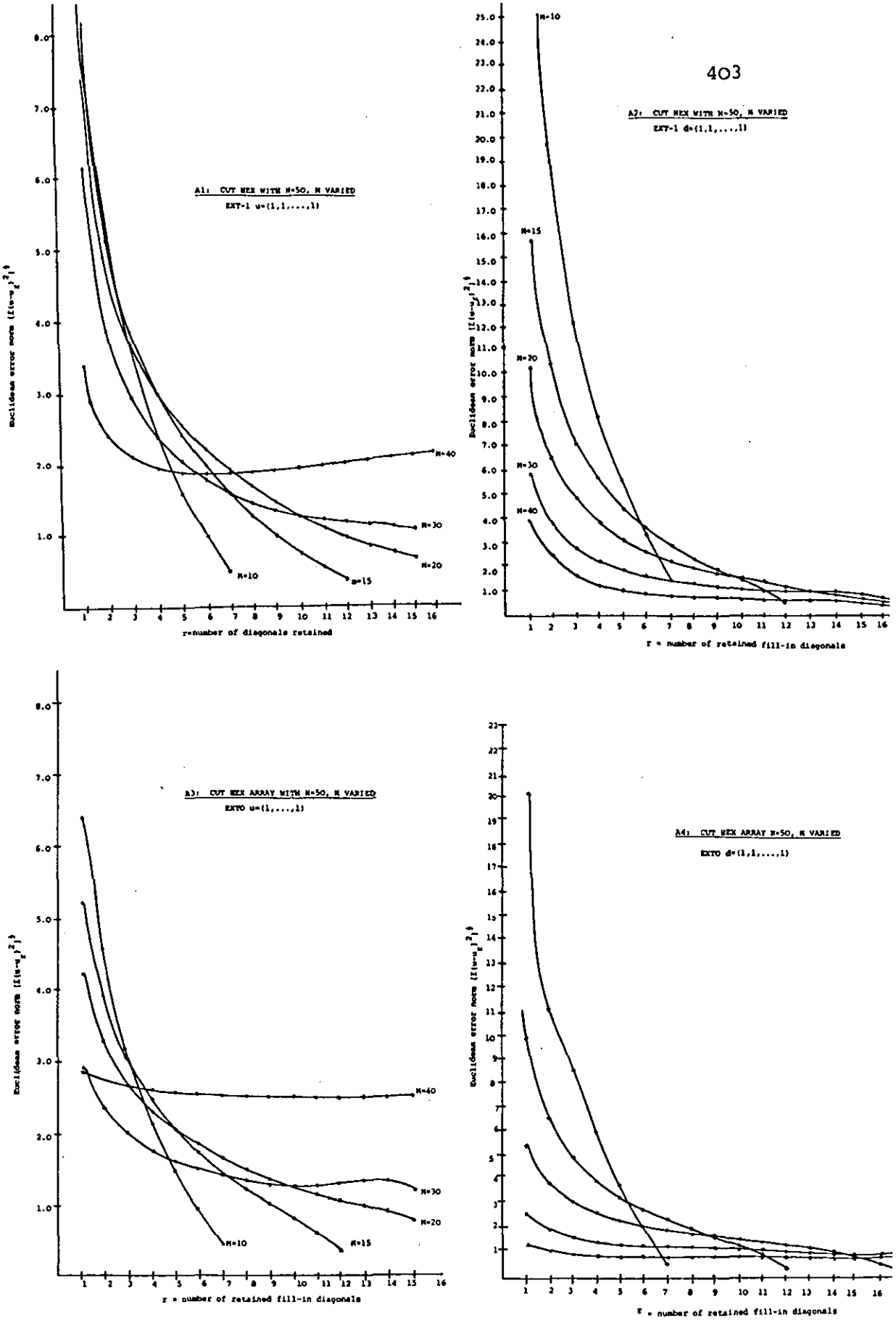
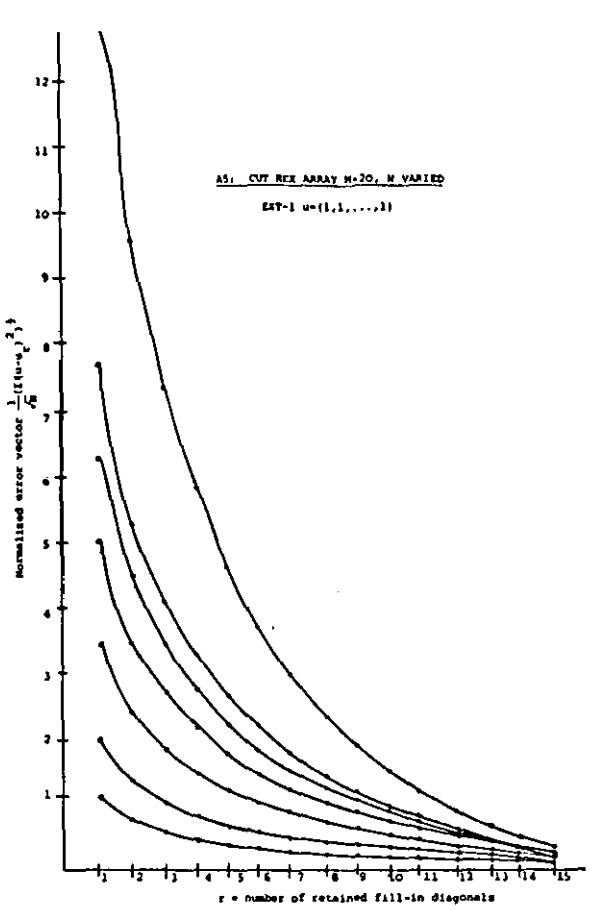
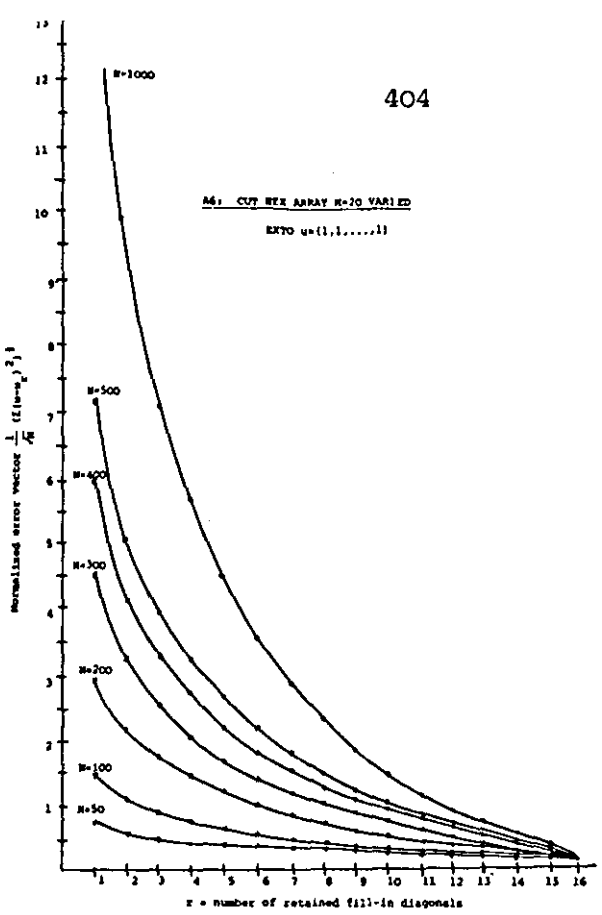
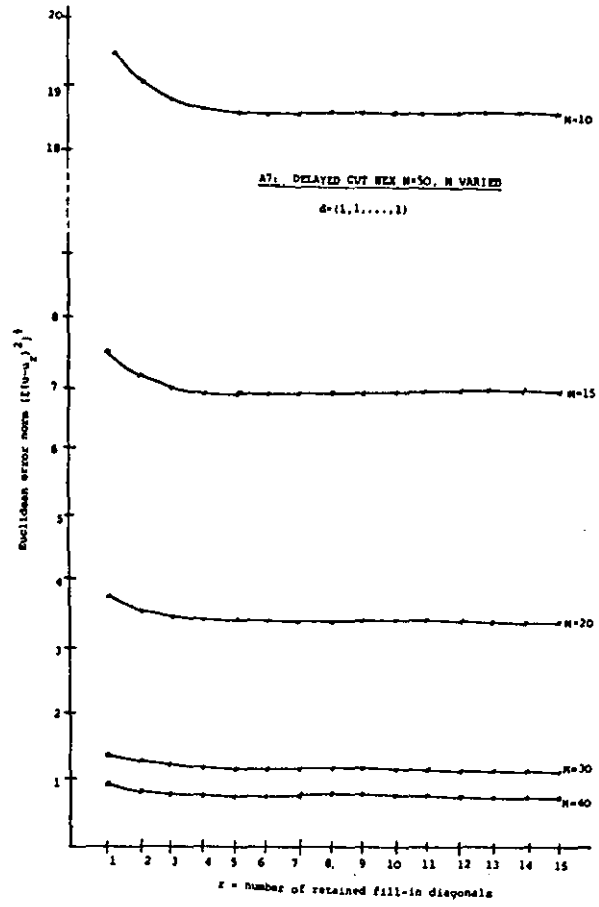


FIGURE 6.3.6: Cut hex experiments







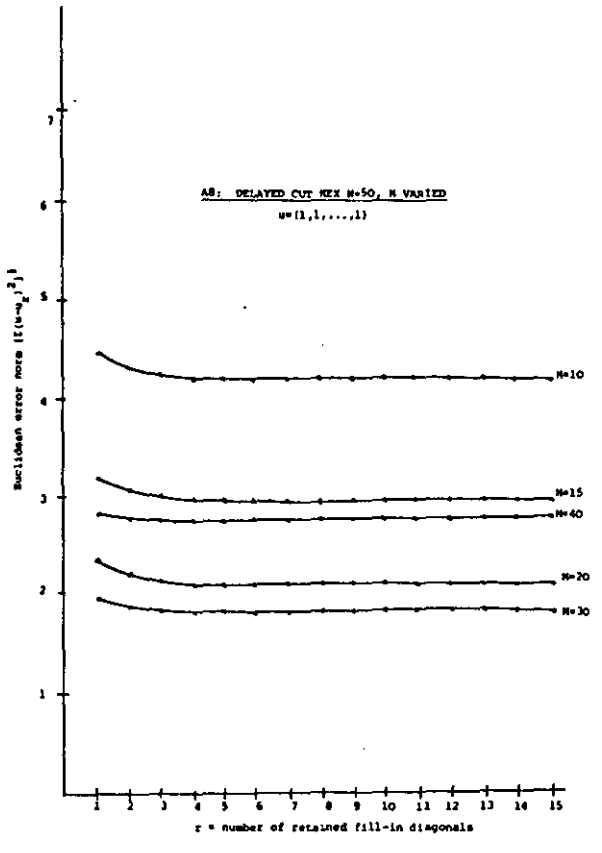
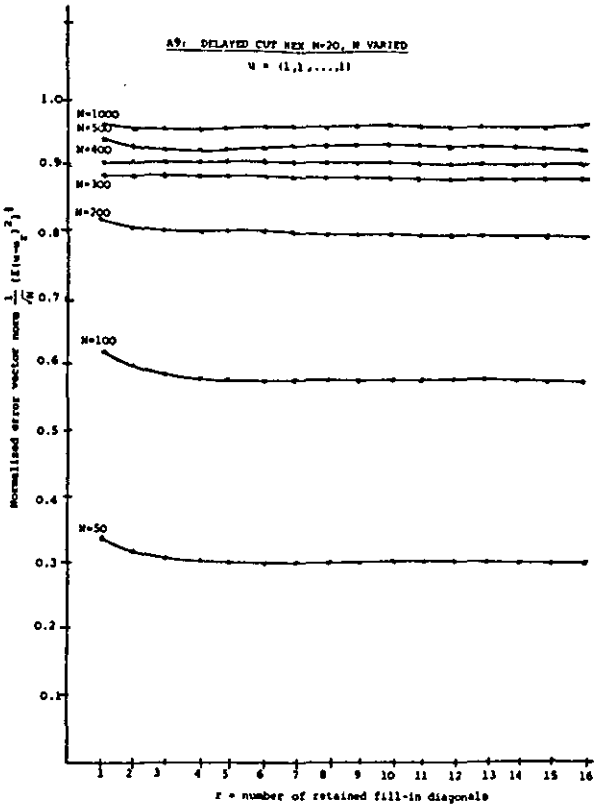
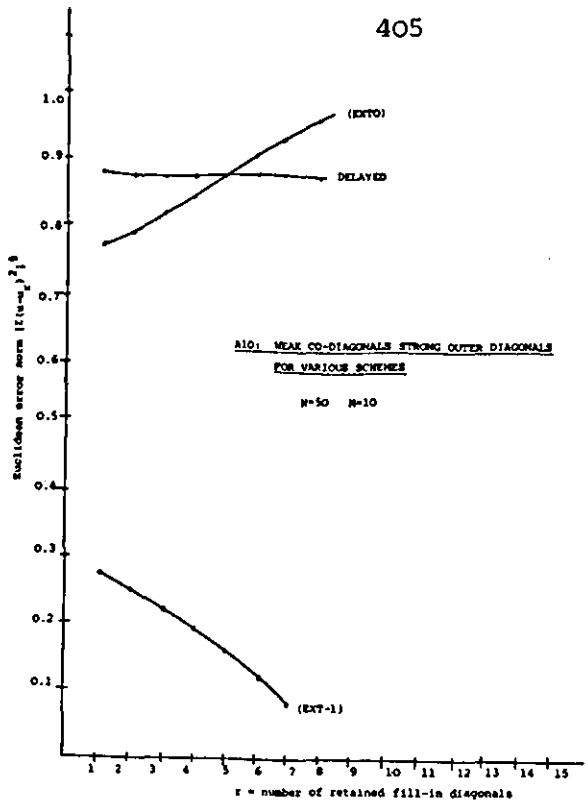
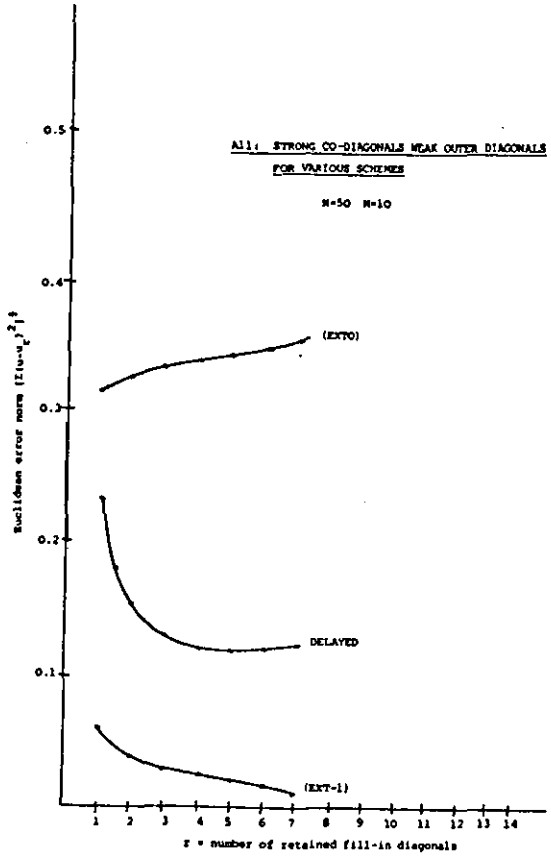


FIGURE 6.3.6(cont.)







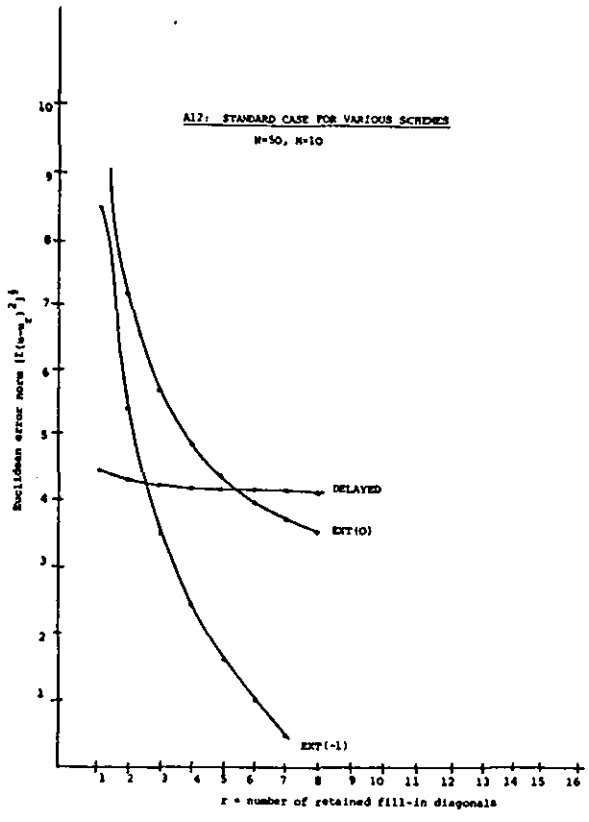


FIGURE 6.3.6(cont.)

indicate that the approximation may grow as the bandwidth relative to N increases. Further tests with large N and m fixed indicated that if the error did diverge, it did so at a much slower rate than the initial error decrease associated with a small number of retained bands. In fact by including more diagonals the error divergence rate became slower, with N=200, m=150, for example 10 extra diagonals were required to alter the error norm by 0.1. Additional experiments also showed that the divergence became observable generally when $m \geq N/2$, and can be explained by the interference caused by the distance that the diagonals must be moved in order to use a cut hex. As problems I and II always have $m<N/2$ no problems should be encountered with the SIF method.

In contrast to the SIF algorithm an alternative factorisation which incorporated some data re-timing and cut hex modifications was considered. This new method, the delayed cut hex algorithm, is given below.

b) Delayed cut hex algorithm

DCHEX (N,m,r,A)
/* Tri-diagonal Factorisation*/
$w_1=b_1$; $d_1=a_2$; $g_1=c_1/w_1$;
FOR i=2 TO N-1
  {$w_i=b_i-d_{i-1}*g_{i-1}$; $d_i=a_{i+1}$; $g_i=c_i/w_i$}
$w_N=b_N-d_{N-1}*g_{N-1}$
FOR j=1 TO N-m+1
  {$e_{1j}=v_{j+m-1}$; $h_{1,j}=u_j/w_j$;
   FOR k=2 TO r+1
     {$h_{k,j}=-h_{k-1,j}*d_{k+j-2}/w_{k+j-1}$;
      $e_{k,j}=-e_{k-1,j}*g_{k+j-2}$
     }
  }

The essential idea is to prevent fill-in entries from modifying the central band elements by a re-synchronising column of delay cells placed in between adjacent columns of cut hex cells which belong to different fill-in bands of problem I and II.

Thus central band elements are factorised as a tridiagonal form, and fill-in diagonals are computed using the multipliers generated for each row. The basis for this idea is that the multipliers of problems I and II have the same order of magnitude and so may produce a good approximation. The results in Fig.(6.3.6) graphs (A7)-(A9) indicate that virtually the same error of approximation occurs, irrespective of the number of retained diagonals. However if we get an initially large error, there is no chance of decreasing it by retaining more diagonals.

Finally, the effect of the element sizes on the codiagonal and outer diagonals on the error was considered for three cases, i.e.,

(i) Codiagonals strong, outer diagonals weak

(ii) Both co- and outer diagonals of the same order

(iii) Codiagonals weak, outer diagonals strong.

These results are shown in Fig.(6.3.6) (A10-A12), where we can observe that divergence can occur with the cut hex when EXTO, while with EXT-1 errors are always reduced, and the delayed cut hex maintains the same initial error. The results indicate that the SIF algorithm with EXT-1 should be adopted when cases (i) and (iii) occur, and EXTO for case (ii). A more general form of the SIF method then follows easily by selecting EXTα with α chosen according to some averaging (e.g. min, max, etc.) criterion.

Now although the SIF algorithm gives higher approximation errors than the EL method the reduction in cell count clearly compensates as Table (6.3.1) shows.

			SIF PROBLEM I		SIF PROBLEM II		COMPLETE LU	
n	m	p	r=7	r=4	$r_1=r_2=7$	$r_1=r_2=4$	I	II
10	11	101	81	36	256	100	121	10201
20	21	401	"	"	"	"	441	160801
30	31	901	"	"	"	"	961	811801
40	41	1601	"	"	"	"	1681	2.5×10^6
50	51	2501	"	"	"	"	2601	6.2×10^6
100	101	10001	"	"	"	"	10201	1.0×10^8

TABLE 6.3.1: Cell requirements for cut hex(SIF) and complete factorisation hex arrays

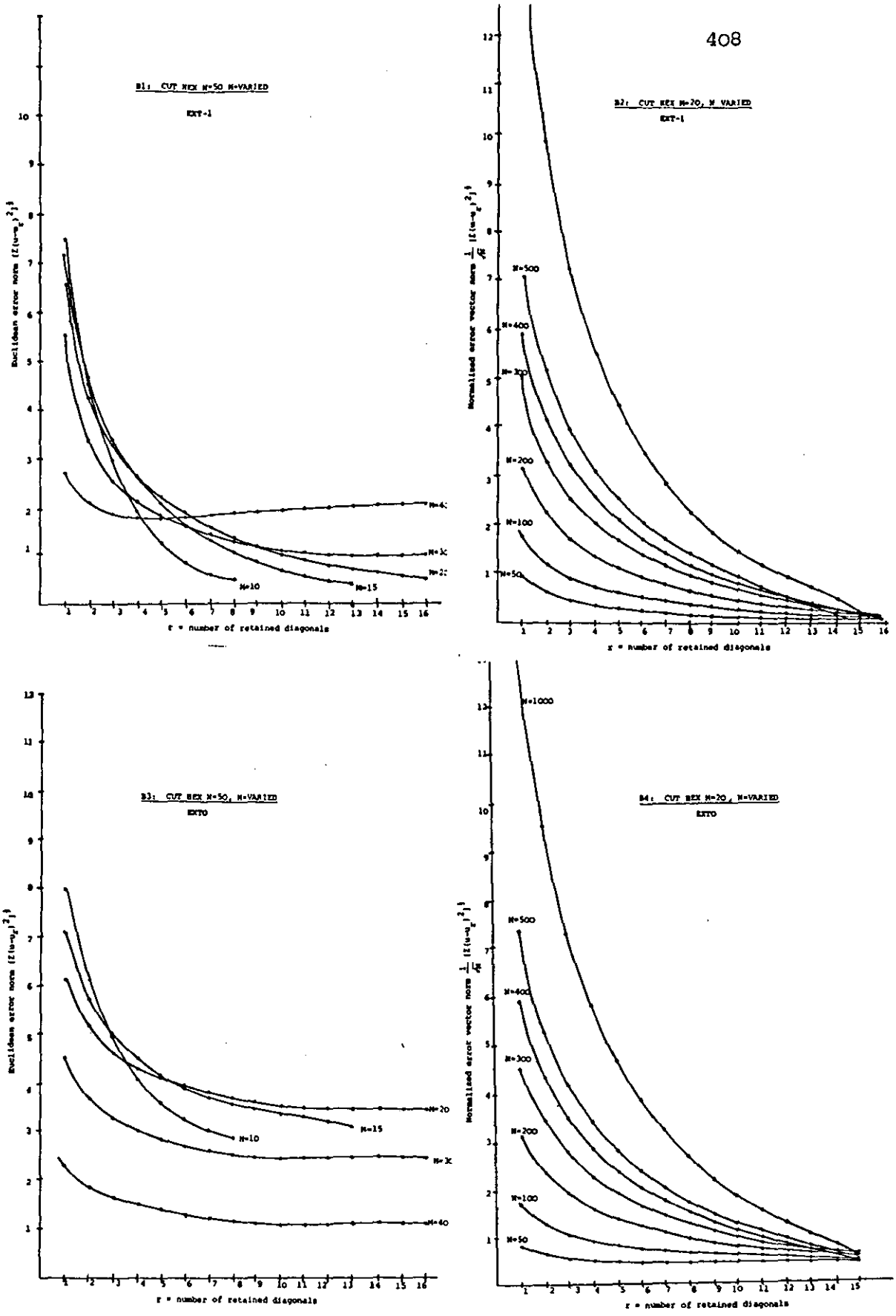


FIGURE 6.3.7: Experiments for double pipe backsubstitution

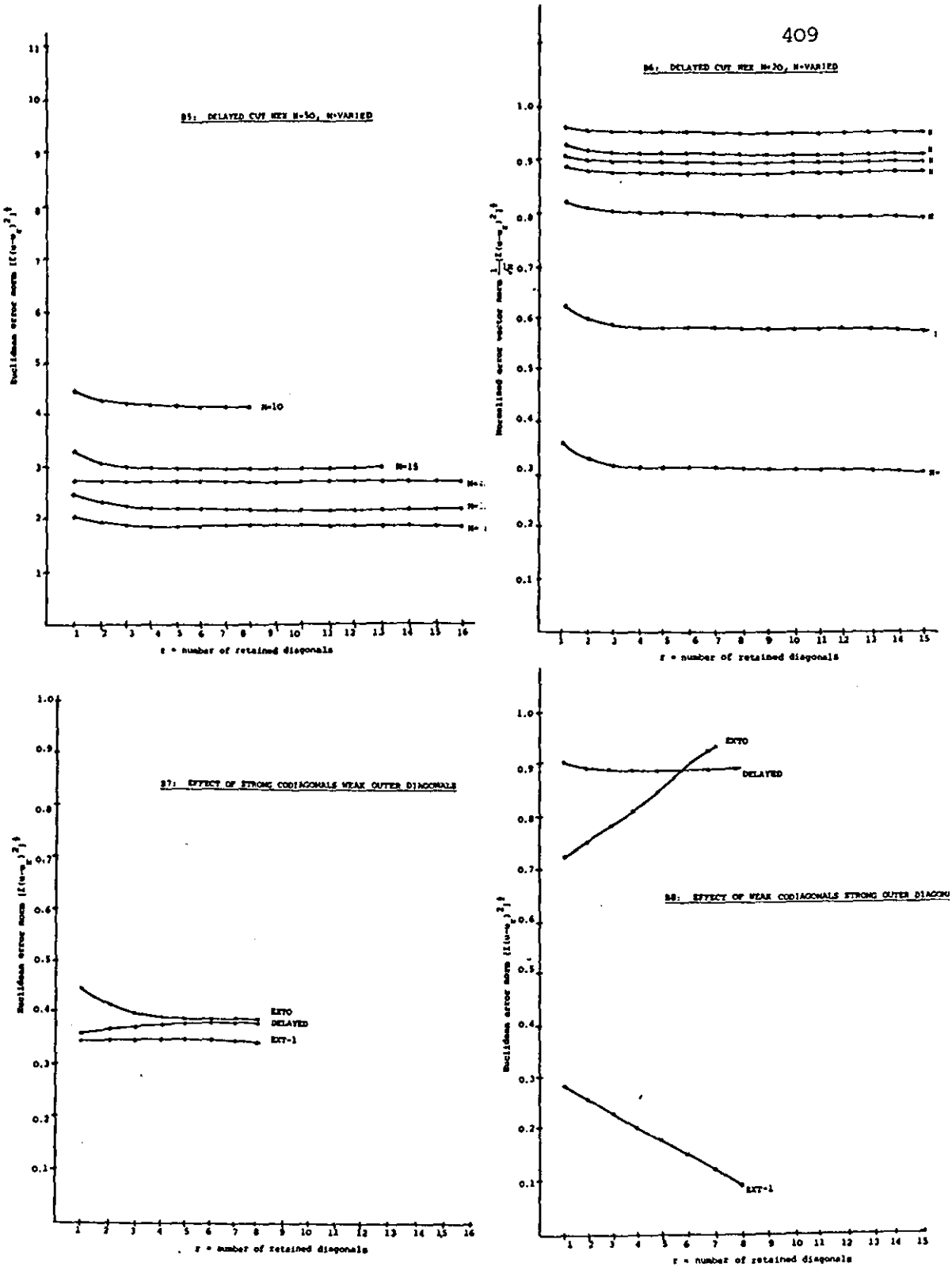


FIGURE 6.3.7(cont.)

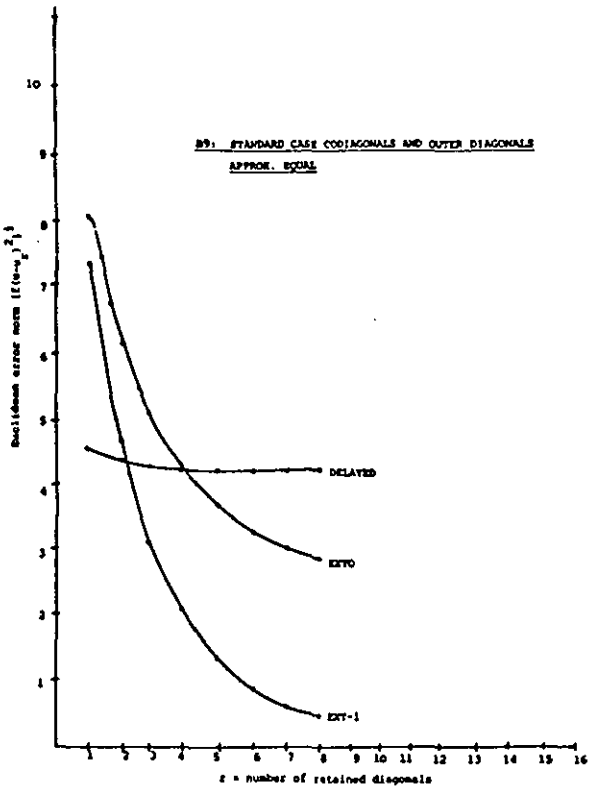
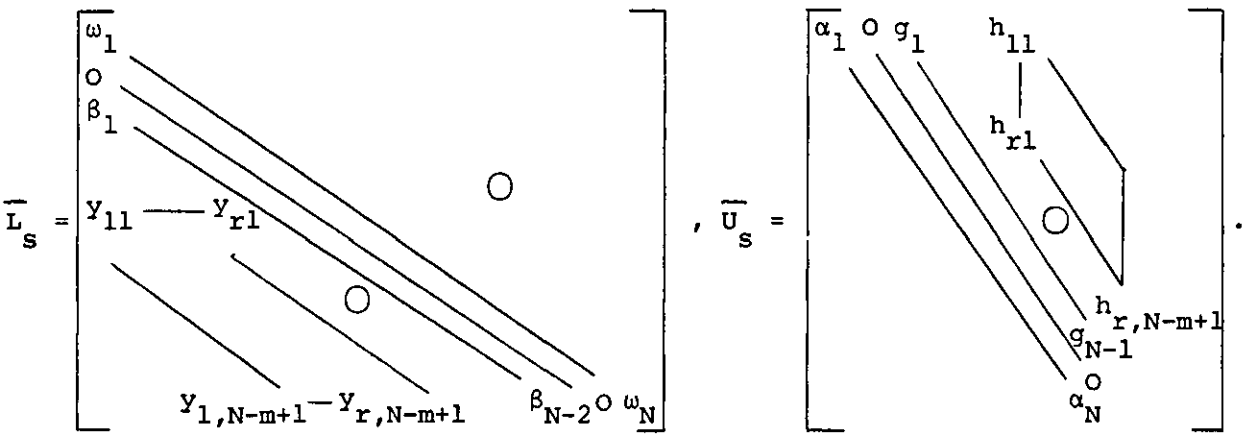


FIGURE 6.3.7(cont.)

Indeed it would be impractical to build some of the large arrays but extremely attractive to construct a fixed sized SIF hex. A cut hex of 256 hexes could solve both problems I and II by allowing r=7 for I and $r_1=r_2=7$ for II, giving a fixed sized array and variable accuracy for a range of bandwidths.

It should also be clear that array operations can be accelerated by applying double pipe and block partitionings of the previous chapters. In particular the double pipe substitution array of Robert & Tcheunte [84] (Fig.(4.2.1)) can be applied directly with a simple modification to the production of $\tilde{L}_s$ and $\tilde{U}_s$ to yield,



The results are shown in graphs (B1)-(B9) of Fig.(6.3.7), from which we observe that EXT-1 reduces the error by a small amount, EXTO increases the error and the delayed cut hex has approximately the same results, as the ordinary SIF method. Thus adopting the double pipe substitution with EXT-1 actually reduces the approximation error further, while the effect on timing is as follows,

$$T = (3N+W_s)+2*(2N+q)$$

using the ordinary single pipe solver and no pipelining between hex and solver arrays and where,

$$N = \begin{cases} n^2 & \text{problem I,} \\ n^3 & \text{problem II,} \end{cases} \qquad q = \begin{cases} n+1 \text{ , problem I} \\ n^2+1, \text{ problem II} \end{cases}$$

and

$$W_s = \begin{cases} n+1 & \text{complete} \\ 2+r & \text{incomplete} \end{cases} \Big\} \text{ problem I}$$

$$\qquad\quad \begin{cases} n^2+1 & \text{complete} \\ 2+r_1+r_2 & \text{incomplete} \end{cases} \Big\} \text{ problem II}$$

and with double pipe solvers,

$$T = (3N+W_s)+2(N+\lceil q/2 \rceil+1) \ .$$

Thus we conclude that an incomplete factorisation reduces the area by >75% and also reduces the computation time significantly when double pipe solvers are applied.

6.4 SYSTOLIC ARRAYS FOR INCOMPLETE ELIMINATIONS

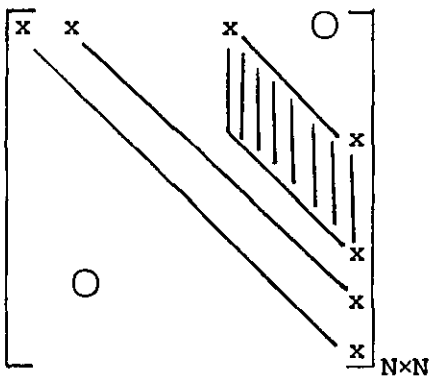
The above compaction technique is easily extended to produce a third type of systolic preconditioning preprocessor based on incomplete elimination methods.

To illustrate this technique we shall consider solving (6.1.1) with A the coefficient matrix associated with problem I of Section (6.3). An exact elimination yields,

$$Uu = \bar{d} \ , \tag{6.4.1}$$

where U is an upper triangular matrix of the form,

$$U = \tag{6.4.2}$$



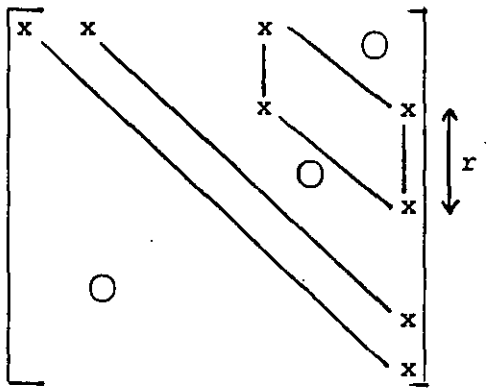
with x=non-zero, the shaded area marking the fill-in zone, and $\bar{d}$ the modified right hand side of (6.1.1). An incomplete elimination scheme retains the sparsity of the original system by selecting a number of diagonals to fill-in and leads to,

$$L_S A \simeq U_S \ ,$$

or

$$U_S u \simeq L_S d \ , \qquad (6.4.3)$$

with,

$U_S =$ 

(6.4.4)

which can be performed as follows:

```
GAUSS(N,p,r,A,f){
g_1=c_1/b_1; h_1,p=e_1/b_1; f_1=d_1/b_1;
G_p,1=v_p; D_p,1=b_p; g_p=c_p; F_p,1=d_p;
FOR i=2 TO p-1
 {
  w_i=b_i-a_i g_i-1; g_i=c_i/w_i; f_i=(d_i-a_i f_i+1)/w_i;
  FOR k=i+p-r TO i+p-2{h_i,k=-a_i h_i-1,k/w_i};
  h_i,p-i+1=e_i/w_i;
  G_p,i=-g_i-1 G_p,i-1; D_p,i=D_p,i-1-G_p,i-1 h_i-1,p;
  F_p,i=F_p,i-1-G_p,i-1 f_i-1; g_p=g_p-G_p,i-1 h_i-1,p+1;
  FOR k=p-r TO i{h_p,p+k=h_p,p+k-G_p,i-1 h_i-1,p+k};
  g_p-1=g_p-1+h_p-1,p; h_p-1,p=0;
};
D_p,p=D_p,p-1-(G_p,p-1+a_p)(g_p-1+h_p-1,p);
f_p=[f_p,p-1-(G_p,p-1+a_p)f_p-1]/D_pp;
g_p=[g_p-(G_p,p-1+a_p)h_p-1,p+1]/D_pp;
FOR k=p-r+1 TO p-1{h_p,p+k-1=[h_p,p+k-1-(G_p,p-1+a_p)h_p-1,p+k-1]/D_p,p}
h_p,2p-1=e_p/D_p,p;
FOR i=p+1 TO N
 {G_i,i-p+1=v_i; D_i,i-p+1=b_i; F_i,i-p+1=d_i; G_i=c_i;
  FOR j=i-p+2 TO i-1
    {
     G_ij=G_ij-(g_j-1+h_j-1,j)G_i,j-1;
     a_i=a_i-G_i,j-1 h_j-1,i-1;
     F_i,j=F_i,j-1-G_i,j-1 f_j-1; g_i=g_i-G_i,j-1 h_j-1,i+1;
     D_ij=D_i,j-1-G_i,j-1 g_j-1,i;
     FOR k=j+1 TO i-2{G_i,k=G_i,k-G_i,j-1 h_j-1,k};
     FOR k=p-r TO p-1{h_i,i+k=h_i,i+k -G_i,j-1 h_j-1,i+k};
    }
```

$D_{ii}=D_{i,i-1}-(G_{i,i-1}+a_i)(g_{i-1,}+h_{i-1,i})$;
$f_i=[F_{i,i-1}-(G_{i,i-1}+a_i)f_{i-1}]/D_{ii}$;
$g_i=[g_i-(G_{i,i-1}+a_i)h_{i-1,i+1}]/D_{ii}$;
FOR k=p-r+1 TO p-1
$\{h_{i,i+k-1}=[h_{i,i+k-1}-(G_{i,i-1}+a_i)h_{i-1,i+k-1}]/D_{ii}\}$;
$h_{i,i+p-1}=e_i/D_{ii}]$

Taken from Evans [83c],

where v=f in (6.3.4) and U_s is normalised with H_{ij} the elements in the shaded part of (6.4.2) and g the co-diagonal elements, which plays the same role as the ALUBOT algorithm for factorisation but produces a sequence of approximate elimination algorithms satisfying

$$\lim_{k\to z} (A-L_{s_k}^{-1}U_{s_k}) = 0 , \tag{6.4.5}$$

and yielding the algorithmic specification,

```
SEQ
  GAUSS(N,m,r,A,d)                                   (6.4.6)
  SUB(N,m,r,U_s,d-bar)
```

with SUB a simple backward substitution routine modified to take account of the sparsity in U_s. To derive an incomplete array we must first find a systolic architecture which can be substituted soft-systolically for the above numerical procedure. Arrays for triangularising a matrix are discussed by Gentleman & H.T. Kung [81] and provide a basis for our discussions. A triangular array for reduction of a full matrix and which incorporates nearest neighbour pivoting is given in Fig. (3.2.2.2). An alternative array based on orthogonal reduction using Sameh & Kuck's elimination scheme,

```
*
7  *
6  8  *
5  7  9  *                                           (6.4.7)
4  6  8  10 *
3  5  7  9  11 *
2  4  6  8  10 12 *
1  3  5  7  9  11 13 *
```

and suited to banded matrices is given in Fig.(6.4.1).

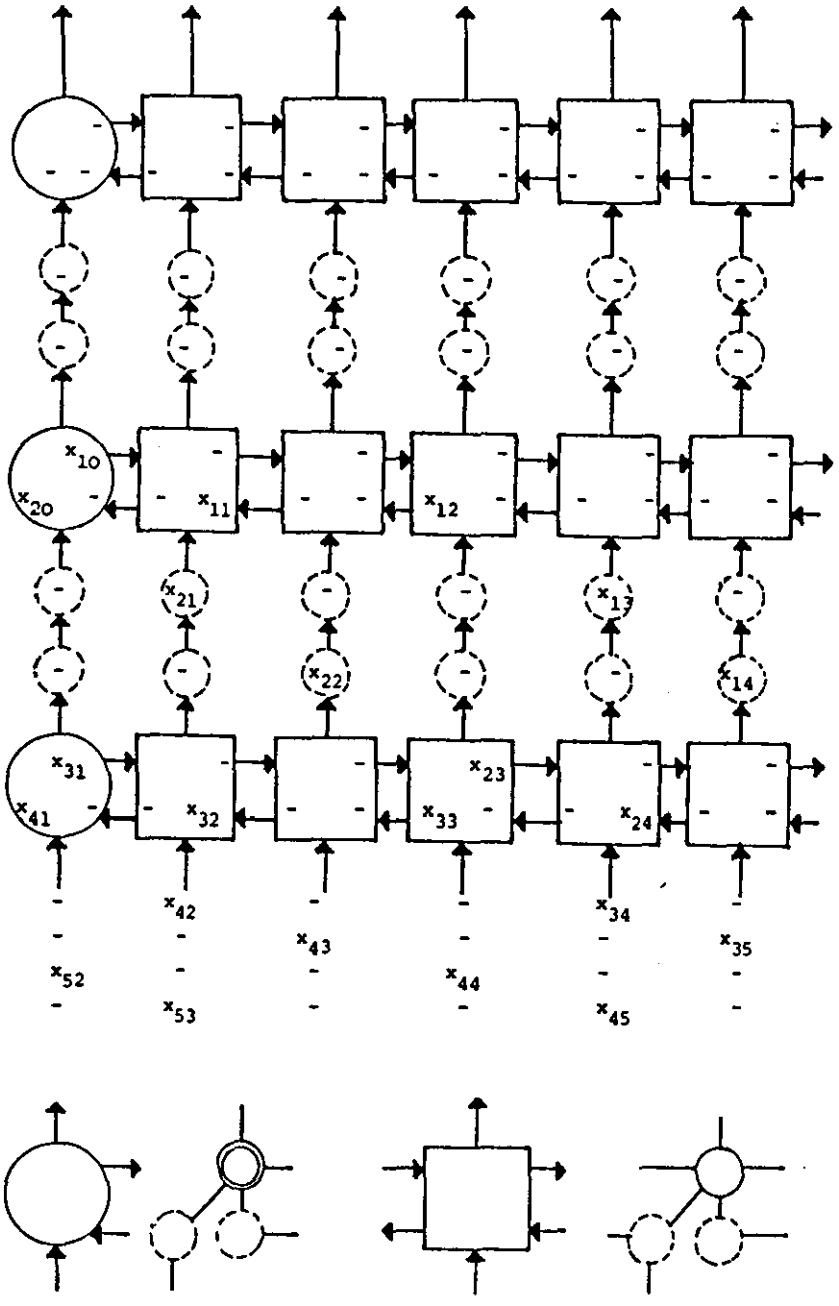


FIGURE 6.4.1: Systolic array for triangularising a band matrix

The latter array is more relevant for our discussions, but we mention the former to characterise the type of array applicable to incomplete eliminations. Fig.(3.2.2.2) accepts inputs in a column ordering rather than the diagonal ordering used by Fig.(6.4.1) and the incomplete

factorisation array compacted in Section (6.3). Thus, a candidate for compaction and hence area/time reduction must have a diagonal input format, and we can confine our attention to the array in Fig.(6.4.1). If the semi-bandwidth of A is W_s, the array requires W_s linear arrays separated by two rows of delay cells between each array, with $2W_s+1$ cells in each row. Each linear array consists of a boundary cell on the left for computing row modification data and modifying cells to update rows using the definitions (3.2.2.6)-(3.2.2.7). Modifications to these basic arrays for general bandwidths follow naturally, and special versions which utilise special properties such as symmetry are considered in Heller and Ipsen [82]. It should be clear that for matrices with a dense band the array structure of Fig.(6.4.1) can be used for either Gauss elimination or an orthogonal reduction approach with eliminations occurring in the order of (6.4.7). However for the matrix associated with problem I the band is not dense and some multipliers associated with elimination are inconsistent requiring a divide by zero. Consequently to utilise the array which can be compacted we must concentrate on triangularisation using rotations. But, the incomplete strategy is based on eliminations and the behaviour of the Gauss method, changing to the orthogonal method could seriously affect the final approximation error. The orthogonal scheme certainly creates additional fill-in outside the main band, which is not present in the elimination method. Putting these problems aside for a moment and considering (6.4.4) in conjunction with Fig.(6.4.1) the array compaction procedure can be applied as follows.

(i) Identity neutral cells: because the input to the array is in diagonal form tracing the path of a single element in each diagonal

not retained for fill-in marks the neutral cells. An example for W_s=6 and r=1 (only the outer most diagonal retained) is shown in Fig.(6.4.2a) These neutral cells can be replaced by delay cells.

(ii) Define array cuts: For purposes of illustration suppose r=3, then Fig.(6.4.2b) defines a cut set which identify the regions of delay cells which can be removed to leave tessalating disjoint regions.

(iii) Compaction: The tessalating sections are fitted back together to make a smaller array, allowing further redundant rows to be omitted (see Fig.(6.4.2c).

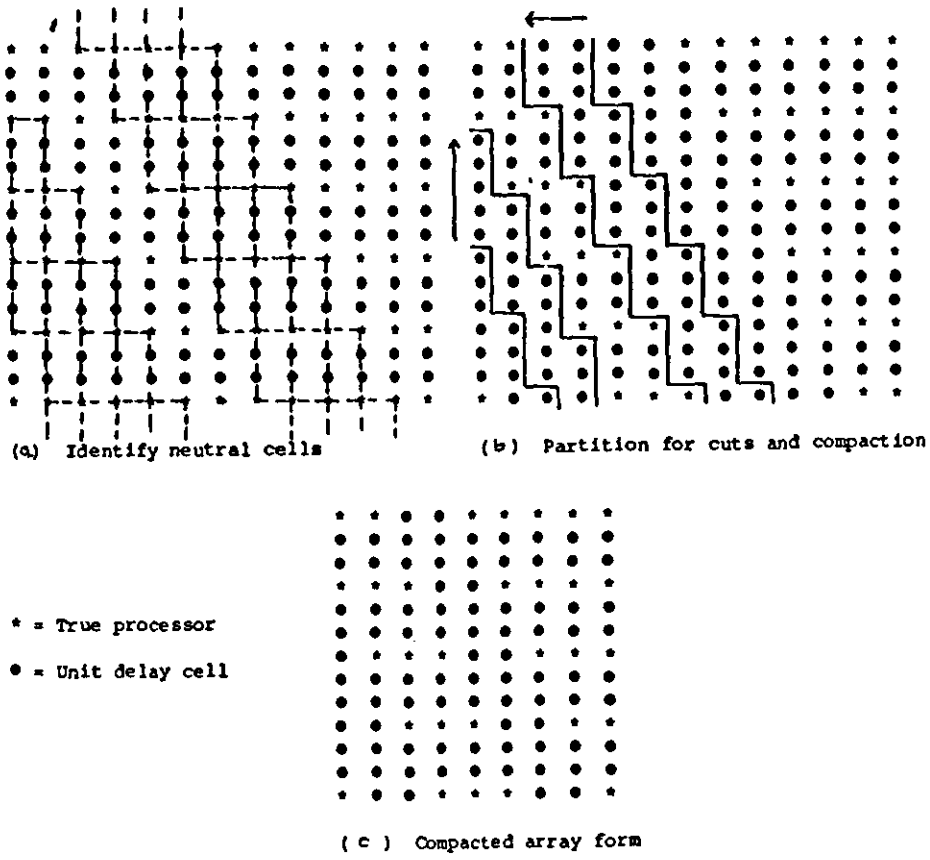


FIGURE 6.4.2: Systolic compaction for incomplete elimination

As before the compacted array solves a similar problem to the original but with a reduced bandwidth given by,

$$\bar{A}\tilde{u} = d \ , \tag{6.4.8}$$

with $\bar{A}$ the matrix in (6.3.24) where $r_1=r$, and $\tilde{u}$ is an N×1 vector of unknowns. After passing $\bar{A}$ and d through the array we solve,

$$\bar{U}\tilde{u} = \tilde{d} \ , \tag{6.4.9}$$

with $\bar{U}$ the modified upper triangular form of $\bar{A}$ and $\tilde{d}$ the updated form of d to yield the exact (complete) solution of (6.4.8) and the approximation error,

$$E = ||u-\tilde{u}||_2 \tag{6.4.10}$$

as in incomplete solution to (6.1.1) for problem I. A compensating factor for controlling the distance between (6.1.1) and (6.4.8) by the use of shifting diagonals can then be applied as follows:

(i) triangularise $\bar{A}$ to produce $\bar{U}$

(ii) shift the outer and retained fill-in diagonals of $\bar{U}$ to form $\hat{U}$ which has the same structure as U_s in (6.4.4).

(iii) solve $\hat{U}u=\tilde{d}$ using backward substitution.

This defines the Systolic Incomplete Elimination Algorithm (SIEA) with the definition,

```
SEQ
  GAUSS(N,r+2,r,Ā,d)                    (6.4.11)
  SUB(N,m,r,Û,d̃)
```

which allows the use of the compacted array. Notice that we have assumed that a Gaussian elimination can be performed on a compacted version of Fig.(6.4.1), generally this is not the case because null diagonals retained for fill-in prevent the formation of some multipliers. However from a numerical viewpoint an algorithm like (6.4.11) can be implemented where the normal elimination ordering rather than (6.4.7) is adopted.

To date incomplete methods have been applied only to elimination algorithms due to the fact that the orthogonal versions require more arithmetic operations and permit fill-in entries beyond the outer diagonals. This fill-in defeats any attempt to reduce storage or time requirements, as the additional fill-in is proportional to the number of subdiagonals. In the case of the SIE algorithm the compacted triangularisation array reduces both area and time. Area is reduced because of cells removed by the cuts, and time because of the decreased latency for data to pass through the compacted array. This leaves only the problem of minmising the approximation error. Below we test two algorithms for both elimination and orthogonal triangularisation referred to as the shifted and unshifted SIEA forms corresponding to (6.4.9) and (6.4.11) respectively, under two conditions:

(i) fixed system size (N) with semi-bandwidth m varied

(ii) fixed m and variable system size (N).

The results are shown in Fig.(6.4.3) T1-T8, notice that when m=10 for example only 8 results are given, this is because only 8 diagonals can be retained for the fill-in (the diagonal and co-diagonal making 10). The tables can be compared sensibly using the following cases:

(i) Elimination schemes (shifted vs unshifted)

(ii) Orthogonal schemes (" " ")

(iii) Elimination vs orthogonal

First consider Tables T1-T4, T1-T2 examine elimination schemes T3-T4 orthogonal versions, where $x_r=\tilde{u}$ for the unshifted case, $x_r=\hat{u}$ for the shifted case, and x=u, with N=50 fixed.

Case (i): Elimination methods compared

The first valuable piece of information is that both the shifted

TABLES OF $E=||x-x_r||_2$ WITH (r=retained diagonals, m=semibandwidth)

r \ m	10	15	20	30	40
1	22.81	36.57	50.87	69.92	69.92
2	15.76	26.12	36.10	51.49	51.49
3	11.42	19.64	28.37	40.02	40.02
4	8.66	15.42	22.71	32.46	32.46
5	6.87	12.5	18.76	27.19	27.19
6	5.6	10.37	15.94	23.35	23.35
7	4.64	8.9	13.81	20.43	20.43
8	3.9	7.89	12.1	18.17	18.17
9	-	6.87	10.79	16.31	16.31
10	-	6.22	9.83	14.89	14.89
11	-	5.61	8.88	13.58	13.58
12	-	4.89	8.28	12.56	12.56
13	-	4.38	7.98	11.78	11.78
14	-	-	7.62	10.98	10.98
15	-	-	6.92	10.17	10.17
16	-	-	6.16	9.55	9.55
17	-	-	5.67	9.18	9.18
18	-	-	5.27	8.8	8.8

T1 ELIMINATION METHOD WITH NO SHIFT (N=50)

r \ m	10	15	20	30	40
1	170.85	119.94	130.66	118.65	84.71
2	172.14	88.75	78.18	63.94	44.66
3	127.73	58.73	50.52	41.38	28.74
4	107.12	44.24	36.91	30.13	20.66
5	102.99	36.36	29.31	23.47	15.85
6	4.89	31.7	24.57	19.06	12.62
7	0.00	29	21.27	15.92	10.25
8	-	27.59	18.84	13.58	8.43
9	-	26.61	17.17	11.76	6.98
10	-	26.68	16.02	10.48	5.9
11	-	27.15	14.91	9.44	5.08
12	-	2.46	14.26	8.78	4.6
13	-	0.00	13.86	8.34	4.19
14	-	-	13.43	7.66	3.73
15	-	-	12.90	6.68	3.32
16	-	-	12.85	5.91	3.15
17	-	-	1.58	5.43	3.10
18	-	-	0.00	5.00	3.06

T2 ELIMINATION METHOD WITH SHIFT (N=50)

TABLES OF $E=||x-x_r||_2$ WITH (retained diagonals, m=semi-bandwidth)

r \ m	10	15	20	30	40
1	57.1	129.19	220.19	343.8	343.8
2	27.95	65.31	112.1	175.4	175.4
3	15.49	38.35	66.74	104.99	104.99
4	9.14	24.67	43.83	69.55	69.55
5	5.5	16.76	30.66	49.28	49.28
6	3.04	11.72	22.44	36.59	36.59
7	1.32	8.48	16.89	28.09	28.09
8	0.00	6.39	12.86	22.17	22.17
9	-	4.40	10.10	17.7	17.7
10	-	3.30	8.18	14.57	14.57
11	-	2.22	6.26	11.79	11.79
12	-	0.90	5.18	9.89	9.89
13	-	0.00	4.72	8.5	8.5
14	-	-	4.08	7.03	7.03
15	-	-	2.82	5.54	5.54
16	-	-	1.49	4.61	4.61
17	-	-	0.71	4.18	4.18
18	-	-	0.00	3.8	3.8

T3 ORTHOGONAL CASE WITH NO SHIFT (N=50)

r \ m	10	15	20	30	40
1	783.02	1925.64	3017.58	4192.56	4129.33
2	7.72	15.15	21.89	26.77	22.76
3	5.88	9.84	12.99	15.36	13.39
4	6.26	9.2	11.15	12.68	10.87
5	5.71	9.28	10.44	11.52	9.62
6	3.89	9.98	9.84	11.01	8.66
7	1.99	10.03	9.45	10.50	8.06
8	0.00	9.03	9.40	10.16	7.79
9	-	7.02	9.91	10.15	7.75
10	-	4.93	9.68	9.85	7.62
11	-	2.48	8.45	9.26	7.36
12	-	1.20	7.49	8.97	7.35
13	-	0.00	6.09	8.35	6.96
14	-	-	4.19	7.53	6.42
15	-	-	2.42	6.87	6.10
16	-	-	1.5	6.58	6.08
17	-	-	0.9	6.19	6.00
18	-	-	0.00	5.58	5.8

T4 ORTHOGONAL METHOD WITH SHIFT (N=50)

FIGURE 6.4.3: Test case for SIE algorithm

TABLES OF $E = \frac{1}{\sqrt{n}}\|x-x_r\|_2$ WITH r-retained diagonals, N=order of system

r \ N	50	100	200	300	400
1	31.14	34.93	36.67	37.24	37.5
2	15.85	15.5	72.19	201.89	286.98
3	9.44	5.93	42.46	8.08	7.02
4	6.19	3.1	6.27	5.98	*
5	4.34	16.81	179.26	558376.31	*
6	3.17	1.81	6.07	57.15	6073876.0
7	2.39	1.81	2.31	3.00	9620.03
8	1.82	2.1	1.94	161.2	15253616
9	1.43	2.7	1.89	949.61	586278.63
10	1.16	3.92	2.94	3.07	27411.68
11	0.88	8.3	11.66	330.55	29333.06
12	0.73	54.01	1.88	23.75	11828.27
13	0.67	6.00	1.82	23.17	3836.94
14	0.58	2.72	3.89	223.51	478144.91
15	0.4	1.5	1.03	5.1	16492.67
16	0.21	0.92	1.16	7.77	18596.34
17	0.1	0.66	1.00	5.16	419.68
18	0.00	0.46	1.56	8.54	2720.83

T5 ELIMINATION METHOD WITH NO SHIFT (m=20)

r \ N	50	100	200	300	400
1	18.48	61.14	1120.97	27863.99	774982.66
2	11.06	50.3	1732.06	84016.92	4514921
3	7.14	31.82	996.95	43561.84	2146412.5
4	5.22	22.66	650.89	25988.51	1175797.8
5	4.14	18.05	500.24	19353.06	853503.63
6	3.47	15.48	434.29	17120.54	773883.81
7	3.01	13.93	410.64	17153.11	826388.94
8	2.67	12.97	414.13	18925.63	1003660.8
9	2.43	12.52	443.95	22842.81	1373093.2
10	2.27	12.40	501.62	29904.15	2097930.5
11	2.11	12.56	595.5	42412.86	3584678.5
12	2.02	12.99	743.18	65525.31	6923220.0
13	1.96	13.59	979.89	111735.8	15449002.0
14	1.9	14.63	1387.24	216521.77	41596356
15	1.82	16.19	2159.95	499348.94	*
16	1.82	18.51	3862.79	1489468.5	*
17	0.22	0.58	1.91	5.14	1408.02
18	0.00	0.00	0.00	0.00	0.00

T6 ELIMINATION METHOD WITH SHIFT (m=20)

TABLES OF $E = \frac{1}{\sqrt{n}}\|x-x_r\|_2$ WITH r-retained diagonals, N=order of matrix

r \ N	50	100	200	300	400
1	X	X	X	X	X
2	15.85	17.74	18.61	18.89	19.03
3	9.44	10.54	11.04	11.21	11.29
4	6.2	6.9	7.23	7.33	7.38
5	4.33	4.82	5.04	5.11	5.15
6	3.17	3.52	3.68	3.73	3.75
7	2.39	2.64	2.76	2.79	2.81
8	1.82	2.01	2.1	2.12	2.13
9	1.43	1.56	1.63	1.64	1.66
10	1.16	1.24	1.29	1.30	1.3
11	0.89	0.99	1.02	1.03	1.03
12	0.73	0.81	0.81	0.81	0.81
13	0.67	0.63	0.63	0.62	0.62
14	0.58	0.47	0.47	0.46	0.46
15	0.4	0.38	0.33	0.32	0.32
16	0.21	0.22	0.2	0.19	0.19
17	0.1	0.1	0.1	0.09	0.09
18	0.00	0.00	0.00	0.00	0.00

T7 ORTHOGONAL METHOD WITH NO SHIFT (m=20)

r \ N	50	100	200	300	400
1	426.75	1655437.75	X	X	X
2	3.00	3.89	2.6	2.66	2.95
3	1.84	2.79	11.74	63.07	369.78
4	1.58	3.47	31.72	382.95	4994.77
5	1.48	4.23	63.94	1306.31	29110.53
6	1.39	4.88	112.44	3279.58	111948.19
7	1.34	5.62	176.89	6726.82	300117.94
8	1.33	6.14	241.00	11994.77	599996.25
9	1.4	6.82	274.82	16186.43	1019278.37
10	1.37	7.2	296.57	16416.67	1059895.75
11	1.2	6.53	260.38	13155.61	751249.625
12	1.06	5.27	175.85	7327.98	335885.47
13	0.86	3.54	89.07	2664.53	X
14	0.59	2.18	32.99	X	X
15	0.34	1.12	8.97	X	X
16	0.21	0.53	1.94	X	X
17	0.13	0.25	0.48	X	X
18	0.69	0.89	0.96	X	X

T8 ORTHOGONAL METHOD WITH SHIFT (m=20)

* results too large to be outpu

x values omitted.

FIGURE 6.4.3: (cont)

and unshifted versions (in T1 and T2) preserve the preconditioning property of error reduction as $r \to m$. The shifted case also exhibits a steeper error reduction for the inclusion of only a few diagonals when compared with the unshifted case, but this is offset by the larger initial error when r=1. Notice also that the two tests exhibit opposing characteristics as m increases, with the initial error increasing in the unshifted test, but decreasing in the shifted test. Consequently as m increases the diagonal shift before substitution makes it superior to the unshifted scheme.

Case (ii): Orthogonal methods compared

For the orthogonal case both methods produce increasing initial errors (for r=1) which increase as m increases, and which are significantly higher than the elimination schemes. Again the initial error of the unshifted test is much less than that of the shifted method, but the latter exhibits a much steeper error reduction curve between r=1 and r=2. Consequently retaining only a few diagonals (which minimises array size) minimises the error when the orthogonal SIEA is adopted.

Case (iii): Orthogonal vs elimination methods

For the two unshifted schemes the behaviour is similar with elimination superior for small m but replaced by the orthogonal form when $r \approx m$. With the shifted schemes, the orthogonal method has a higher initial error but its steeper error reduction compensates to produce better approximations than the elimination for small r.

Next we consider the effect of matrix size with bandwidth (m=20) fixed as shown by T5-T8.

Case (i): Elimination method comparison

The first remarkable feature is that increasing N causes a number of peaks to develop where approximation error increases even for a significant number of retained diagonals. The larger N is relative to m the more peaks appear causing the approximation error to be more erratic. Both the shifted and unshifted cases give increasing initial errors as N grows, but the latter controls the error better and this shows up in the oscillating peak values as errors develop.

Case (ii): Orthogonal method comparison

In both the shifted and unshifted methods the initial error (for r=1) is often very large and is neglected in some of the tables. The error for r=2 in the unshifted method is small and increases with N, while the error of the shifted form is even smaller (and can actually decrease). However the most interesting feature is that the unshifted form retains the 'idealized' error reduction form while the shifted form develops a single peak, as N increases. As this peak develops it becomes apparent that the best approximation uses $r<3$.

Case (iii): Orthgonal vs Elimination schemes

It is clear that the orthogonal schemes outperform the elimination schemes. In both the unshifted cases the determining factor is the error growth at r=2, and the subsequent approximations as $r \to m$. The orthogonal methods provide a much more predictable error pattern which allows a reliable estimation of the value r and hence the size of the compacted array. Choosing $r<8$ and $r<3$ in the unshifted and shifted orthogonal schemes gives a reasonable approximation error.

A theoretical explanation of the behaviour of these systolic incomplete methods has proved to be problematical. The main difficulty

arises from the fact that the incomplete algorithms were modified to satisfy minimal area constraints of systolic arrays. This constraint is essentially non-mathematical and making its effect on changes on the numerical method difficult to trace in a logical manner. However an intuitive understanding of the results can be gleaned from the analysis in Lipitakis [78] giving a justification for the original approximate (or incomplete) methods. Essentially we require just one theorem which is stated without proof.

Theorem 6.4.1: Let A be an $N^{\times}N$ matrix of bandwidth m (like (2.5.1.11)) and consider the factorisation $A=DT^{t}TD$ where $DT^{t}=L_{s}$ and $TD=U_{s}$ with D a diagonal matrix. Let $t_{i,j}$, i=1(1)m, j=1(1)N-m+1 be the elements of T with r the number of diagonals retained in the bandwidth. Then the elements $t_{i,j}$ are monotonically decreased i=1(1)m-r, (i.e. the sequence $t_{1,j},t_{2,j},\ldots,t_{m-r,j}$, j=1(1)N-m+1 decreases monotonically).

It follows that the elements on diagonals moving away from the outer diagonals of A have less and less effect on the method, and can be omitted. However, under certain circumstances the monotonicity relationship breaks down. For instance with narrow banded matrices the values in T are not monotonic after i>m-r. This fact is unimportant for the numerical versions of incomplete methods for problem I always satisfy Theorem (6.4.1). But narrow banded matrices are ideal for systolic arrays and the compaction technique is aimed at deriving a new procedure which satisfies this criteria. Consequently, the shift in technique used in compaction creates an artifically narrow band which may contradict Theorem (6.4.1), and explains why the SIF and SIE errors are generally higher and more erratic than those of Lipitakis. In the case of the SIF method the shift inwards is balanced

by the shift out of the L_s and U_s matrix elements which effectively smooths the error approximation. For the Elimination based technique this smoothing is unbalanced because the shift in of subdiagonal parts of A are used as multipliers to modify the righthand side d directly, with only the super diagonals of the triangularised U creating an error smoothing effect on the shift out. A possible solution to this problem is to simulate a shift of multipliers by suitable operations on the modified vector d before backsubstitution. In the case of the orthogonal method we have two favourable characteristics. First the rotation matrices involved in triangularisation provide inherently stable computations. Second, the additional fill-in outside the main band compensates to some extent for the effects of righthand side modification after shift in. (T8 indicates that the effect of this additional fill-in is most detrimental when $r \approx m/2$).

Finally, we remark that the errors produced by incomplete triangularisation are larger than those for factorisation, which in turn are worse than the original incomplete schemes. We have examined only one shifting strategy for smoothing the error associated with the monotonicity break down and there may be more suitable ones. Furthermore our designs have been restricted to matrices for problems I and II, other forms may also benefit from the array compaction technique, and produce much smoother error approximation curves.

6.5 ITERATIVE ARRAYS FOR PRECONDITIONING

Let us recall the global structure of the proposed preconditioned solver in Fig.(6.1.1). From our experiences of previous preprocessors

we are now in a position to develop the cascaded iteration array or CIA, with respect to the preconditioning schemes described in Section (6.1). Intuitively the ideal CIA is a sequence of pipelined linear arrays with each array computing a single iteration, and r arrays producing r iterations. Fig.(3.2.3.1) illustrates the structure more clearly and was adopted by Berzins, Buckley and Dew [83] to develop unpreconditioned Jacobi and Gauss-Seidel iterative schemes. Below we shall compare these unpreconditioned arrays with new preconditioned arrangements for the following cases:

(i) Unlimited amount of hardware:- the CIA can have as many iterations as required for convergence.

(ii) A finite number of iterations:- the length of the CIA is bounded.

(iii) "Bag of" approach:- a collection of arrays such as the CIA and preconditioning preprocessor hang on the same host bus and compute sequentially. (Small granularity in sys-pack).

These cases allow the effectiveness of preconditioned strategies to be assessed with realistic restrictions on actual implementation. For instance using Fig.(3.2.3.1) and Theorem (3.2.3.1) the minimum time,

$$T_0 = 2N+2r(p+1)-1 \ , \qquad (6.5.1)$$

is obtained for the unpreconditioned overrelaxed Jacobi iterative scheme, and provides a suitable base time for comparisons using Case (i).

REMARK: The latency of each iteration is taken as 2(p+1) rather than 2(p+1)-1 to allow time for the overrelaxation calculations.

To apply case (ii) we define r_1 as the total number of iterations required for convergence, and r (fixed) the number of arrays in the CIA.

Full convergence is achieved by a multi-pass arrangement yielding,

$$T_1 = 2\left\lceil\frac{r_1}{r}\right\rceil N + 2\left\lceil\frac{r_1}{r}\right\rceil r(p+1) - \left\lceil\frac{r_1}{r}\right\rceil$$

or

$$T_1 = 2\frac{r_1}{r}N + 2r_1(p+1) - \frac{r_1}{r} \quad , \tag{6.5.2}$$

when r_1 is divisible by r. This result gives a better assessment of time for a real implementation.

6.5.1 Implicit Preconditioned Arrays

The first task is to illustrate how restrictive the systolic constraints are on the preconditioning problem. Assuming we apply the incomplete arrays to produce the L_s and U_s factors the additional hardware and latency associated with the preconditioner of Fig.(6.1.1) is minimal. The effect of improved convergence rate can be applied directly to the CIA to balance cell and time reductions and produce an improved area/time trade-off.

However due to the implicit nature of (6.1.7) the CIA cannot in general acquire the cascaded form necessary for high throughput. To see this we arrange (6.1.7) into the following procedure,

$$\left.\begin{array}{ll} \text{STEP 1:} & z = d - Au^{(i)} \\ \text{STEP 2:} & L_s U_s \Delta u = \alpha z \text{ (i.e. (6.1.8))} \\ \text{STEP 2:} & u^{(i+1)} = u^{(i)} + \Delta u \text{ : IF not converged THEN GOTO STEP 1} \end{array}\right\} \tag{6.5.1.1}$$

where step 2 demands the solution of coupled systems,

$$\left.\begin{array}{l} L_s y = \alpha z \\ U_s \Delta u = y \end{array}\right\} \tag{6.5.1.2}$$

which reveals the problem of cascading iterations immediately. L_s and U_s are sparse lower and upper triangular matrices which requires a switching of vector orientation for forward and backward substitution

to solve (6.5.1.2) making (6.5.1.1) a sequential task. From (6.1.7) a linear array for the CIA can be produced with the form of Fig.(6.5.1), which contains the undesirable feature of LIFO's whose size is related to N not the bandwidth of the matrix. Apart from the difficulty of pipelining iterations, for large systems the size of each linear array now prohibits cascading. Simple analysis of the standard matrix-vector and substitution arrays produces an iteration latency of 4N+(2p+3) giving,

$$T_2 = 4Nr + (2p+3)r \ , \qquad (6.5.1.3)$$

for r iterations. It follows that under the case (i) assumptions the unpreconditioned scheme is superior for any rate of convergence. For case (ii) assumptions, we define r_1 and r_2 as the number of unpreconditioned and preconditioned iterations required for convergence. As cascading is difficult in (6.5.1.1) we fix r=1 for (6.5.1.3) which also minimises area, and derives a computation speed-up by

$$S_p = \frac{\frac{2r_1}{r} + \frac{2r_1(p+1)}{N} - \frac{r_1}{rN}}{4r_2 + \frac{(2p+3)r_2}{N}} \qquad (6.5.1.4)$$

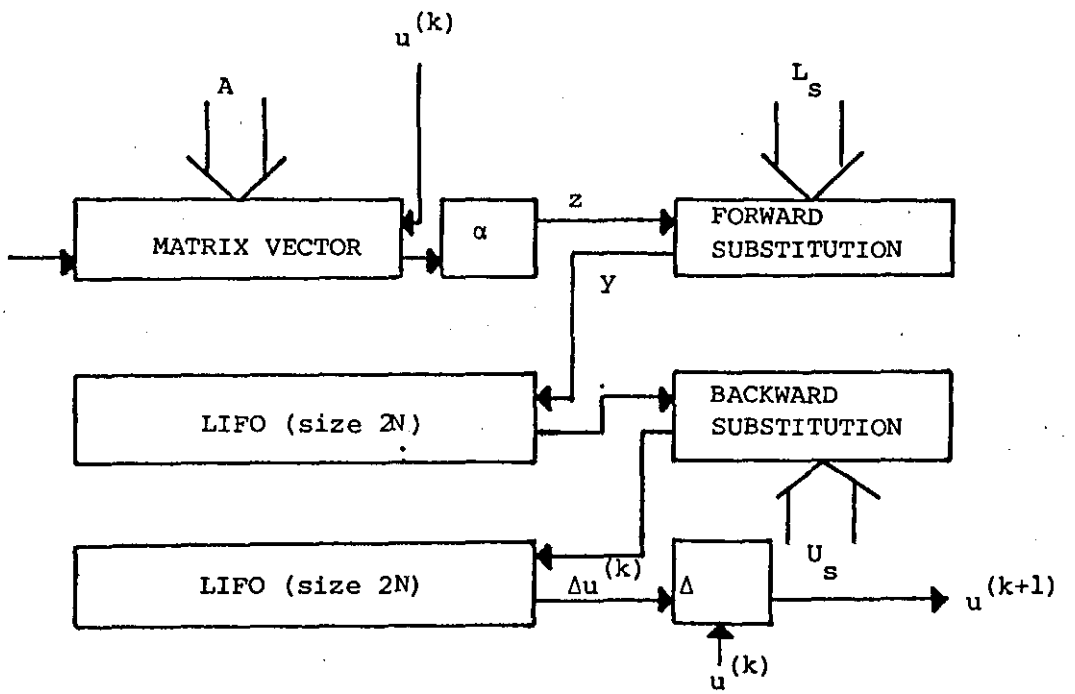


FIGURE 6.5.1: Implicit preconditioned iteration

neglecting small terms with $n \gg r_1, r_2, p$ we have,

$$\frac{2r_1}{r} + \frac{2r_1(p+1)}{N} > 4r_2 + \frac{(2p+3)r_2}{N}$$

$$r_1 > 2r_2 r + \frac{r}{2N}\{(2p+3)r_2 - 2r_1(p+1)\}$$

and since $\frac{r}{2N} \to 0$ as $N \to \infty$ because r is fixed, the final result is,

$$r_1 > 2r_2 r \ , \qquad (6.5.1.5)$$

consequently a speed-up occurs if the unpreconditioned scheme requires 2r times the preconditioned iterations for convergence. In the cases where r is small this is easily satisfied, and the only way for the unpreconditioned method to compete is on area terms. From Theorem (3.2.3.1) the unpreconditioned form requires rw ips cells and approximately $2(r-1)wp$ synchronising delay registers between iterations. Thus,

$$2(2N+w) > 2(r-1)wp + rw$$

which after some manipulation yields

$$\frac{2N+w(p+1)}{w(p+0.5)} > r \ , \qquad (6.5.1.6)$$

giving the unpreconditioned scheme less area for reasonable values of N and r. It follows by substitution of (6.5.1.6) into (6.5.1.5) that the implicit preconditioners save area and time iff

$$r_1 > 2\left\{ \frac{2N+w(p+1)}{w(p+0.5)} \right\} r_2 \qquad (6.5.1.7)$$

which is unlikely with N large.

An alternative approach, which allows cascading, is to use the incomplete arrays to produce a better initial starting vector as an input for an unpreconditioned CIA. This idea is reminiscent of the iterative refinement technique where a system is solved directly and rounding errors cleaned up by successive improvements. In our case

the initial solution is only approximate (from the SIF or SIE methods) and we can employ more general iterative methods to achieve convergence. Now allowing 4N+c cycles for the full incomplete solution, where c>0 is the latency of the incomplete array plus delays for the forward and backward solvers and 4N is the time to fill vector reversal LIFOs similar to those in Fig.(6.5.1), the timing for this method is,

$$T_3 = \left\{\frac{2r_2}{r} N + 2r_2(p+1) - \frac{r_2}{r}\right\} + 4N + c. \qquad (6.5.1.8)$$

So for a speed-up on the unpreconditioned form,

$$S_p = \frac{\frac{2r_1}{r} + \frac{2r_1(p+1)}{N} - \frac{r_1}{rN}}{\left\{\frac{2r_2}{r} + \frac{2r_2(p+1)}{N} - \frac{r_2}{rN}\right\} + 4 + \frac{c}{N}}$$

and after neglecting smaller terms yields,

$$\frac{r_1}{r} > \frac{r_2}{r} + \frac{(p+1)}{N}[r_2 - r_1] + 2$$

or

$$r_1 > r_2 + 2r \text{ , for large } N, \qquad (6.5.1.9)$$

which is a weaker condition for the reduction in the total iterations necessary for the alternative method to improve the speed. Notice, however, that (6.5.1.1) ensures a larger modification Δu on each iteration, rather than one large jump before iteration starts in the alternative arrangement. Consequently, the difference between r_1 and r_2 may not be as great, as for (6.5.1.1) giving poorer performance. Result (6.5.1.9) also relies on the assumption that both CIA forms contain the same amount of hardware, and ignores the preprocessor overhead. A more rigorous set of relations can be derived by introducing additional parameters for each CIA, allowing a trade-off between speed-up and array sizes. To develop these relationships further we examine explicit preconditioning techniques.

6.5.2 Explicit Preconditioning Arrays

Consider the preconditioned Jacobi scheme in (6.1.15) which is easily formulated as the two step process,

$$\left.\begin{array}{l} \text{STEP 1:} \quad M = B^2 \ , \quad z = (I+B)d \\ \text{STEP 2:} \quad u^{(i+1)} = Mu^{(i)} + z \\ \qquad\qquad \text{IF NOT converged THEN GOTO STEP 2} \end{array}\right\} \qquad (6.5.2.1)$$

which partitions naturally onto the preconditioned array format of Fig.(6.1.1) as illustrated in Fig.(6.5.2), where the CIA consists of pipelined (cascaded) matrix vector arrays. The preprocessor is comprised of three separate sections, a modified matrix vector array for computing z, the reduced hexagonal array of Section (6.2), and a reformatting array for modifying the hex output, for correct CIA input. Operation of the preprocessor is trivial but we point out the salient features. The vector z is computed as z=d+Bd (i.e. set $x_i=y_i^{(1)}=d_i$ in (3.2.1.4)) so that explicit knowledge of I+B is not required and B can be pipelined directly into the reduced hex. The expander is simply the preprocessing described by Fig.(6.2.5) and the hex array computes with dataflow like Fig.(6.2.2) with $A_r=B_r=B$. Consequently the latency of the hex output is determined from Theorem (6.2.2) with k=1, i.e. $2w_o+2c_o+3$ where $c_o>0$ is a small constant and w_o is the bandwidth of B. This leaves only the reformatting array which is essentially a linear array of delay queues adding the right amount of skew to the B^2 hex output and synchronises it with vectors z and $u^{(0)}$ for input to the CIA. To complete the picture of the preprocessor consider the dataflow and input format B. In order to use the reduced hex B must have the standard hex format with two delays to separate successive input elements of the same stream, and is incompatible with the standard

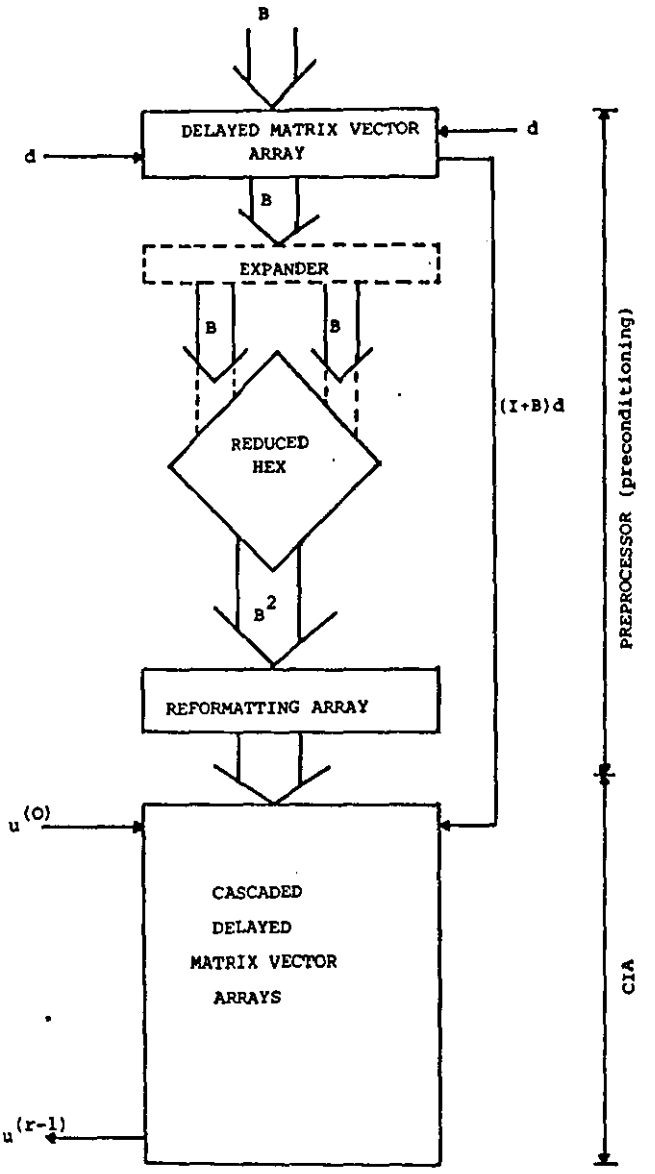


FIGURE 6.5.2: Systolic preconditioned Jacobi iteration

linear array format of Fig.(3.2.1.3). Adjusting the number of dummy elements or retiming data 'on-the-fly' is both difficult and messy so to overcome the problem we define a special matrix vector array which accepts hex input format. The array and operation snapshots are shown in Fig.(6.5.3). Notice that the inputs on the left side of the array are skewed slightly, the expander must remove this skew for hex input, and the reformatter must restore it for hex output. Both tasks are achieved by simple data delay queues.

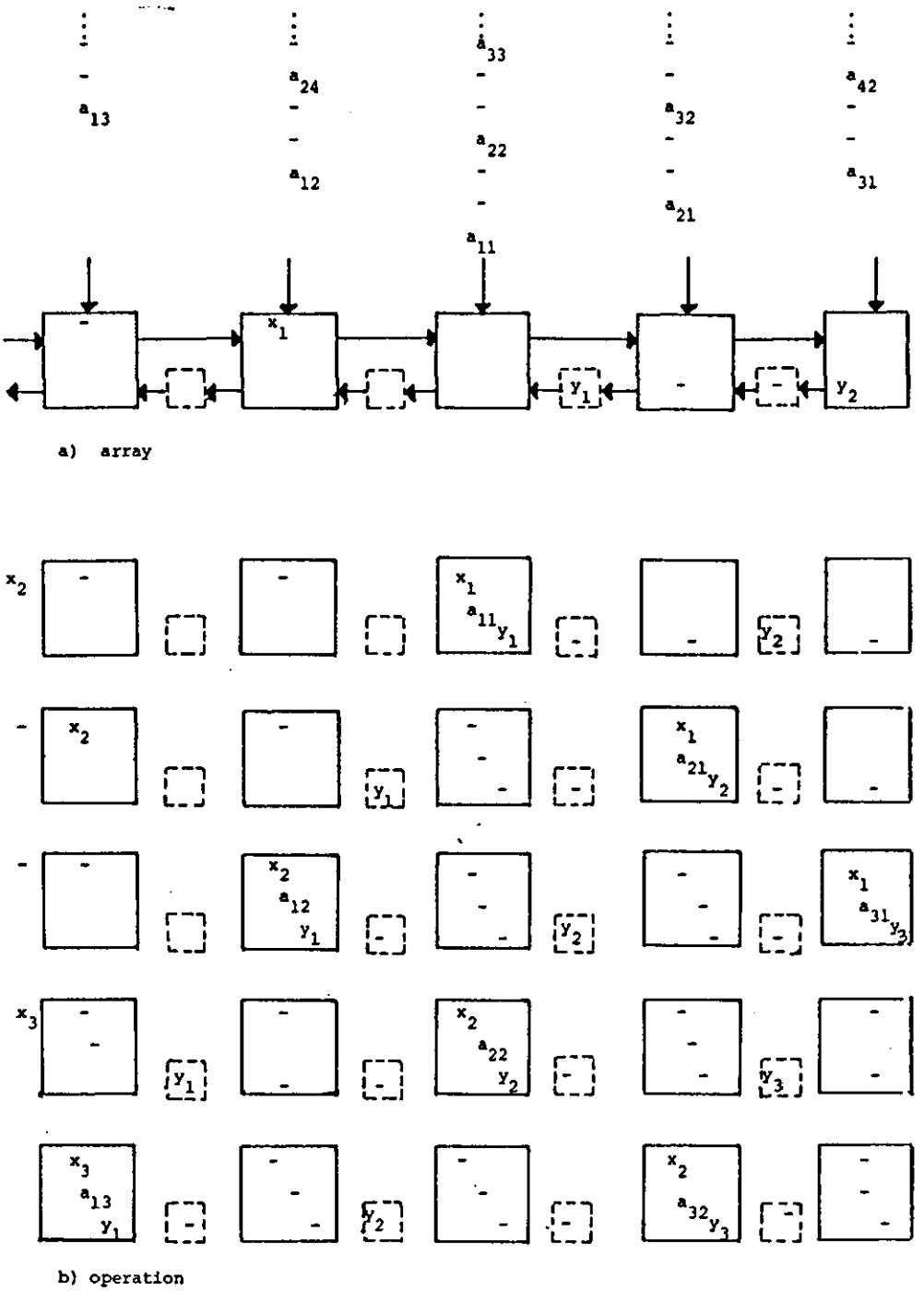


FIGURE 6.5.3: Snapshots of delayed matrix vector computation

Theorem 6.5.2.1: The delayed matrix vector product Ax=y for an n×n matrix A of bandwidth w=p+q-1 requires T=3n+2w-1 ips cycles, w ips cells and w-1 delay registers.

Proof: [Observe the array and dataflow in Fig.(6.5.3)].

The delayed matrix vector must also be used as an iteration component in the CIA and the cascaded timing analogous to Theorem (3.2.3.1) is derived as follows:

(i) The initial delay to synchronise x and y (above) in the first iteration is MAX(p-1,2(q-1)).

(ii) the ith iteration starts 2(p-1)+p cycles after the start of iteration (i-1), for i=2(1)r.

(iii) hence the rth iteration starts after (r-1)[3p-2]+MAX(p-1, 2(q-1)) cycles and begins outputting after a further delay of 2(p-1)+1 cycles.

As the length of output is 3n the total CIA time is

$$T = 3n+(r-1)[3p-2]+MAX(p-1,2(q-1))+2(p-1)+1$$

$$= 3n+r[3p-2]+MAX(p-1,2(q-1))-p+1 . \qquad (6.5.2.2)$$

For the N×N matrices considered in problems I and II discussed previously p=q thus producing,

$$T = 3N+r[3p-2]+p-1 . \qquad (6.5.2.3)$$

Now the input to the CIA in Fig.(6.5.2) is B^2, and if B has bandwidth w_o, from (6.2.3) has bandwidth $2w_o-1$. Consequently substituting 2(p-1) for p in (6.5.2.3) yields the true CIA time

$$T = 3N+r[6p-8]+2p-3 . \qquad (6.5.2.4)$$

From Theorem (6.2.2) the latency of the preprocessor is at least

$$c_1 = 2w+2c_o+3 = 2(2p-1)+2c_o+3$$

(excluding the delays of the matrix vector and reformatter which are

also proportional to p). Thus the preconditioned Jacobi iterative method gives a total time of

$$T_2 = 3N+r[6p-8] + c \ , \qquad (6.5.2.5)$$

where c is the total preprocessor delay. Although the scheme uses cascaded arrays it still contains a number of undesirable features:

(i) The computation time for matrix vector has increased from O(2N) to O(3N) cycles.

(ii) Each iteration requires twice as much hardware in the preconditioned case than for the unpreconditioned case.

(iii) There is a hardware overhead for the preconditioning.

How does this affect array performance?

Case (i): Unlimited hardware

Using (6.5.1) and (6.5.2.5),

$$S_p = \frac{T_O}{T_1} = \frac{2N+2r_1(p+1)-1}{3N+r_2(6p-8)+c}$$

which for a speed-up demands $S_p>1$ implying,

$$2N+2r_1(p+1)-1 > 3N+r_2(6p-8) + c$$

or,

$$r_1 > \frac{1}{2(p+1)}\{(N+c+1) + r_2(6p-8)\} \ , \qquad (6.5.2.6)$$

where r_1 and r_2 are the number of unpreconditioned and preconditioned iterations respectively. If $N>>r_1,r_2$ and B is really banded the unpreconditioned schemes always outperform the preconditioned schemes. But if $r_1 \approx N$, or $r_1>N$ a speed-up becomes likely as by definition $r_1>r_2$. Irrespective of whether a speed-up does occur we can still save hardware provided the weaker condition $r_1>2r_2$ is satisfied. Each iteration of the ordinary method uses w ips cells compared with 2w-1 in the preconditioned form, giving a saving,

$$s = r_1 w-(2w-1)r_2 = w(r_1-2r_2) + r_2 \qquad (6.5.2.7)$$

and with $r_1=2r_2$ we save $s=r_2$. This saving can be used to offset the preprocessor cost and the additional delay registers associated with the modified matrix vector array and compensates in a small way for a lack of speed-up when r_1 is large.

Case (ii): Fixed length CIA's

Let $\bar{r}_1$ and r_1 be the number of iterations and number of arrays in the CIA for the unpreconditioned Jacobi form, and $\bar{r}_2, r_2$ the corresponding numbers for the preconditioned Jacobi method. We now define the speed-up as,

$$S_p = \frac{\bar{r}_1\left\{\frac{2N}{r_1} + 2(p+1) - \frac{1}{r_1}\right\}}{\bar{r}_2\left\{\frac{3N}{r_2} + 2(3p-4) + \frac{c}{r_2}\right\}} \qquad (6.5.2.8)$$

and for $S_p>1$ we must have,

$$\frac{\bar{r}_1}{\bar{r}_2} > \frac{\left\{\frac{3N}{r_2} + 2(3p-4) + \frac{c}{r_2}\right\}}{\left\{\frac{2N}{r_1} + 2(p+1) - \frac{1}{r_1}\right\}} \qquad (6.5.2.9)$$

or

$$\frac{\bar{r}_1}{\bar{r}_2} > \frac{3}{r_2}\left(\frac{r_1}{2}\right) \text{ for N sufficiently large,}$$

giving,

$$\bar{r}_1 > \frac{3}{2}\left(\frac{r_1}{r_2}\right)\bar{r}_2 \qquad (6.5.2.10a)$$

Now suppose $r_1=\alpha r_2$ for $\alpha>1$ the saving in hardware is given by,

$$S = wr_1-(2w-1)r_2 = r_2 w(\alpha-2)+r_2 \ , \qquad (6.5.2.10b)$$

which relates the convergence rates to array speed-up, and the saving in cells to the relative sizes of the two CIA's. It follows that if bounds on the rates of convergence of the ordinary and preconditioned iteration matrices are known, a bound,

$$\frac{2\bar{r}_1}{3\bar{r}_2} > \alpha \qquad (6.5.2.10c)$$

on α can be derived and the maximum cell savings which still achieve a speed-up located. Furthermore from (6.5.2.10b) if $\alpha>2$ and $r_2>w$ (which is perfectly feasible for narrow banded systems and a good preconditioning) we obtain the saving $s=O(w^2)$. Now, the reduced hex requires w^2 cells and the delayed matrix vector array in the pre-processor an additional w cells. Thus the cell savings compensate for the additional preprocessor cells and still achieves a speed-up.

Case (iii): The 'bag of' approach

The 'bag-of' approach is attractive from the following viewpoints:

(i) A fixed sized array like Fig.(6.5.2) operated in a multi-pass mode requires the preconditioning to be performed again and again needlessly.

(ii) If we perform preconditioning and iteration sequentially rather than pipelining, the preprocessor output could be reformatted in the host to allow the faster standard matrix vector array to be used in the CIA.

With the latency of the preprocessor denoted by c, the time for the sequential operation of the preconditioner is,

$$T_3 = 3N + c \ , \qquad (6.5.2.11a)$$

and for the CIA is given by,

$$T = 2N+r(2p-1) \ , \qquad (6.5.2.11b)$$

using Theorem (3.2.3.1) and 2p-1 substituted for p. Consequently, for a fixed sized CIA,

$$S_p = \frac{\bar{r}_1\left\{\frac{2N}{r_1} + 2(p+1) - \frac{1}{r_1}\right\}}{(3N+c) + \bar{r}_2\left\{\frac{2N}{r_2} + (2p-1)\right\}} \qquad (6.5.2.11c)$$

which for a speed-up yields,

$$\frac{\bar{r}_1}{r_1} > \frac{3}{2} + \frac{\bar{r}_2}{r_2}$$

or

$$\bar{r}_1 > r_1\left[\frac{3}{2} + \frac{\bar{r}_2}{r_2}\right] \qquad (6.5.2.12a)$$

with the cell savings still given by (6.5.2.10b), and the bound on α now of the form,

$$\frac{2\bar{r}_1}{[3r_2+2\bar{r}_2]} > \alpha \ . \qquad (6.5.2.12b)$$

Thus for the 'bag of' approach to be faster than pipelining we must have,

$$\bar{r}_2\left\{\frac{3N}{r_2} + 2(3p-4) + \frac{c}{r_2}\right\} > (3N+c) + \bar{r}_2\left\{\frac{2N}{r_2} + (2p-1)\right\}$$

which after some manipulation and rejection of small terms yields,

$$\bar{r}_2 > 3r_2 \ , \qquad (6.5.2.13)$$

which by substituting into (6.5.2.12b) for $3r_2$ gives,

$$\frac{2}{3}\left(\frac{\bar{r}_1}{\bar{r}_2}\right) > \alpha \ ,$$

for the speed-up and area savings as before.

Given this success in relating the preconditioning strategy convergence rate to speed-up and hardware savings we may attempt further improvements by more preprocessing. For example, the Jacobi scheme in (6.1.15) can be modified to perform two iterations for every linear array of the CIA. Two successive iterations can be written as,

$$\left.\begin{aligned} u^{(i+1)} &= B^2u^{(i)} + (I+B)d \\ u^{(i+2)} &= B^2u^{(i+1)} + (I+B)d \end{aligned}\right\} \qquad (6.5.2.14)$$

and on substitution produces,

$$u^{(i+2)} = B^4u^{(i)} + B^2(I+B)d+(I+B)d \qquad (6.5.2.15)$$

which has the form,

STEP (i) compute B^2 and $v=(I+B)d$

STEP (ii) compute $M=B^2*B^2$ and $w=(B^2+I)v$

STEP (iii) $u^{(i+2)}=Mu^{(i)}+w$

IF not converged THEN GOTO STEP (iii)

Steps (i) and (ii) now form an extended preprocessor as shown in Fig. (6.5.4) which uses two pipelined reduced hex arrays separated by delayed matrix vector arrays. The delay through the reduced hex is $4w_1+4c_o+6$, where $w_1=2w_o-1$, and $c_o>0$ is a constant derived from Theorem (6.2.2), with further delays for the delayed matrix vector arrays (2 cycles) and reformatting, we conclude that the total preconditioner is again proportional to the bandwidth of B and not very significant. From (6.2.3) B^4 has the bandwidth $w_2=4w_o-3=4(2p-1)-3$ for problem I. Thus each compressed linear array of the CIA uses approximately four times the hardware of the unpreconditioned array. The time of this new compressed iterative pipeline can be derived directly from (6.5.2.3) by substituting 4p-3 for p giving,

$$T = 3N + r(12p-11)+4(p-1) \ ,$$

and with $c_2>0$ the delay of the preprocessor is,

$$T_4 = 3N + r(12p-11) + c_2 \ . \qquad (6.5.2.16)$$

Observe that as each compressed array performs two iterations (6.5.2.16) is the time for 2r iterations. The array speed-up over the ordinary scheme is given by,

$$S_p = \frac{2\bar{r}_1\left\{\frac{2N}{r_1} + 2(p+1) - \frac{1}{r_1}\right\}}{\bar{r}_2\left\{\frac{3N}{r_2} + (12p-11) + \frac{c_2}{r_2}\right\}} \qquad (6.5.2.17)$$

as the compressed method requires only $(\frac{\bar{r}_2}{r_2})$ passes, and for $S_p>1$, we must have,

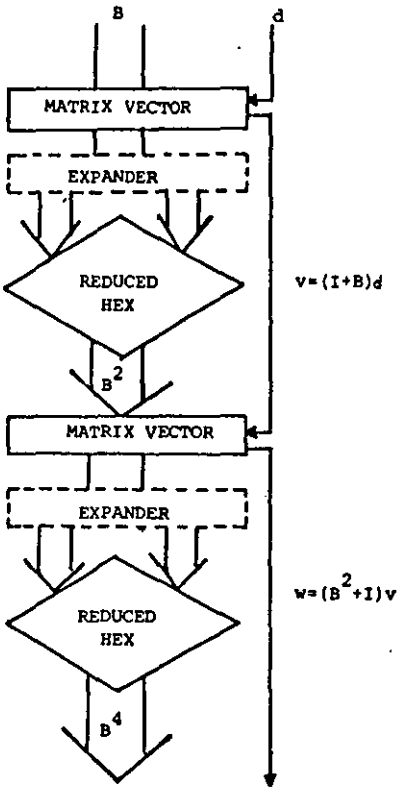


FIGURE 6.5.4: Preprocessor for compressed Jacobi iteration

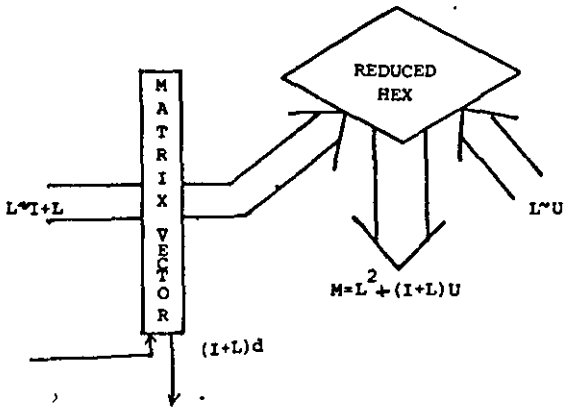


FIGURE 6.5.5: Gauss-Seidel preprocessor

$$\frac{2\bar{r}_1}{\bar{r}_2} > \frac{\left\{\frac{3N}{r_2} + (12p-11) + \frac{c_2}{r_2}\right\}}{\left\{\frac{2N}{r_1} + (2p+1) - \frac{1}{r_1}\right\}} \qquad (6.5.2.18)$$

or,

$$\frac{\bar{r}_1}{\bar{r}_2} > \frac{\frac{3}{r_2}}{\frac{4}{r_1}} = \frac{3}{4}\left(\frac{r_1}{r_2}\right) \qquad (6.5.2.19a)$$

for N sufficiently large. With $r_1=\alpha r_2$ the cell saving is defined as,

$$S = wr_1-(4w-3)r_2 = r_2w(\alpha-4)+3r_2 \ , \qquad (6.5.2.20b)$$

and yields the bound,

$$\bar{r}_1 > \frac{3}{4}\alpha\bar{r}_2 \ , \text{ and } \ \alpha\geq 4. \qquad (6.5.2.21)$$

We conclude that the minimum saving occurs when $\alpha=4$, implying that the preconditioning must reduce the number of iterations by at least a third. The preprocessor requires $2(2w-1)^2$ hex cells in the reduced hex arrays, and 3w-1 cells for the modified matrix vector arrays which is a considerable increase. Consequently significant saving must occur in the CIA. Putting $r_2=w$ and $\alpha=8$ saves enough hardware to cover pre-processor costs and demands that after preconditioning only $\frac{1}{6}\bar{r}_1$ iterations are required for convergence. The speed-up can be maximised by adopting the 'bag-of' approach replacing (6.5.2.11b) with,

$$T = 2N + r(4p-3) \ . \qquad (6.5.2.22)$$

Then by re-evaluating (6.5.2.11c), (6.5.2.12) and noting that if,

$$\frac{\bar{r}_2}{2}\left\{\frac{3N}{r_2} + (12p-11) + \frac{c_2}{r_2}\right\} > (3N+c) + \frac{\bar{r}_2}{2}\left\{\frac{2N}{r_2} + (4p-3)\right\}$$

or,

$$\bar{r}_2 > 6r_2 \ , \qquad (6.5.2.23)$$

then a speed-up of the compressed pipelined form is guaranteed (for N sufficiently large).

EXAMPLES OF PRECONDITIONING

Below are some experimental results from Evans [83d] which provide realistic estimates of $\overline{r_1}$ and $\overline{r_2}$ for the discussions in this section. Two problems are considered.

I. LAPLACE

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$$

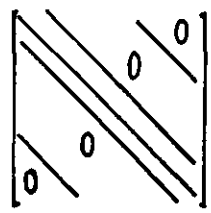
over the unit square with boundary conditions

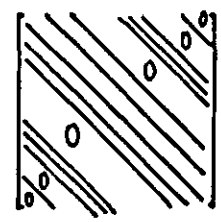
$$u(0,y) = u(1,y) = 0, \quad u(x,1) = 0 \; u(x,0) = 1$$

II. BIHARMONIC

$$\frac{\partial^4 u}{\partial x^4} + 2\frac{\partial^4 u}{\partial x^2 \partial y^2} + \frac{\partial^4 u}{\partial y^4} = 0$$

again over the unit square with values u and $\frac{\partial^2 u}{\partial^2 n}$ specified along boundaries, n is the direction of the outward normal producing striped matrix structures suitable for incomplete methods.

I. 

II. 

Both are positive definite, sparse and ill conditioned and provide important preconditioning theory tests.

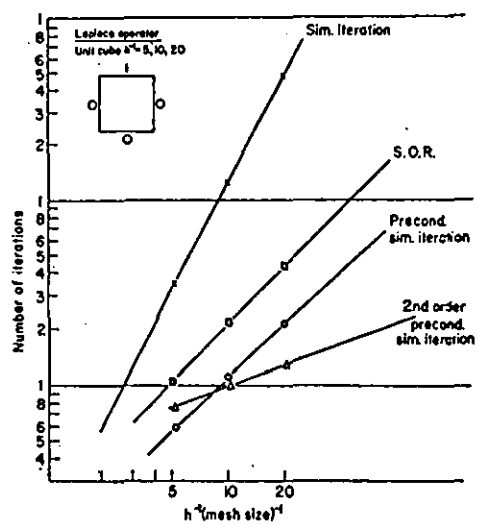


FIGURE 6.5.6: Preconditioning for Laplace equation

Problem	Grid Size h^{-1}	Precond-itioning Parameter ω	P condition Number P	Preconditioned Simult.Displ. Method	2nd Order Pre-Conditioned Richardson Method	2nd-Order Pre-Conditioned Chebyshev Method
				Number of Iterations		
Laplace operator	5	0	9.47	35	8	15
	10	0	39.86	120	32	30
	20	0	161.47	480	65	59
	5	1.3	1.65	8	8	7
	10	1.6	2.81	11	10	8
	20	1.8	5.42	21	13	14
Biharmonic operator	7	0	48.49	142	17	23
	10	0	286.44	290	40	41
	20	0	1916.21	1432	124	120
	7	1.4	4.33	18	8	8
	10	1.6	13.39	54	15	14
	20	1.8	104.28	174	40	36

TABLE 6.5.1: p-condition and number of iterations for 5×10^{-6} accuracy with basic (ω=0) and optimal preconditioning

Problem	Grid Size	Precon-ditioning parameter ω	P-condition number P	Preconditioned gradient method	Preconditioned Conjugate Gradient Method
				No.of iterations	No.of iterations
Laplace	5	0	9.47	32	7
	10	0	39.86	142	19
	20	0	161.47	258	37
	5	1.3	1.65	8	6
	10	1.6	2.81	12	8
	20	1.8	5.42	17	11
Biharmonic	7	0	48.49	142	17
	10	0	286.44	290	40
	20	0	1916.21	>1300	124
	7	1.4	4.33	18	8
	10	1.6	13.39	54	15
	20	1.8	104.28	174	50

TABLE 6.5.2: p-condition and number of iterations for 5×10^{-6} accuracy for steepest descent and conjugate gradient methods

Method	h^{-1} mesh size	Basic method			Sparse Elimination Preconditioned Method		
		Accel.factors α	β	No.of iterations	Accel.factors α	β	No.of iterations
1st order simultaneous displacement method	10	1.0	0	120	1.18	0	15
	20	1.0	0	120	1.45	0	33
2nd order Richardson method	10	1.53	0.53	32	0.98	1.04	12
	20	1.73	0.73	65	0.95	1.05	30
2nd order Chebyshev method	10			30			12
	20			59			30

TABLE 6.5.3: Results for a sparse elimination algorithm on the Laplace problem with unit square (matrices of order 99 and 361 respectively)

We now consider the Gauss-Seidel preconditioned form in (6.1.17) which produces the following algorithm:

$$\left.\begin{array}{ll} \text{STEP (i):} & \text{compute} \\ & \text{a)} \quad (I+L)b=v \\ & \text{b)} \quad M_1=L^2 \text{ and } M_2=(I+L)u \\ & \text{c)} \quad \text{Form } M=M_1+M_2 \\ \text{STEP (ii):} & u^{(i+1)}=Mu^{(i)}+v \\ & \text{IF NOT converged THEN GOTO step (ii)} \end{array}\right\} \qquad (6.5.2.24)$$

Again step (i) forms a preprocessor with step (ii) the CIA. The preprocessor is shown in Fig.(6.5.5) which is a delayed matrix vector array and a normal hex with two problem instances interleaved as shown in Fig. (6.5.7).

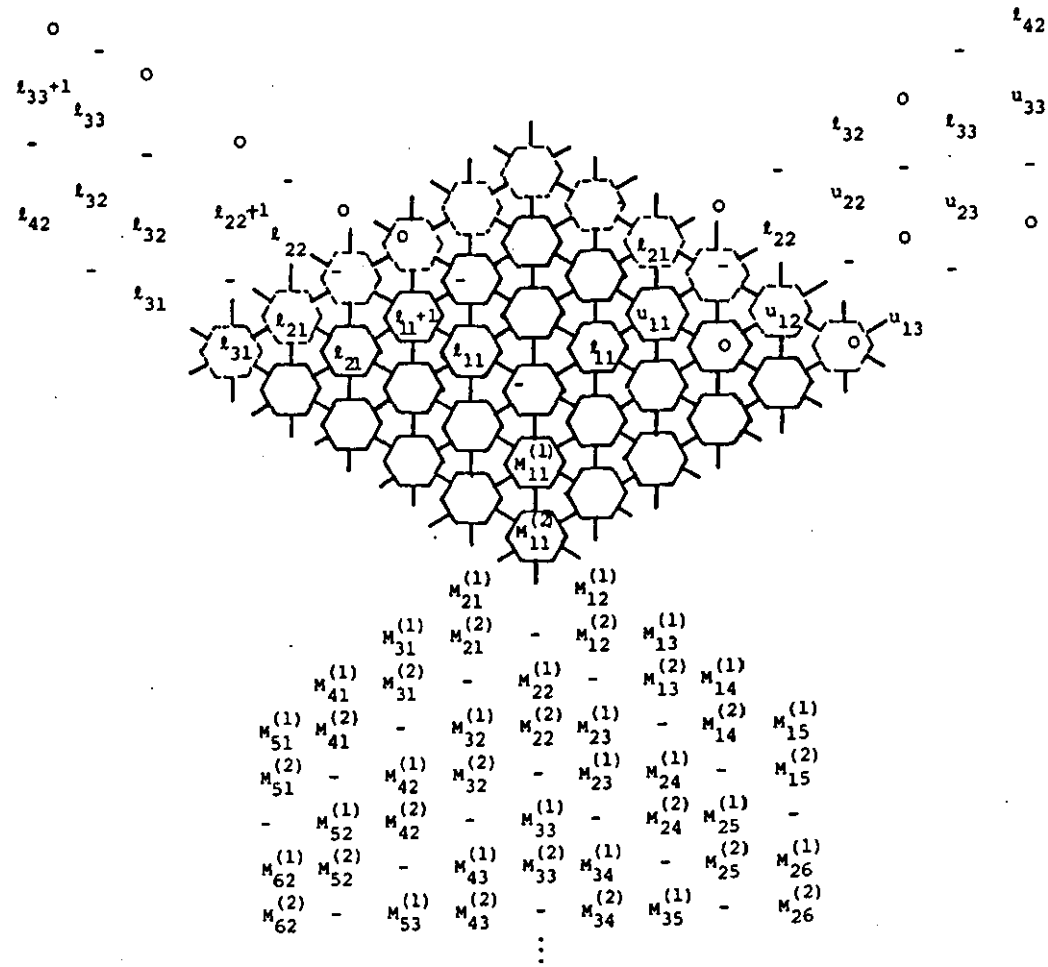


FIGURE 6.5.7: Hex input for preconditioned Gauss-Seidel

An extra boundary of cells is added with a special function of simply adding the interleaved results of M_1 and M_2 to form M and which contributes only a single cycle to the preprocessor delay. The use of the L and U matrices in different products compels the unidirectional flow of the Jacobi preprocessor to be discarded losing a good systolic feature. However the Gauss-Seidel method produces superior area trade-offs when the speed-up analysis is applied. The latency of the preprocessor is $c\alpha w+2$ where the cost of reformatting is ignored, where the hex requires $(w+1)^2$ cells and the matrix vector q cells (for q subdiagonals in L, and q=p in problem I). Notice that only the lower triangular part of A, equation (6.1.1), is squared producing a preconditioned matrix with bandwidth $w_1=w+q-1=3p-1$ for problem I. But the latency of each CIA array is controlled by the upper triangular portion of A, so adjusting for the delayed matrix-vector computation yields,

$$T_4 = 3N + r(3p-2) + c \ , \tag{6.5.2.25}$$

which yields the speed-up relation,

$$\frac{\bar{r}_1}{\bar{r}_2} > \frac{\left\{\frac{3N}{r_2} + (3p-2) + \frac{c}{r_2}\right\}}{\left\{\frac{2N}{r_1} + 2(p+1) - \frac{1}{r_1}\right\}}$$

or

$$\frac{\bar{r}_1}{\bar{r}_2} > \frac{3}{2}\left(\frac{r_1}{r_2}\right) \tag{6.5.2.26a}$$

for N sufficiently large.

The saving in hardware is computed simply, by,

$$S = \alpha r_2 w - r_2(w+q-1) = r_2[(\alpha-1)w+q-1] \ , \tag{6.5.2.26b}$$

for $r_1=\alpha r_2$ and produces the bound,

$$\bar{r}_1 > \frac{3}{2}\bar{r}_2 \ , \text{ for } \alpha=1 \ , \tag{6.5.2.26c}$$

which achieves a saving of $r_2(q-1)$ cells. Also notice that when $\alpha=\frac{w-q+1}{w}$, $s=0$. Consequently $\alpha\geq 1$ guarantees a saving, and if $r_2>w$ preprocessor hardware can be offset against CIA reductions. These results confirm that the Gauss-Seidel method saves more hardware than the Jacobi method when both have the same number of iterations $\bar{r}_2$ with respect to the unpreconditioned schemes. But by the fact that the Gauss-Seidel method uses the most recently computed u_j values it attains a faster convergence rate than the Jacobi method. Hence the bounds in (6.5.2.26) are more easily satisfied, and it becomes possible to scale the computation time so that both methods compute at approximately the same speed, but with the former scheme using considerably less hardware. Applying the 'bag of' principle yields even more savings.

6.6 A FAST ARRAY FOR SOLUTION OF TRIDIAGONAL LINEAR SYSTEMS

To complete this chapter we consider the fast systolic solution of (6.1.1) where A is an n×n tridiagonal matrix. The design draws together the main themes of double pipes, QI permutations, block partitioning, and incomplete arrays discussed in this and previous chapters.

A simple Gaussian elimination array can be derived easily from Fig.(6.4.1) and adopts a standard matrix vector input format. The array is shown in Fig.(6.6.1) and consists of three cells, a boundary cell to the left and two simple inner product cells to the right, augmented with a row interchange facility (for nearest neighbour pivoting). The boundary cell decides on the next pivot row setting a flag c to indicate interchanges and also computes a multiplier (m) to update the adjacent (non-pivot) row. Snapshots of the array operation

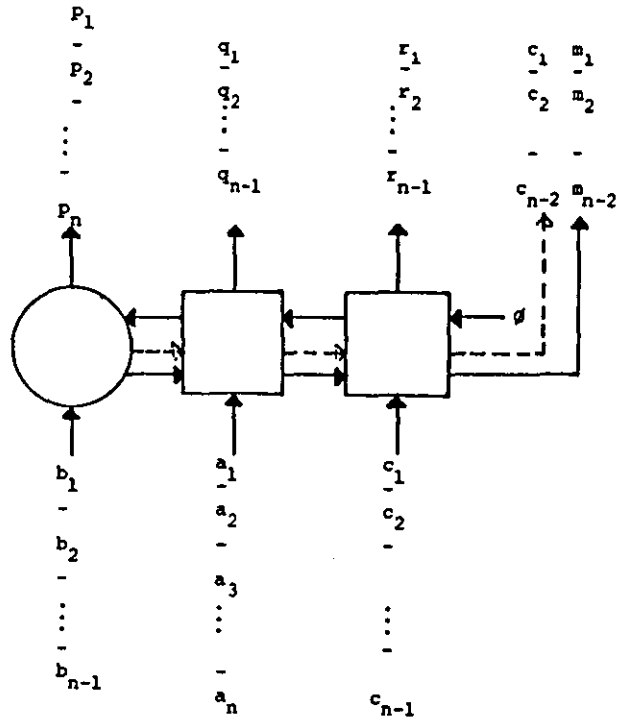
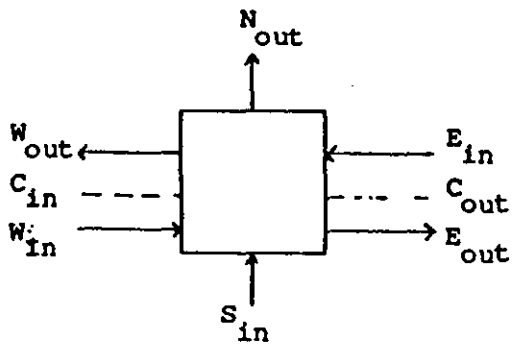


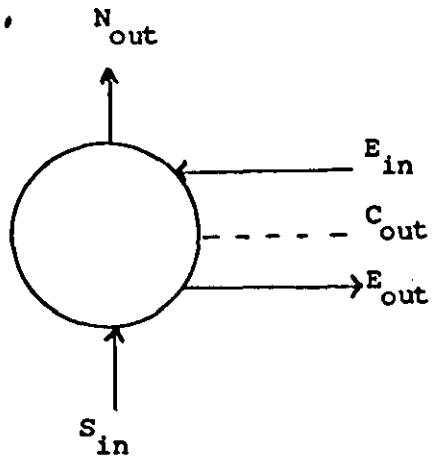
FIGURE 6.6.1: General tridiagonal Gaussian elimination

are shown in Fig.(6.6.2) and operates according to the following cell definitions.



Modifier cell

N_{out}-rold$_1$, W_{out}-rold$_2$
S_{in}-r_2, E_{in}-r_1
E_{out}-mold, C_{out}-cold
W_{in}-m , C_{in}-c
IF c=1 THEN
  {t1:=r2, t2:=r1}
ELSE
  {t1:=r1, t2:=r2}
rold$_2$:=t2-m*t1
rold$_1$:=t1
cold:=c
mold:=m



Boundary cell

N_{out}-P_{old_1}, E_{out}-m, C_{out}-c

E_{in}-P_1, S_{in}-P_2

$\alpha = P_1/P_2$, $\beta = P_2/P_1$

F=α-β

IF (F<O) AND (SIGN(P1) ⊕ SIGN(P2)) THEN
{c=1, m=α, Pold=P_2}
ELSE
{c=O, m=β , Pold=P_1}

REMARK: SIGN(x) = { O if x +ve ; 1 if x -ve }

and F<O and sign(x) can be simply noted by checking the sign bits of the numbers in question. The IF condition is simply a combinational logic expression with a negligable propagation delay.

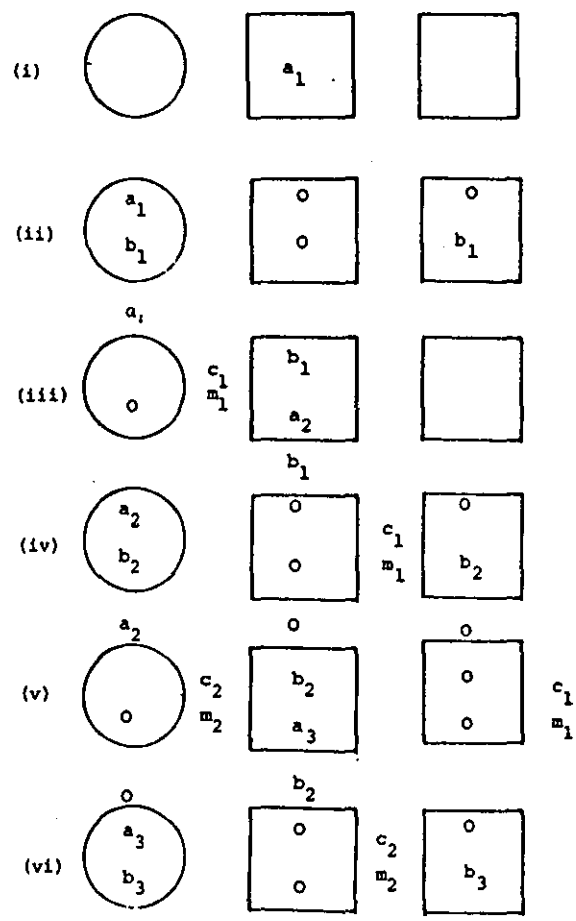


FIGURE 6.6.2: Snapshots of symmetric elimination

The computation of the boundary cell is easily justified as follows.

First, we must compute c and m where,

$$c = \begin{cases} 1 \text{ iff } abs(p_2) > abs(p_1) \\ 0 \text{ otherwise} \end{cases}$$

and

$$m = \begin{cases} \dfrac{p_2}{p_1} & \text{when } c=0 \\ \dfrac{p_1}{p_2} & \text{when } c=1 \end{cases}$$

Now,

$$F = \frac{p_1}{p_2} - \frac{p_2}{p_1} = \frac{p_1^2 - p_2^2}{p_2 p_1}$$

consequently, if,

$$\left.\begin{array}{l} p_1 \text{ - ve and } p_2 \text{ + ve} \\ \text{or} \\ p_1 \text{ + ve and } p_2 \text{ - ve} \end{array}\right\} F<0 \text{ iff } p_2^2 - p_1^2 < 0 \Rightarrow |p_1| > |p_2|$$

and when,

$$\left.\begin{array}{l} p_1 \text{ - ve and } p_2 \text{ - ve} \\ p_1 \text{ + ve and } p_2 \text{ + ve} \end{array}\right\} F<0 \text{ iff } p_1^2 - p_2^2 < 0 \Rightarrow |p_2| > |p_1|$$

It follows that,

$$(F<0) \text{ AND } (\text{sign}(p_1) \oplus \text{sign}(p_2)) = 1 \text{ for } m = \frac{p_2}{p_1} \text{ and } c=0$$

and

$$(F<0) \text{ AND } (\text{sign}(p_1) \odot \text{sign}(p_2)) = 1 \text{ for } m = \frac{p_1}{p_2} \text{ and } c=1,$$

requiring the cell structure in Fig.(6.6.3).

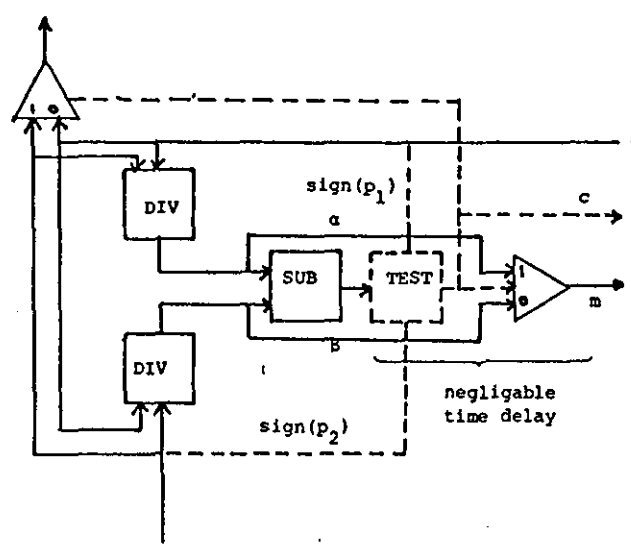


FIGURE 6.6.3: Boundary cell organisation

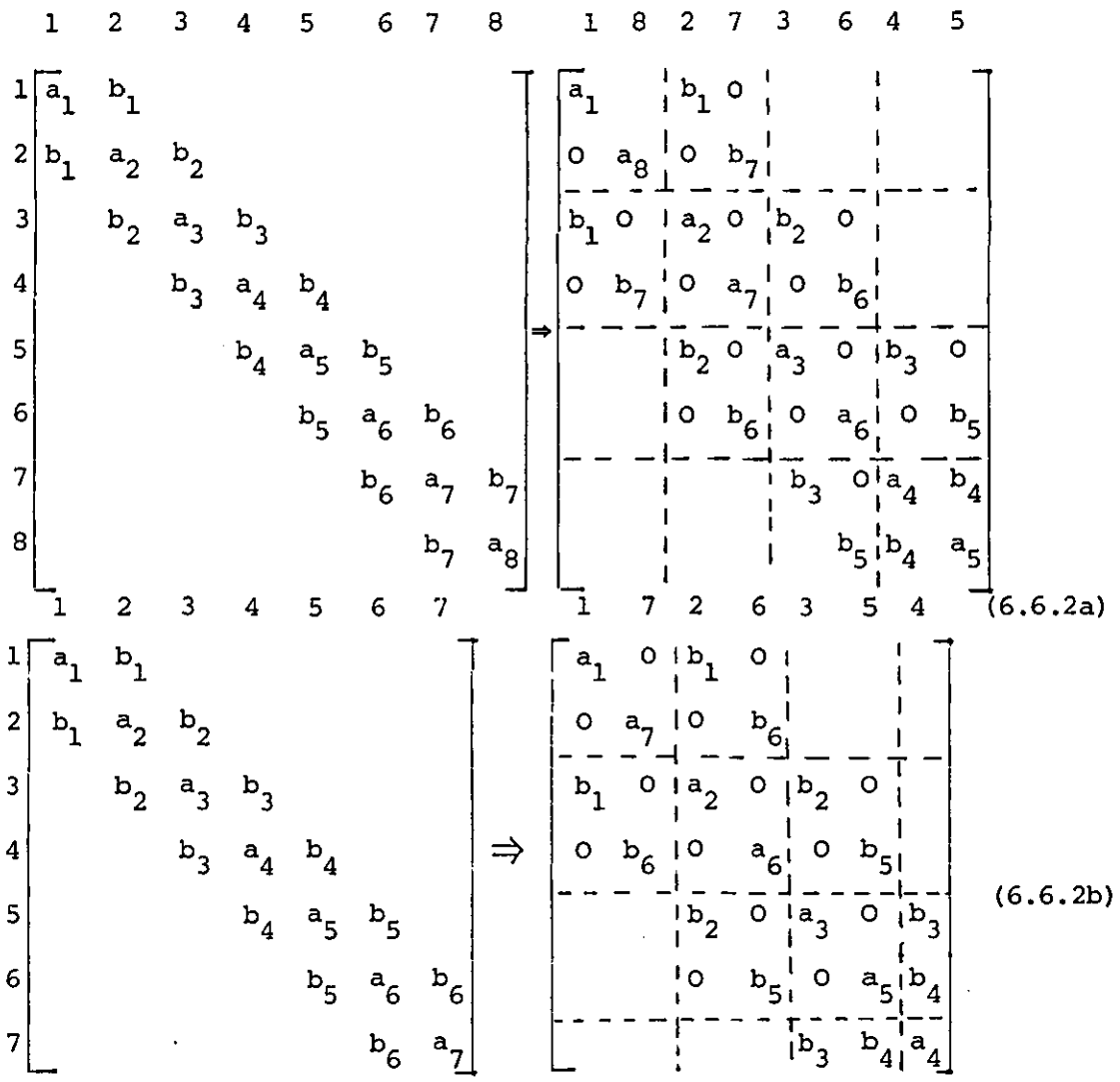
This ensures that both the boundary and modifying cells have a basic cycle time equivalent to a single inner product step. Also, with no interchanges the array generates only two columns of output data, but when interchanges do occur three outputs are needed as the second super-diagonal fills in. Hence the eliminated upper triangular matrix U takes the general form:

$$U = \begin{bmatrix} p_1 & q_1 & r_1 & & & \\ & p_2 & \ddots & \ddots & & \\ & & \ddots & \ddots & r_{n-2} & \\ & & & \ddots & q_{n-1} & \\ & & & & p_n & \end{bmatrix} \qquad (6.6.1)$$

The c_i and m_i, i=1(1)n-1 move rightwards eventually leaving the array to be stored (for identification of interchanges) or can be used directly to modify the righthand side of (6.1.1), but requires the addition of an extra modifier cell to Fig.(6.6.1).

The array timing is trivially observed to be $T=2n+k_1$ with k_1 a constant derived from the delay of the first output, and from Fig. (6.6.2) $k_1=2$.

Next we apply the QI-permutation to give the problem an implicit 2×2 block structure. For example, when n=8 and n=7 the tridiagonal form of a symmetric matrix becomes:



(Notice that the permutation produces a 2×2 block symmetric matrix). The distribution of zeroes in the above forms allows implicit block calculations to be performed using a double pipe version of Fig.(6.6.1) where the essential idea is to compute the elimination by modifying rows starting at the 1st and last row of the unpermuted form and moving inwards to the center. (Hence the similarity with QI methods). Figs. (6.6.4) and (6.6.5) illustrate the data flow for the odd and even cases, and reveals the starting input formats:

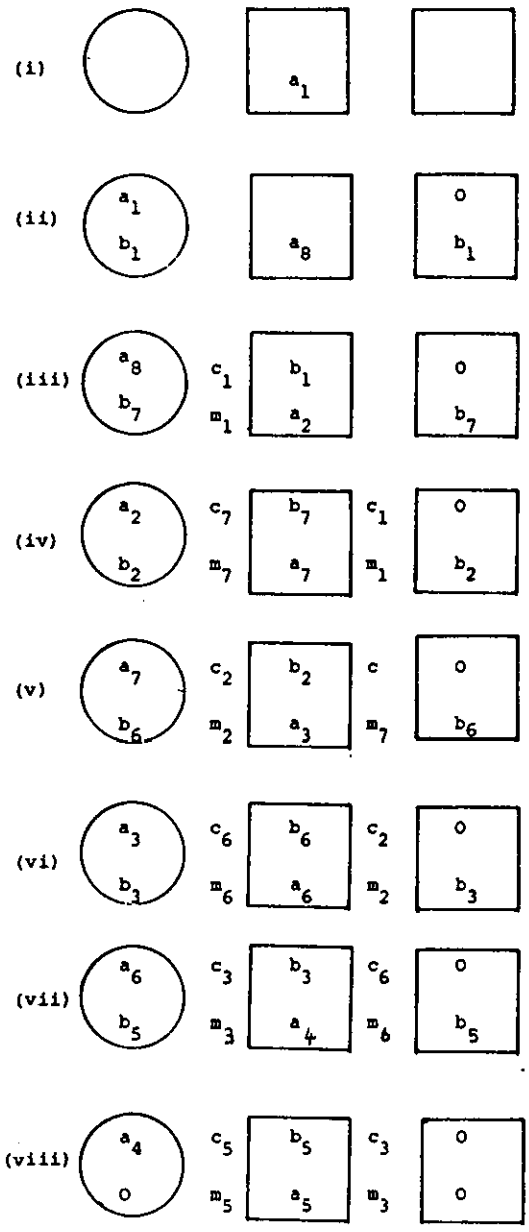


FIGURE 6.6.4

Implicit block symmetric elimination (n=even)

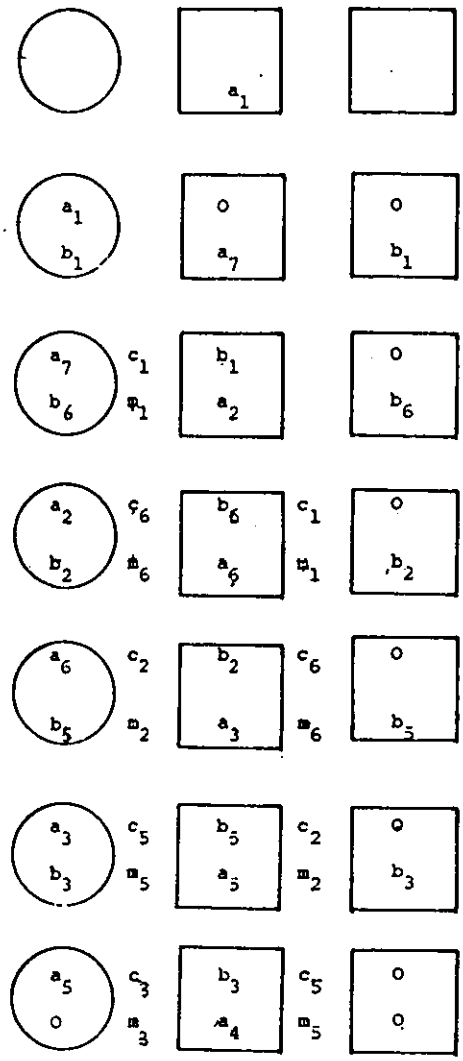
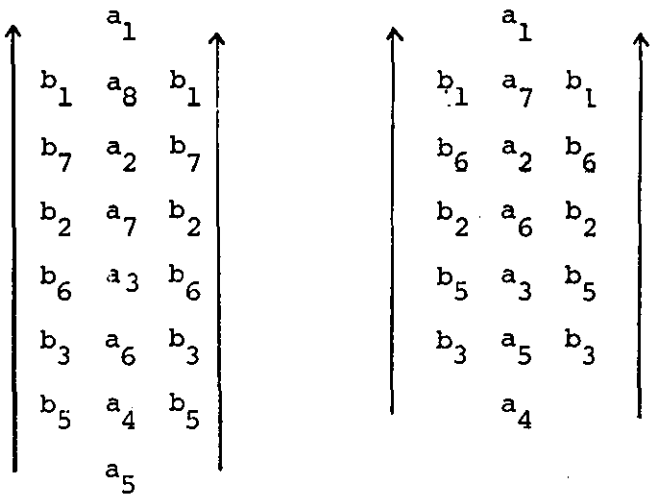


FIGURE 6.6.5

Implicit block symmetric elimination, (n=odd)



The data is size n and as everything else is unchanged apart from dataflow, it follows that the array computation time is T=n+2, and T=n+3 if a cell is added for the righthand side modification. Observe that nearest neighbour pivoting is still possible due to the sparsity pattern in (6.6.2). When the computation reaches the bottom right corners of the permuted matrices the zero pattern breaks down as the two interleaved eliminations collide and interfere with one another. Notice that the input data pattern above omits the b_4 values to avoid interference, and this introduces an incomplete calculation as the array only approximates the matrix U in (6.6.1).

The additional modifications to the array output for the even case (n=8) can be summarized as follows. On step (vi), (c_3,m_3) is computed by the boundary cell eliminating b_3. On the next cycle (b_3,a_4) are modified leaving $(0,b_4)$ to be updated to complete the elimination step. Likewise on step (vii) (c_5,m_5) are evaluated and eliminates b_5, and on the next cycle (b_5,a_5) are modified leaving $(0,b_4)$ to be updated using (c_5,m_5). Thus, when the array computation is finished we must perform these two additional modifications to produce a complete

elimination of (6.6.1). Since all the interchange and multiplier data is available this presents no problem. For purposes of argument the fixing procedure produces the 2×2 block form,

$$\begin{bmatrix} \bar{a}_4 & b_4 \\ b_4 & a_5 \end{bmatrix}$$

and b_4 is eliminated using the correct multipliers, (b_4, a_5) must be updated and the righthand side part modified.

REMARK: For simplicity we have assumed that no interchanges occur, the argument is similar if they do, or when n is odd.

We argue that the saving of n cycles in the elimination, outweighs the five additional cycles (including multiplier evaluation) which is negligible and can be delegated to the host machine.

Notice that this fixing strategy leaves a 2×2 block in the bottom right corner, and by adding an extra two modifications can be converted to an upper triangular block giving the permuted form of (6.6.1) the appearance,

$$\bar{U} = \left[\begin{array}{cc|cc|cc|} p_1 & 0 & q_1 & 0 & r_1 & 0 \\ 0 & p_8 & 0 & q_7 & 0 & r_7 \\ \hline & & p_2 & 0 & q_2 & 0 & r_2 & 0 \\ & & 0 & p_7 & 0 & q_6 & 0 & r_6 \\ \hline & & & & p_3 & 0 & q_3 & 0 \\ & & & & 0 & p_6 & 0 & q_5 \\ \hline & & & & & & p_4 & 0 \\ & & & & & & 0 & p_5 \end{array}\right] \qquad (6.6.3)$$

for n=8. The solution of (6.1.1) can now be written as a backward substitution,

$$\bar{U}\,\tilde{u} = \bar{d}\ , \qquad (6.6.4)$$

where $\tilde{u}$ is the permuted unknown vector and $\bar{d}$ the modified and permuted righthand side vector. As (6.6.3) has a null first subdiagonal we can apply the double pipe substitution array of Robert & Tchuente [84] as illustrated in Fig.(4.2.1). This array requires 6 ips cells and a time T=n+3+1=n+4 cycles to solve (6.6.4), and by noting that the bottom tier corresponds to zero subdiagonals can be compacted to use only 3 cells. Furthermore, the elimination and substitution by virtue of the output ordering have to be computed sequentially giving a total time,

$$T = (n+3) + (n+4) + 7 = 2n+14 , \qquad (6.6.5)$$

which includes the time for fixing the incomplete elimination.
In contrast a straightforward complete elimination requires the time

$$T = (2n+3) + (2n+3) = 4n+6 , \qquad (6.6.6)$$

using an elimination as shown in Fig.(6.6.2) and a traditional substitution array. Thus a factor of two speed-up and increase in cell efficiency results from our double piped, implicit block structured incomplete array, provided some calculations are performed by the host machine. This reliance on the host is acceptable in this case because the output of the eliminator cannot be pipelined directly into the substitution array, and for the sake of <10 ips the saving is dramatic.

6.7 SUMMARY

In this chapter, the systolic design for preconditioned iterative procedures was developed. The global design consisted of two component arrays, a preconditioning preprocessor, and a cascaded iterative array (CIA) for pipelined iterations. Three types of preprocessor were developed suitable for both implicit and explicit types of preconditioning.

For explicit methods we considered the problem of repeatedly

squaring a banded matrix (M) using a pipelined arrangement which permitted the evaluation of the sequences M^{2^i}, M^i and accumulation of the Neumann expansion. A modified (reduced) hexagonal array for matrix products was described which compressed both input and output into a single direction and reduced the communication bandwidth of the systolic array compared with the usual matrix product arrangement. The increase in computation time for the new array related to the matrix bandwidth rather than its order. A pipeline component for power generation was then constructed by the addition of a preprocessor which accepted a single copy of the input matrix (M) and produced separate duplicate copies providing a neat expansion of the host interface. A systolic preprocessor of this form will be useful in the future for similar roles in other problems, and we suggested optical preprocessing as a cheap way of producing large array inputs from a relatively small number of host connections, using the properties of waveguided and low signal interaction to avoid non-planar wiring.

For implicit preconditioning methods we developed preprocessors based on the so-called incomplete techniques applied to the solution of linear systems derived from certain 2-D and 3-D partial differential equations. These techniques control the fill-in associated with the solution process by selecting certain diagonals to be retained in the calculations, to yield approximate answers. The idea of a sequence of arrays with varying hardware providing an exact solution and a range of approximate results led to a method of array compaction. The essential concept being the identification of primary neutral cells which played no part in the calculation, which in turn identified secondary neutral cells that could be replaced by synchronising delay

cells. The technique was applied to matrix factorisation and triangularisation methods giving rise to optimal area arrays and the Systolic Incomplete Factorisation (SIF) and Elimination (SIE) algorithms respectively. For the model 2-D and 3-D problems considered, the SIF design required cells proportional to the semi-bandwidth of the non-zero diagonals plus the diagonals retained for fill-in. This is compared with cells proportional to the semi-bandwidth of the smallest band enveloping all non-zero diagonals in the original methods. A similar result holds for the SIE method, and in general a cell reduction of >75% was observed. The approximate solution errors of the systolic methods were generally higher than the original schemes with the SIF better than the SIE methods. The variations in approximations being attributable to a balanced shifting process introduced to allow optimal array compaction, which become unbalanced in the SIE algorithm. The compact arrays used considerably less area than uncompacted forms making them more practical to construct. Future consequences for systolic applications are clear. We can define a sequence of designs which trade accuracy against circuit area to produce fast, economic, and practical parallel devices for quickly approximating solutions to given problems. Such low cost devices could make suitable add-ons for existing architectures.

The preconditioning preprocessors were then considered in conjunction with cascaded iterative arrays (CIA's) to assess both speed-up and area savings over existing iterative schemes. It was found that most of the methods considered reduced the CIA to a series of simple matrix vector arrays rather than forward substitution and matrix vector used by the unpreconditioned Gauss-Seidel form. Relationships

between the convergence rates of unpreconditioned and preconditioned schemes and the finite size of their CIA's were established and conditions for speed-ups and area savings derived. We concluded that for reasonable preconditioners enough hardware could be saved to account for the preprocessors while still achieving a speed-up. The 'bag of' methods which performed preconditioning and iteration sequentially proved to be the fastest variation of all and the preconditioned Gauss-Seidel gave the best area saving.

Finally we described a fast tridiagonal solver which illustrated how double pipes, block structuring by QI permutations, and incomplete methods could be combined to derive fast area efficient designs for more general problems.