This item was submitted to Loughborough's Research Repository by the author.
Items in Figshare are protected by copyright, with all rights reserved, unless otherwise indicated.

## Systolic algorithms and applications

PLEASE CITE THE PUBLISHED VERSION

## PUBLISHER

© Chunru Wan

LICENCE

CC BY-NC-ND 4.0

REPOSITORY RECORD
Wan, Chunru. 2019. "Systolic Algorithms and Applications". figshare. https://hdl.handle.net/2134/10479.

## This item was submitted to Loughborough University as a PhD thesis by the author and is made available in the Institutional Repository (https://dspace.Iboro.ac.uk/) under the following Creative Commons Licence conditions.

## (c) creative

C O M M O N S D E E D

Attribution-NonCommercial-NoDerivs 2.5

You are free:

- to copy, distribute, display, and perform the work

Under the following conditions:

BY Attribution. You must attribute the work in the manner specified by the author or licensor


Noncommercial. You may not use this work for commercial purposes.

No Derivative Works. You may not alter, transform, or build upon this work

- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder

Your fair use and other rights are in no way affected by the above.

This is a human-readable summary of the Leqal Code (the full license).
Disclaimer $\left.{ }^{[ }\right]$

For the full text of this licence, please go to:
http://creativecommons.org/licenses/by-nc-nd/2.5/

Pilkington Library

$\qquad$
Accession/Copy No.
040129612.


# SYSTOLIC ALGORITHMS AND APPLICATIONS 

by Chunru Wan

March 1996

A doctoral thesis submitted in partial fulfilment of the requirements for the Degree of Doctor of Philosophy in

Department of Computer Studies
Loughborough University of Technology
Loughborough, Leicestershire, LE11 3 TU, UK



To my darling wife
Zhang Yu
for her love, support and encouragement

## ABSTRACT

The computer performance has been improved tremendously since the development of the first allpurpose, all electronic digital computer in 1946. However, engineers, scientists and researchers keep making more efforts to further improve the computer performance to meet the demanding requirements for many applications.

There are basically two ways to improve the computer performance in terms of computational speed. One way is to use faster devices (VLSI chips). Although faster and faster VLSI components have contributed a great deal on the improvement of computation speed, the breakthroughs in increasing switching speed and circuit densities of VLSI devices will be difficult and costly in future. The other way is to use parallel processing architectures which employ multiple processors to perform a computation task. When multiple processors working together, an appropriate architecture is very important to achieve the maximum performance in a cost-effective manner. Systolic arrays are ideally qualified for computationally intensive applications with inherent massive parallelism because they capitalize on regular, modular, rhythmic, synchronous, concurrent processes that require intensive, repetitive computation.

This thesis can be divided into three parts. The first part is an introductory part containing Chap. I and Chap. 2. The second part, composed of Chap. 3 and Chap. 4 concerns with the systolic design methodology. The third part deals with the several systolic array design for different applications.

The systolic array architecture in parallel processing came as a product of certain environment, the means and the background knowledge for its realisation. The needs can be outlined as the everincreasing tendency for faster computations, especially in areas like real-time signal processing and large-scale scientific computation. The means were provided by the remarkable advances in VLSI technology and computer aided design (CAD). Finally the background includes the applications of parallel processing and the design of parallel computers of various architectures including pipelined vector processors, SIMD architectures, array processors, MIMD architectures and so on.

Ever since H.T. Kung proposed the systolic architecture in 1978, its elegant solution to demanding problem and its potential performance have attracted great attention. Systolic arrays have find applications in matrix arithmetic, signal processing, image processing, solution of differential
equations, data structure, graph algorithms, computer aided design (CAD), and so forth. Many systolic algorithms have been designed for a great diversity of areas. Few problems resist attack from systolic arrays. A lot of efforts have been made to implement the systolic array in a VLSI chip, in a programmable processor array, or in a reconfigurable systolic array. While systolic arrays were originally used for fixed or special purpose architecture, the systolic array concept has been extended to general-purpose SIMD and MIMD architectures.

One of the important design problems in systolic array processing is the development of systematic methodology for transforming an algorithm represented in some high-level constructs into a systolic architecture specified by the timing of data movement and the interconnection of processing elements such that the design requirements can be satisfied. Among a number of systolic design methodologies, two types of systematic design methodologies have been studied in this thesis. The first method starts from the RIA (regular iterative algorithm) representations of a given algorithm, extracts the data dependency from the RIA algorithm, and then map the algorithm into a systolic array by using algebraic approach. This method is applied to the matrix-matrix multiplication problem. We have obtained 19 different systolic designs for the matrix multiplication problem. A detailed performance analysis for all the 19 designs is presented. Based on the performance measures, we can choose the right design for given requirements. The other design method we have studied uses the SFG (signal flow graph) of a given algorithm to design systolic arrays through graphic mapping and retiming. An algorithm is first represented by a DG (dependence graph). Then the DG is mapped into an SFG by a graph projection. Cut-set retiming procedure is then applied to the obtained SFG to derive a regular and temporally localized SFG, from which a systolic array design is immediately available. Systolic arrays for matrix triangularisation, including both LU decomposition and QR decomposition, have been designed using this method.

A number of systolic arrays have been designed in this thesis for different application areas. Firstly, we designed a systolic array for linear and inverse matrix system. The systolic array combines LU (or QR) decomposition, backward and forward substitution, and matrix multiplication into a trapezoidal array. The array can be used to solve a variety of linear systems and other problems such as matrix multiplication. When QR decomposition is used, the systolic array can solve linear least square problems or perform pseudo-inverse computation. Then a systolic array is designed to compute the Schur complement which often occurs in the domain decomposition method and other problems in linear algebra such as the numerical solution of elliptic partial differential equations and sparse linear system. A systolic array architecture has been designed and simulated for QR decomposition of block
structured sparse systems which find applications in areas such as photogrammetry, Doppler radar positioning, geodetic problems, cluster analysis and pattern matching. A novel systolic array is derived for computing matrix inversion by Newton Iteration which has a good feature that some simper initial approximations can always guarantee a convergence. A systolic array architecture for Capon's DOA estimation in sonar and radar applications is derived based on the QR decomposition technique. Finally a novel linear systolic array for conjugate gradient (CG) atgorithm is proposed to solve linear system efficiently.

Systolic array architecture is an formidable approach to exploiting massive parallelism with a minimum communication overhead. Featuring modularity, regularity and local communication, systolic arrays are amendable to VLSI implementation. VLSI technology advances, parallel processing and demanding scientific and engineering applications have made the systolic array architecture a leading approach for handling computationally intensive applications. Systolic arrays hold great promise to be a pervasive form of massive parallel processing in demanding applications.

## KEYWORDS

Systolic Array; Systolic Algorithm Design; Parallel Processing; Computer Architectues; Matrix Computations; Linear Systems; Least Squares; Systolic Array Applications

## ACKNOWLEDGEMENT

I wish to acknowledge the very useful, stimulating supervision and guidance of Prof. D.J. Evans of the Department of Computer Studies, Loughborough University of Technology (LUT). I also greatly appreciate his effort of providing and seeking financial support for me during the period I stayed in LUT. His kindness and friendliness also made my stay with him in the department a very happy experience. I would also like to thank him for all the encouragement and patience he has given to me during my study.

I am grateful to Prof. J.W.R. Griffiths of the Department of Electrical and Electronic Engineering, LUT, for his supervision and support when I was staying in the department. His hard-working attitude gives me a deep impression. I am very sad that he passed away several months ago.

I would also like to thank Prof. C.F.N. Cowan of the Department of Electrical and Electronic Engineering, LUT for his supervision.

I would like to thank my parents and my sisters and brother for their support, inspiration and confidence since my early education. I am also indebted to my parents-in-law for all their support and encouragement.

Perhaps the deepest appreciation and thanks must go to my wife Yu for her patience, tolerance, understanding and love during my research and study as well as preparation of this thesis: Since the writing of this thesis was done, for the most part, in the spare time which is generally allocated to family life, the loss of a period of three years for her is no small contribution to this thesis. During this period, she took over all the housework including fixing those nice dinners everyday for us while she was also engaging in a fulltime job. Without her full support, this thesis would not be possible. Many thanks to her with my love.

## CONTENTS

1. INTRODUCTION ..... 1
1.1 Introduction .....  1
1.2 Overview of Computer System History .....  2
1.2.1 The Definition of Computers ..... 2
1.2.2 The Evolution of Computers ..... 3
1.2.3 The Varieties of Computers ..... 6
1.2.4 General Purpose Versus Special Purpose. ..... 8
1.3 Evolution of IC Technology ..... 9
1.3.1 Five Generations of ICs ..... 9
1.3.2 VLSI Technology Constraints ..... 11
1.3.3 VLSI Architecture Design Principles ..... 12
1.4 Parallel Computer Architectures ..... 14
1.4.1. Flynn's Taxonomy ..... 15
1.4.2 Duncan's Taxonomy ..... 16
1.4.2.1 Synchronous Architectures ..... 17
1.4.2.2 MIMD Architectures ..... 20
1.4.2.3 MIMD Paradigms ..... 21
1.5 Organization of the Thesis ..... 24
1.5.1 Part 1: Introduction ..... 24
1.5.2 Part 2: Systolic Array Design Methodologies ..... 25
1.5.3 Part 3: Systolic Array Architectures and Applications ..... 25
1.6 Summary ..... 26
2. OVERVIEW OF SYSTOLIC ARRAY ARCHITECTURES ..... 27
2.1 What is a Systolic Array? ..... 27
2.2 Why Systolic Architectures? ..... 30
2.2.1 The External Causes ..... 30
2.2.2 The Internal Causes ..... 31
2.2.2.1 Simple and Regular Design ..... 32
2.2.2.2 Concurrency and Communication ..... 32
2.2.2.3 Balancing Computation with I/O ..... 33
2.3 Classifications of Systolic Architectures ..... 34
2.3.1. Array Geometry Pattern. ..... 34
2.3.2 Application scope ..... 36
2.3.3 Implementation Level ..... 36
2.3.4 Granularity ..... 37
2.3.5 Data/Instruction Stream ..... 38
2.3.6 Design Restriction ..... 39
2.4 Applications of Systolic Arrays ..... 40
2.4.1 Numerical Applications ..... 41
2.4.2 Signal and Image Processing Applications ..... 42
2.4.3 Non-Numerical Applications ..... 43
2.5 Example Systolic Architectures ..... 44
2.5.1 Special Purpose Systolic Arrays ..... 44
2.5.2 General Purpose Systolic Arrays ..... 46
2.6 Systolic Array Design Methodologies ..... 51
2.7 Some Related Issues to Systolic Architecture ..... 53
2.7.1 Performance of Systolic Architecture Versus Other Parallel Architectures53
2.7.2 Area-Time Complexity Theory ..... 55
2.7.3 Systolic Arrays Versus Wavefront Arrays ..... 56
2.8 Summary ..... 57
3. SYSTOLIC ARRAY SYNTHESIS USING RIA
REPRESENTATIONS ..... 59
3.1 Introduction ..... 59
3.2 Basic Concepts and Definitions ..... 62
3.2.1 On the Exploitation of Parallelism in Algorithms ..... 62
3.2.2 Definition of Regular Iterative Algorithm. ..... 64
3.3 A Systolic Array Architecture Synthesis Procedure ..... 69
3.3.1 Processor Space and Iteration Space ..... 69
3.3.2 Obtaining a Schedule. ..... 71
3.3.3 Compatibility of Schedule and Iteration Space ..... 74
3.3.4 A Step-by-Step Synthesis Procedure ..... 75
3.4. Systolic Array Design for Matrix Multiplication ..... 77
3.4.1 Dense Matrix Multiplication ..... 77
3.4.2 Band Matrix Multiplication. ..... 83
3.5 Systolic Array Performance Analysis. ..... 85
3.5.1 The Definition of Systolic Array Performance Measures ..... 85
3.5.2 Performance Analysis of the Matrix Multiplication Systolic Arrays ..... 87
3.6 Definition of Systolic Array ..... 89
3.7 Summary ..... 91
4. SYSTOLIC ARRAY SYNTHESIS USING SFG REPRESENTATIONS ..... 101
4.1 Introduction ..... 101
4.2 Mapping DGs onto SFGs ..... 102
4.2.1 DG Design ..... 102
4.2.2 Signal Flow Graph (SFG) ..... 104
4.2.3 Mapping DG to SFG ..... 104
4.3 Mapping SFGs to Systolic Arrays ..... 107
4.3.1 Cut-Set Retiming Procedure ..... 108
4.3.2 Systolization Procedure ..... 110
4.4 A Step-by-Step Mapping Procedure. ..... 110
4.5 Systolic Array for Matrix Triangularization ..... 112
4.5.1 Mapping Algorithms to DGs ..... 112
4.5.1.1 LU Decomposition ..... 112
4.5.1.2 QR Decomposition ..... 113
4.5.2 Mapping DGs to SFGs ..... 117
4.5.2.1 LU Decomposition ..... 117
4.5.2.2 QR Decomposition ..... 119
4.5.3 Mapping SFGs Systolic Arrays. ..... 120
4.5.3.1 LU Decomposition ..... 120
4.5.3.2 QR Decomposition ..... 122
4.6 Discussions on the SFG Design Methodology ..... 124
4.6.1 Optimization of the Systolic Array Design ..... 124
4.6.1.1 Optimization in DG Design Stage ..... 125
4.6.1.2 Optimization in SFG Design Stage ..... 126
4.6.1.3 Optimization in Systolization Stage ..... 126
4.6.2 Non-Linear Assignment ..... 127
4.7 Summary ..... 128
4.8 Occam Programming Lists. ..... 129
4.8.1 Systolic Algorithm for LU Decomposition ..... 129
4.8.2 Systolic Algorithm for QR decomposition ..... 131
4.8.3 Systolic Algorithm for Square-Root-Free QR decomposition ..... 132
4.8.4 A Brief Introduction to Transputers and Occam ..... 135
4.8.4.1 Transputers ..... 135
4.8.4.2 Occam ..... 135
5. SYSTOLIC ARRAY FOR LINEAR AND INVERSE MATRIX SYSTEM ..... 142
5.1 Introduction ..... 142
5.2 Solution of Linear Systems and Matrix Inverse ..... 143
5.3 Systolic Array for Linear and Inverse Matrix Systems ..... 145
5.3.1 LU Decomposition ..... 145
5.3.2 Back Substitution ..... 147
5.3.3 Systolic array for linear system ..... 148
5.4 Least Squares Problems ..... 150
5.5 Possible Applications ..... 152
5.6 Performance Analysis ..... 154
5.6.1 The Right Side Linear System Solution. ..... 154
5.6.2 Faddeev's Algorithm. ..... 155
5.6.3 Matrix Inverse. ..... 156
5.7 A More Efficient Systolic Array for Linear System ..... 157
5.8 Summary ..... 158
5.9 Occam Program List ..... 159
5.9.1 Occam Program for the Systolic Architecture ..... 159
5.9.2 Numerical Examples ..... 162
6. SYSTOLIC ARRAY FOR SCHUR COMPLEMENT COMPUTATION ..... 163
6.1 Introduction ..... 163
6.2 Schur Component and Domain Decomposition ..... 164
6.3 Systolic Array for Schur Complement Computation ..... 167
6.3.1 Systolic Array for Linear and Inverse Matrix System (SALIMS) ..... 167
6.3.2 Systolic Array for Schur Complement Computation ..... 168
6.3.3 Computation of the Right Hand Side ..... 169
6.4 Solution of the Arrowhead Matrix System ..... 170
6.5 Two Possible Implementation Methods ..... 172
6.5.1 Internal Control Method ..... 172
6.5.2 External Control Method. ..... 173
6.5.3 The Comparison of the Two Methods ..... 174
6.6 Performance Analysis ..... 174
6.7 Summary ..... 176
7. SYSTOLIC ARRAY FOR QR DECOMPOSITION OF BLOCK STRUCTURED SPARSE SYSTEMS ..... 177
7.1 Introduction ..... 177
7.2 Block Structured Sparse System ..... 178
7.3 QR Decomposition of the $\mathrm{BS}^{3}$. ..... 180
7.4 Systolic Array Design ..... 182
7.4.1 Gentleman and Kung's Systolic Array for QR decomposition. ..... 182
7.4.2 Systolic Array for the Computations in Eq. (7.11) ..... 183
7.4.3 Systolic Array for the Computation in Eq. (7.13). ..... 185
7.4.4 The Overall Structure of the Systolic Array Architecture for $\mathrm{BS}^{3}$ ..... 186
7.4.4.1 The systolic array architecture ..... 186
7.4.4.2 The PE definition ..... 187
7.5 Simulation of the Systolic Algorithm ..... 188
7.5.1 Simulation Method. ..... 188
7.5.2 Simulation Result ..... 189
7.6 Summary ..... 190
8. SYSTOLIC ARRAY FOR MATRIX INVERSION BY NEWTON ITERATION ..... 192
8.1 Introduction ..... 192
8.2 Newton Method for Matrix Inversion ..... 193
8.2.1 The Algorithm for Iterative Matrix Inversion ..... 193
8.2.2 Initial Approximations for Special Matrices ..... 195
8.3 Systolic Array Building Blocks for the Newton Method ..... 197
8.3.1 Systolic Arrays for Matrix Multiplication. ..... 197
8.3.2 Systolic Array for Matrix Transposition ..... 199
8.4 Systolic Array for Iterative Matrix Inverse ..... 199
8.4.1 Overall Architecture ..... 199
8.4.2 Definitions of the Processing Elements ..... 202
8.4.3 An Example ..... 203
8.5 Modifications to the Basic Systolic Architecture ..... 206
8.5.1 Residual Matrix Evaluation. ..... 206
8.5.2 Initial Approximation Computation ..... 207
8.6 Summary ..... 208
9. A SYSTOLIC ARCHITECTURE FOR CAPON'S DOA ESTIMATION METHOD ..... 210
9.1 Introduction. ..... 210
9.2 Capon's Method for DOA Estimation ..... 211
9.2.1 The array and signals. ..... 211
9.2.2 Capon's Method ..... 212
9.3 Computational Tasks of Capon's Method ..... 214
9.3.1 Capon's Method Based on QR Decomposition Technique ..... 214
9.3.2 QR Decomposition. ..... 216
9.3.3 Forward Substitution to Compute ..... 217
9.3.4 Compute the Norm of a Vector ..... 217
9.3.5 The Total Computational Load for Capon's Method ..... 218
9.4 Systolic Array for Capon's Method ..... 219
9.4.1 Systolic Array for QR Decomposition ..... 219
9.4.2 Systolic Array for Forward Substitution ..... 220
9.4.3 Systolic Array for Vector Norm Computation ..... 220
9.4.4 Systolic Array for Capon's Method ..... 221
9.5 Occam Simulation ..... 224
9.5.1 Simulation Method ..... 224
9.5.2 Simulation Result ..... 225
9.6 Summary ..... 225
10. A LINEAR SYSTOLIC ARRAY FOR CONJUGATE GRADIENT METHOD ..... 227
10.1 Introduction ..... 227
10.2 Conjugate Gradient Method ..... 228
10.3 DG Design for CG Method ..... 231
10.3.1 Algorithm Modification ..... 231
10.3.2 Dependece Graph Design ..... 233
10.3.2.1 DGs for the Two CG Algorithms ..... 233
10.3.2.2 Modification to the DG for CG Algorithm 8. ..... 235
10.4 Systolic Array Design for CG Method ..... 237
10.4.1 Mapping the DG onto SFG ..... 237
10.4.2 Systolization of the SFG ..... 239
10.4.3 Systolic Arrays Design ..... 239
10.5 Preconditioned Conjugate Gradient (PCG) Method ..... 242
10.5.1 PCG Algorithm ..... 242
10.5.2 Systolic Array for PCG Method. ..... 244
10.6 Summary ..... 245
11. SUMMARY AND FURTHER RESEARCH ..... 247
11.1 Systolic Array Processing. ..... 247
11.2 Summary of the Thesis ..... 248
11.2.1 Overview of Computer Architecture and Systolic Array Processing ..... 248
11.2.2 Systolic Array Design Methodologies ..... 249
11.2.3 Systolic Arrays Design and Applications ..... 250
11.3 Further Research ..... 250
REFERENCES ..... 252

## LIST OF FIGURES

Figure 1.1. Past growth and future projections for computer performance . . . . . . . . . . . . 6
Figure 1.2 Integrated circuit complexity trends . . . . . . . . . . . . . . . . . . . . . . . . . 10
Figure 1.3 The four types of computer architectures (a) SISD; (b) SIMD; (c) MISD; (d) MIMD 15

Figure 1.4 High-level taxonomy of parallel computer architectures . . . . . . . . . . . . . . . 17
Figure 1.5 Pipelined vector parallel architecture. . . . . . . . . . . . . . . . . . . . . . . . . 18
Figure 1.6 Processor array architecture . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
Figure 1.7 Associative memory architecture . . . . . . . . . . . . . . . . . . . . . . . . . . 19
Figure 1.8 Systotic architecture . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
Figure 1.9 Distributed memory architecture . . . . . . . . . . . . . . . . . . . . . . . . . . 20
Figure 1.10 MIMD interconnection network topologies: (a) ring; (b); mesh; (c) tree; (d) hypercube; (e) tree mapped to a reconfigurable mesh

Figure 1.11 MIMD shared memory architectures: (a) bus interconnection; (b) crossbar; (c) multistage interconnection network. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21

Figure 1.12 MIMD/SIMD architecture . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
Figure 1.13 Dataflow architecture . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
Figure 1.14 Reduction architecture: (a) program; (b) demand token production; (c) result token production.22
Figure 1.15 Wavefront array architecture ..... 23
Figure 2.1 A typical systolic system ..... 29
Figure 2.2 Systolic array configuration geometries ..... 35
Figure 2.3 The Warp machine architecture ..... 48
Figure 2.4 iWarp architecture ..... 49
Figure 2.5 CAP architecture ..... 50
Figure 2.6 Speedup of $N$-processor parallel system over a single processor ..... 55
Figure 3.1 Y chart for transformational systems. ..... 60
Figure 3.2 Overview of RIA design methodology ..... 61
Figure 3.3 The index spaces of the RIAs in Examples 1-3 ..... 67
Figure 3.4 The DGs of the RIAs in Examples 1-3 ..... 68
Figure 3.5 The RDGs for RIAs in Examples 1-3 ..... 68
Figure 3.6 Matrix multiplication: (a) Index space; (b) DG; (c) RDG ..... 77
Figure 3.7 Systolic array for matrix multiplication: Version $1, \boldsymbol{U}^{\mathrm{T}}=\left[\begin{array}{lll}0 & 0 & 1\end{array}\right]$ ..... 94
Figure 3.8 Systolic array for matrix multiplication: Version 2, $\boldsymbol{U}^{\mathrm{T}}=\left[\begin{array}{lll}0 & 1 & 0\end{array}\right]$ ..... 94
Figure 3.9 Systolic array for matrix multiplication: Version $3, \boldsymbol{U}^{\mathrm{T}}=\left[\begin{array}{lll}1 & 0 & 0\end{array}\right]$ ..... 94
Figure 3.10 Systolic array for matrix multiplication: Version 4, $U^{\mathrm{T}}=\left[\begin{array}{lll}0 & 1 & 1\end{array}\right]$ ..... 94
Figure 3.11 Systolic array for matrix multiplication: Version $5, U^{\mathrm{T}}=\left[\begin{array}{lll}0 & 1 & -1\end{array}\right]$ ..... 95
Figure 3.12 Systolic array for matrix multiplication: Version $6, \boldsymbol{U}^{\mathrm{T}}=\left[\begin{array}{lll}1 & 0 & 1\end{array}\right]$ ..... 95
Figure 3.13 Systolic array for matrix multiplication: Version 7, $\boldsymbol{U}^{\mathrm{T}}=\left[\begin{array}{lll}1 & 0 & -1\end{array}\right]$ ..... 95
Figure 3.14 Systolic array for matrix multiplication: Version $8, \boldsymbol{U}^{\mathrm{T}}=\left[\begin{array}{lll}1 & 1 & 0\end{array}\right]$ ..... 95
Figure 3.15 Systolic array for matrix multiplication: Version 9, $\boldsymbol{U}^{\mathrm{T}}=\left[\begin{array}{lll}1 & -1 & 0\end{array}\right]$ ..... 96
Figure 3.16 Systolic array for matrix multiplication: Version $10, \boldsymbol{U}^{\mathrm{T}}=\left[\begin{array}{lll}1 & 1 & 1\end{array}\right]$. ..... 96
Figure 3.17 Systolic array for matrix multiplication: Version 11, $U^{T}=\left[\begin{array}{lll}1 & 1 & -1\end{array}\right]$ ..... 97
Figure 3.18 Systolic array for matrix multiplication: Version $12, \boldsymbol{U}^{\mathrm{T}}=\left[\begin{array}{lll}1 & -1 & 1\end{array}\right]$ ..... 97
Figure 3.19 Systolic array for matrix multiplication: Version $13, \boldsymbol{U}^{\mathrm{T}}=\left[\begin{array}{lll}-1 & 1 & 1\end{array}\right]$ ..... 97
Figure 3.20 Systolic array for matrix multiplication: Version $14, \boldsymbol{U}^{\mathrm{T}}=\left[\begin{array}{lll}2 & 1 & 1\end{array}\right]$. ..... 98
Figure 3.21 Systolic array for matrix multiplication: Version $15, U^{\mathrm{T}}=\left[\begin{array}{lll}2 & 1 & -1\end{array}\right]$ ..... 98
Figure 3.22 Systolic array for matrix multiplication: Version $16, U^{\mathrm{T}}=\left[\begin{array}{lll}0 & 1 & 1\end{array}\right]$. ..... 99
Figure 3.23 Systolic array for matrix multiplication: Version $17, U^{\mathrm{T}}=\left[\begin{array}{lll}0 & -1 & 1\end{array}\right]$ ..... 99
Figure 3.24 Systolic array for matrix multiplication: Version $18, \boldsymbol{U}^{\mathrm{T}}=\left[\begin{array}{lll}1 & 1 & 1\end{array}\right]$. ..... 100
Figure 3.25 Systolic array for matrix multiplication: Version $19, \boldsymbol{U}^{\mathrm{T}}=\left[\begin{array}{lll}1 & 1 & -1\end{array}\right]$ ..... 100
Figure 4.1 Y chart of SFG methodology ..... 101

Figure 4.2 DG for matrix-vector multiplication (a) with global communication and (b) with only localcommunication103

Figure 4.3 The SFG of matrix-vector multiplication: (a) a linear projection with projection vector $\boldsymbol{d}$; (b) a linear schedule $\boldsymbol{s}$ and its hyperplanes . . . . . . . . . . . . . . . . . . . . . . . 105

Figure 4.4 Illustration of delay-transfer rule . . . . . . . . . . . . . . . . . . . . . . . . . 108
Figure 4.5 Proof of cut-set procedure . . . . . . . . . . . . . . . . . . . . . . . . . . . . 109
Figure 4.6 Illustration of combing delays into module operations. (a) Module operation with delays in the circle; (b) The corresponding systolic array processing elements . . . . . . . . . 110

Figure 4.7 Illustration of the step-by-step mapping procedure for systolic array design . . . . 111
Figure 4.8 DG for LU decomposition . . . . . . . . . . . . . . . . . . . . . . . . . . . . 113
Figure 4.9 DG for QR decomposition . . . . . . . . . . . . . . . . . . . . . . . . . . . . 116
Figure 4.10 A new DG for QR decomposition . . . . . . . . . . . . . . . . . . . . . . . . 117
Figure 4.11 Ver. I SFG for LU decomposition . . . . . . . . . . . . . . . . . . . . . . . . 118
Figure 4.12 Ver. 2 SFG for LU decomposition . . . . . . . . . . . . . . . . . . . . . . . . 118
Figure 4.13 Ver. I SFG for QR decomposition . . . . . . . . . . . . . . . . . . . . . . . . 119
Figure 4.14 Ver. 2 SFG for QR decomposition . . . . . . . . . . . . . . . . . . . . . . . . 119
Figure 4.15 Ver. 3 SFG for QR decomposition . . . . . . . . . . . . . . . . . . . . . . . . 120
Figure 4.16 Systolic array for LU decomposition (Version 1) . . . . . . . . . . . . . . . . . 120
Figure 4.17 Systolic Array for LU decomposition (Version 2): (a) cut-sets; (b) systolic array

Figure 4.18 Systolic Array for QR decomposition (Version 1): (a) cut-sets; (b) systolic array 122

Figure 4.19 Systolic array for QR decomposition (Version 2) . . . . . . . . . . . . . . . . . 123
Figure 4.20 Systolic array for QR decomposition (Version 3) . . . . . . . . . . . . . . . . 123
Figure 4.21 Two DGs for band-matrix and vector multiplication with reversed vertical arcs and their corresponding SFGs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 125
Figure 4.22 Non-linear assignment of cascaded DG for matrix-vector multiplication . . . . . 127
Figure 4.23 Block diagram of a transputer . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 135

Figure 5.1 Systolic array for LU decomposition: (a) The array; (b) PE definitions . . . . . . 146
Figure 5.2 Systolic array for LU decomposition and forward substitution . . . . . . . . . . 147
Figure 5.3 Systolic array for computing ( $\left.C U^{-i}\right)^{\mathrm{T}}$. . . . . . . . . . . . . . . . . . . . . . . 148
Figure 5.4 Systolic array for matrix multiplication . . . . . . . . . . . . . . . . . . . . . . 148
Figure 5.5 Systolic array for linear and inverse matrix system . . . . . . . . . . . . . . . . 149
Figure 5.6 PE definitions for SALIMS ..... 150
Figure 5.7 A more efficient systolic array for linear system ..... 157
Figure 6.1 Domain decomposition ..... 164
Figure 6.2 Systolic array for linear and inverse matrix system ..... 167
Figure 6.3 Input data for SASCO ..... 168
Figure 6.4 Input data for SASCO ..... 169
Figure 6.5 Attached PEs for computation of the right hand side ..... 170
Figure 6.6 Systolic array for the solution of the arrowhead matrix system ..... 171
Figure 6.7 PE definition for the internal and external control methods ..... 173
Figure 7.1 Dissection of a geodetic problem ..... 178
Figure 7.2 Systolic array for QR decomposition ..... 182
Figure 7.3 PE definition for QR decomposition ..... 183
Figure 7.4 Extended systolic array for orthogonalization operation ..... 184
Figure 7.5 The overall architecture of the systolic array for $\mathrm{BS}^{3}$ ..... 186
Figure 8.1 Systolic array I for matrix multiplication ..... 198
Figure 8.2 Systolic array II for matrix multiplication ..... 198
Figure 8.3 Systolic array for matrix transposition ..... 199
Figure 8.4 Utilization of systolic array I ..... 200
Figure 8.5 Utilization of systolic array II ..... 200
Figure 8.6 Overall architecture for Newton's method ..... 201
Figure 8.7 Definitions of PEs ..... 202
Figure 8.8 Snapshot for $\mathbf{t}=0$ ..... 204
Figure 8.9 Snapshot for $\mathbf{t}=1$ ..... 204
Figure 8.10 Snapshot for $\mathrm{t}=2$ ..... 204
Figure 8.11 Snapshot for $\mathrm{t}=3$ ..... 204
Figure 8.12 Snapshot for $t=4$ ..... 204
Figure 8.13 Snapshot for $t=5$ ..... 204
Figure 8.14 Snapshot for $t=6$ ..... 205
Figure 8.15 Snapshot for $t=7$ ..... 205
Figure 8.16 Snapshot for $\mathrm{t}=8$ ..... 205
Figure 8.17 Snapshot for $\mathrm{t}=9$ ..... 205
Figure 8.18 Snapshot for $t=10$ ..... 205
Figure 8.19 Systolic array with convergence test ..... 206
Figure 8.20 PE definitions for systolic array with convergence test ..... 206
Figure 8.21 Initial approximation ..... 208
Figure 9.1. The sensor array and signals ..... 211
Figure 9.2 Systolic array for QR decomposition ..... 219
Figure 9.3 PE definitions for QR decomposition ..... 219
Figure 9.4 Systolic array for forward substitution ..... 220
Figure 9.5 Systolic array for norm computation ..... 220
Figure 9.6 Systolic array for Capon's DOA estimation ..... 221
Figure 9.7 PE definitions of the systolic array for Capon's method ..... 223
Figure 9.8. Simulation result ..... 225
Figure 10.1 DG for Algorithm 6 ..... 233
Figure 10.2 DG for Algorithm 8 ..... 234
Figure 10.3 Modified DG for Algorithm 8 ..... 235
Figure 10.4 A more regularized DG for Algorithm 8 ..... 236
Figure 10.5 Regularized DG for Algorithm 8 ..... 237
Figure 10.6 Selection of cut-set to systolize the SFG ..... 238
Figure 10.7 Temporally localized SFG array ..... 239
Figure 10.8 Systolic array for the CG method ..... 240
Figure 10.9 Systolic array for the CG method ..... 240
Figure 10.10 Convergence rates of CG and PCG methods ..... 243
Figure 10.11 Regularized DG for PCG method with a diagonal preconditioner ..... 244

## LIST OF TABLES

Table 1.1. Five Generations of ICs ..... 9
Table 2.1 Array geometry pattern schemes ..... 34
Table 2.2 Application scope classification scheme ..... 36
Table 2.3 Implementation level classification scheme ..... 36
Table 2.4 Granularity classification scheme ..... 37
Table 2.5 Date/Instruction stream classification scheme ..... 38
Table 2.6 Design restriction classification scheme ..... 39
Table 2.7 Example Systolic Architectures ..... 47
Table 3.1 Summary of different systolic arrays for matrix multiplication ..... 80
Table 3.2 Systolic array performance measures ..... 85
Table 3.3 Summary of different systolic arrays for matrix multiplication ..... 92
Table 3.4 Performance of different systolic arrays for matrix multiplication ..... 93
Table 5.1 The possible applications of the proposed systolic array ..... 153
Table 9.1 Computational load for Capon's method ..... 218
Table 10.1 PE definitions of systolic array for CG method ..... 241
Table 10.2 PE definitions of systolic array for PCG method with diagonal preconditioner ..... 245

## Chapter 1

## INTRODUCTION

### 1.1 Introduction

THE ever-increasing demands for computation speed and performance in the areas such as realtime signal and image processing, scientific computing, weather forecasting, control, communication, simulation, design and artificial intelligence strongly indicate the need for tremendous computation capability. These demands necessitate a revolutionary super-computing technology [KungSY88a] [KungSY85a].

The computation speed has been increasing with the advancement of integrated circuits (IC) since the late 1960s. Faster and faster very large scale integration (VLSI) electronic components have contributed a great deal on the improvement of computation speed. However, the breakthroughs in increasing switching speeds and circuit densities of VLSI devices will be difficult and costly in the future because circuit densities are fast approaching the limits of optical resolution. Even if switching times become instantaneous, distances between components may not become small enough to make a real increase in computation speed [Evans86].

The logical solution to this problem is the use of parallelism by simultaneously executing operations in multiple processors. In the situation where many processing elements (PEs) are connected together to achieve higher speed, computer architecture is of paramount importance for speeding up the system performance. Different kinds of parallel architectures have been applied in many computer systems and the usage of parallel architectures has been contributing more and more to the speed improvement of computer systems.

Systolic array architecture is one kind of parallel architectures and it possesses some promising features of utilizing VLSI technology for many important scientific computations and real time signal and image processing applications.

The availability of low-cost, high density, fast VLSI devices makes high-speed, parallel processing of large volumes of data practical and cost-effective [Mead80] [KungSY85b]. The more and more sophisticated computer-aided design (CAD) facilities bring about a major breakthrough in the design and application of massively parallel processors. The using of CAD considerably reduces the design cost and design period of a VLSI device. The parallel precessing together with the VLSI and CAD technologies makes feasible ultrahigh throughput and cost-effective computing devices.

The needs for great computational power by scientific and engineering applications keep increasing and it seems that these needs are unlimited. No matter how fast a computer of today is, a faster computer for tomorrow is demanded. The infinitively complex physical world is the driving power behind this demands. On the other hand, the technology advancement in VLSI, parallel architectures and CAD provide the potential improvement on computer systems in terms of speed and performance/cost ratio.

In this introductory chapter, we begin with an overview of the history of computers and the evolution of the integrated circuits (IC) technology. Then we survey the parallel architectures in computer systems. These areas have a very close relationship to systolic array processing.

### 1.2 Overview of Computer System History

### 1.2.1 The Definition of Computers

A computer is a device that solves problems by applying prescribed operations on data entered into it. This definition of a computer is given by The New Encyclopædia Britannica [Gwinn89]. There are two basic types of computers analog and digital. The analog computer operates on data represented by variable physical quantities, such as voltages, and measured continously. By contrast, the digital computer works with numbers, words, and symbols expressed as digits, which it manipulates and counts discretely. A third general class of computers, the hybrid computer, combines the features of the other two and utilizes both analog and discrete representations of data.

The majority of computers in use today are of the digital variety. Extremely versatile, digital computers can carry out multitude of varied tasks from routine accounting and bookkeeping to the
control of spacecraft and analysis of scientific data. We will refer computers to digital type hereafter if no confusion is caused by doing so.

The term computer did not possess the above meaning until 1950s. In US Military nomenclature through the World War II, it referred to a person who did calculations, usually for artillery trajectories, often with the use of mechanical calculators or matrices of numbers called firing tables. The term migrated to electronic devices by the 1950s. It became firmly attached to electronic devices that manipulate digital data according to standardized order to perform calculations and other tasks [Leghart89].

### 1.2.2 The Evolution of Computers

Early Developments. The computer history may date back to much earlier time. From the time man first started using arithmetic, he has been inventing devices to aid him in handling numbers [Bartee72] [Gwinn89]. One of the earliest and most ingenious examples of an aid to computation is the abacus. This primitive ( 4000 to 3000 B.C.) predecessor of modern computers consists of a rectangular frame carrying several parallel wires. Each wire supports a number of beads which are free to slide along the length of the wire. By manipulating the beads, a skilful operator can add, subtract, multiply, and divide with amazing speed.

Mechanical calculating machines were invented in Europe during 17th century. The first such device was an adding machine built in 1642 by the French scientist and philosopher Blaise Pascal. In 1673, the German mathematician Gottfried Wilhelm Leibniz constructed a calculating machine which could not only add and subtract but also multiply, divide and extract square roots.

In 1939 John V. Atanasoff, a U.S. mathematician and physicist, built a breadboard prototype of an electro-mechanical digital computer, a feature of which was the first known use of electronic vacuum tubes for computation. In the same year, Howard Aiken of Harvard University, in association with International Business Machines (IBM) Corporation, began work on a fully automatic large-scale calculator. This work resulted the machine called Automatic Sequence Controlled Calculator, commonly known as the Harvard Mark I. This was an enormous electro-mechanical machine whose operations were controlled by a sequence of instructions coded on punched paper tape.

The First Generation. Since the development of the Mark I, the digital computer has evolved at an extremely rapid pace. J. Presper Echert and John W. Mauchly, both of the University of Pennsylvania, ushered in the first generation of modern computers with the ENIAC (Electronic Numerical Integrator and Calculator). Completed by 1946, ENIAC was the first all-purpose, all electronic (primarily vacuum tubes) digital computer. The ENIAC could execute up to 5,000 basic arithmetic operations per second which was more than 1,000 times faster than its electro-mechanical predecessors. Other 'notable first generation electronic digital computers included EDVAC (Electronic Discrete Variable Automatic Computer), EDSAC (Electronic Delay Storage Automatic Calculator), UNIVAC (Universal Automatic Computer). By 1952, IBM had announced its 701 electronic calculator which used tube memory, magnetic drums, and magnetic tapes [Hwang84].

The Second Generation. The second generation computers began in 1952, when machines employing semiconductor devices known as transistors (which were invented in 1948) became commercially available. The first transistorized digital computer, TRADIC, was built by Bell Laboratories in 1954. The small size of transistor, its greater reliability, and its comparatively low power consumption made it far superior to vacuum tube. By using transistors in control, arithmetic and logic circuits, along with an improved magnetic core memory, computer manufactures were able to produce more efficient, smaller, and faster (up to 100,000 instructions per second) digital systems. Another main advancement in the second generation computer is the development of the high level languages. Fortran (formula translation), Cobol (common business oriented language) and Algol (Algorithmic language) were appeared in 1956, 1959 and 1960 respectively.

The Third Generation. The machines of this period began making heavy use of integrated circuits (ICs). In the small and medium scale integrated (SSI and MSI) circuits, many (up to hundreds of) transistors and other components are fabricated and packaged together in a single small container. The use of ICs permitted the construction of mainframe (large-scale) computers of higher operating speeds, capacity, and reliability at low cost. Their implementation also enabled engineers to design minicomputers. The next major development was large-scale integration (LSI), which made it possible to pack thousands of transistors and related devices on a single integrated circuit. Such microcircuitry yield two devices that revolutionized computer technology. The first of these devices was the microprocessor. Often referred to as "a computer on a chip", the microprocessor is an integrated circuit that contains all the arithmetic, logic, and control circuitry necessary to serve as a central processing unit (CPU). The second important device that emerged from LSI technology was the random-access memory (RAM) chip used in constructing semiconductor memory units in lieu of
magnetic core memory. Core memory was still used in CDC-6600 and other machines but, by 1968, many faster computers like CDC-7600, began to use solid-state memories (RAM). Many highperformance computers, like IBM 360/91, Illiac IV, Cyber-175, STAR-100 developed in the early seventies, belong to the third generation computers.

The Fourth Generation. The computers from 1980s have often been referred to as fourth generation though the differences between them and third-generation systems are not always clear. Most of their distinguishing features are attributable to very large-scale integration (VLSI). This technology has vastly increased the circuit density of microprocessor, memory, and support chips. The VLSI technology enables to integrate into a single chip hundreds of thousands to a few millions of parts. The reduction in cost associated with these developments has been the main factors in making possible the production of microcomputers for use in homes and schools. On the other hand, high density and low cost VLSI chips allow to construct very high speed supercomputers. Parallel processing architectures was widely used in the high performance computers of this generation. Examples of he fourth generation computers include Cyber-205 (1982), IBM 3081 (1980), Cray X-MP (1983) and Fujitsu VP-200 (1983). Today's fastest supercomputers can carry out up to 100,000 million floatingpoint operations per second (mflops).

The Future Generation. How machines can be made faster and how can they be made smarter are two perennial questions that have confronted the future computing field. The future generation computers will be faster by featuring an internal structure that allows parallel processing namely, the simultaneous execution of several or many separate operations by means of numerous integrated circuits. The future generation computers will also become smarter by incorporating artificial intelligence (AI) [Torrero85]. In order to construct a smarter computer system, a very fast computational power is required in order to process many information on which the smarter system makes decision.

To get a better view of the computer technology evolution, Figure 1.1 illustrates the past growth and future projections for the performance of supercomputers and mainframe computers. Sources are from [Halem89], [HPCR93], [IEEE89], [Miller89], [Peterson89], [Riganati84], [Smith90] and [Torrero85].


Figure 1.1. Past growth and future projections for computer performance

### 1.2.3 The Varieties of Computers

The above discussion mainly concerns the most advanced computer systems of their own times in terms of performance. These systems are often referred to supercomputers and mainframe computers. The advancement in supercomputers and mainframe computers has been paralleled by the introduction of minicomputers, workstations, personal computers, and microprocessors. For the high performance supercomputers and mainframe computers, the thrust has been to increase performance while maintaining price, while for the low-performance, low-price end of the product spectrum, the thrust has been to maintain performance while reducing cost. To sum up, computer system can be categorized as follows.

Super Computers: The world's fastest, most powerful computers available at any given time. Super computers are typically significantly more powerful than other computers. They are used in science, engineering, and research for the most difficult processing challenges, such as weather forecasting.

Mainframe Computers: Large, general-purpose computers that serve hundreds or thousands of users, all tied to a corporate data processing centre.

Mini-Supercomputers: Computers that handle complex mathematical computations for engineers, scientists, and researchers. Usually, mini-supercomputers deliver a quarter to half the performance of a supercomputer at only one-tenth the cost.

Minicomputers: Minicomputers can handle the general needs of more than 100 users who typically work on terminals wired to the computer.

Workstations: High-performance computers used by engineers, scientist, and technical professionals who need superior graphics. Workstations can offer the nearly same performance of minicomputer while severing less users than minicomputers.

Personal Computers (Microcomputer): General purpose desktop computers that use 16- or 32-bit, even 64-bit, microprocessors. Personal computers are very popular today due to the increasing performance and decreasing prices.

Microprocessors: A microprocessor is defined as the central arithmetic and logic unit of a computer, together with associate circuitry, scaled down to fit on a single silicon chip (sometimes several chips) holding many thousands (may be up to a few millions with the state-of-the-art technology) of transistors, resistors and similar circuit elements.

Microprocessors are at the opposite end to supercomputers. They have found wide applications in engineering and industrial applications. The evolution of microprocessor technology has been proceeding at an amazing pace. The first microprocessor, the 4-bit Intel 4004, was fabricated in 1971 by Intel. Subsequently, 8-bit, 16-bit and 32-bit microprocessors were introduced in 1972, 1974, and 1981 respectively [Gupta83] [Gupta87]. The 64-bit microprocessors are also commercially available today. The number of devices per chip, the clock frequency and the overall throughput have increased tremendously during the last two decades. Today's microprocessors can integrate one or two millions transistor inside a single chip with a peak performance up to 100 mega flops which is approximately equivalent to the performance of a second or third generation supercomputer. The Intel i860 64-bit microprocessor [Kohn89] and the INMOS T9000 transputer [May92] are typical examples of such high performance 64 -bit microprocessors.

In contrast to the general-purpose microprocessors, a special class of microprocessor, called digital signal processing (DSP) microprocessors, has been progressing at a fast pace during the last two decades. To meet the special requirements of digital signal processing applications, DSP microprocessors have one feature: speed in the execution of certain algorithms [Morris86]. Since the first DSP microprocessor, the Intel 2920 appeared in 1982, many types of DSP microprocessors have been developed with more and more powerful processing capabilities.

### 1.2.4 General Purpose Versus Special Purpose

While the general-purpose supercomputer can provide the high speed in the cost of high expenses, increasing interest, attention and effort have been paid to the special purpose super-performance computer systems which comprise the cost and the generality. Due to the limited demand for a particular special-purpose super machine, their developmental cost must be justified against the need to solve the problem in a time more acceptable than that of a general-purpose machine of similar cost [Norrie84]. A common example of this may be in weather forecasting. A special purpose weather forecasting computer may be able to generate more accurate weather predictions than a generalpurpose machine of the same cost. Fortunately, new innovative architectures, CAD and VLSI are helping to bring the costs of special-purpose machines down, and make them more feasible.

Special purpose computers can be divided into two categories. In the first of these categories, the architecture is modeled to reflect the physical structure of the problem to be solved. An early example of this is a finite-element solving computer being constructed by NASA, which employs an array of microprocessors, one for each node in the finite-element model [Jordan79]. In the second category of special-purpose machines, the architecture is designed to reflect the general solution method for a class of problems.

With the advancement in VLSI technology, it is now possible for customers to construct their own special purpose systems with much less costs than supercomputers by utilizing multiple VLSI devices such as DSP microprocessors in parallel. Some of such systems can outperform even conventional supercomputers [Gunzinger92] at lower cost. For example, the GF11 system with a maximum of 566 DSP processors has a peak performance of 11 GFlops [Witbrock90].

The evolution of computer technology is a history of increasing the computational power. Although today's computers are very powerful in terms of processing speed, many applications demand more powerful computers for solving problems faster or solving larger problems. The advancement of VLSI technology make it possible to construct cost-effective parallel processing system with a large number of processing elements to achieve higher performance.

### 1.3 Evolution of IC Technology

Integrated circuit technology and computer technology are interdependent despite their relative independent origins [Burger84]. On the one hand, computers, as major component of IC design systems (CAD) that aid and automate VLSI design process, have permitted major advances in IC complexity. On the other hand, advances in IC technology have made high-capacity semiconductor devices readily available and have enabled us to design new classes of computing devices, such as systolic arrays that offer substantial performance gain via massive parallelism and local communication.

### 1.3.1 Five Generations of ICs

The evolution of IC technology can be divided into five generations which are not congruent with the five generations of computers. The five generations of ICs have been derived from the progress of device technology over the last several decades. Small, medium, large, very large, and ultra large scale integration-or SSI, MSI, LSI, VLSI and ULSI-are represented by ranges of chip complexity such that the upper

Table 1.1. Five Generations of ICs

| Generation | Complexity |
| :---: | :---: |
| SSI | $2-64$ |
| MSI | $62-2000$ |
| LSI | $2000-64,000$ |
| VLSI | $64,000-2,000,000$ |
| ULSI | $2,000,000-64,000,000$ | limit on each range is 32 times the lower limit.

Table 1.1 lists the five generations of IC with their complexities in terms of the number of components in a single package. The specific values of these limits are not universally accepted, but the general nature of the approach is [Burgur84]. Today's technology is at the intersection edge of VLSI and ULSI.

Shortly after the germanium transistor was demonstrated in the late 1940's, effort were initiated to make a transistor in silicon. By the mid-1950's silicon transistors had become available, and the technology of this material was rapidly advancing. By 1960, photolithography, oxide masking, and impurity diffusion techniques were being applied to produce high performance transistors.

Since its invention in 1959, the IC has undergone rapid growth in which chip complexity figured prominently. After 1959, IC chip complexity doubled every year. In 1973, complexity had reached
nearly 8000 components per chip. Complexity doubled every 1.5 to two years later on. This progression is known as Moore's Law [Moore75]. The complexity growth has been slow down recently as the technology is fast approaching to its limit.

IC technologists often describe IC evolution in terms of feature size, or the linewidth of the smallest dimension to be fabricated in silicon. In metal-oxide semiconductor (MOS) technology, feature size is the field-effect transistor gate length. In bipolar technology, it is the emitter width. In 1960, photolithography produced linewidth of about $37 \mu \mathrm{~m}$. In 1984, devices with feature sizes of $2 \mu \mathrm{~m}$ are in production. Today's technology can fabricate devices with feature sizes of less than $0.5 \mu \mathrm{~m}$.

Another important factor to describe IC technology is chip size. Complexity growth is not only due to feature size reduction but depends also on increases in chip size. Chip area has increased by two order of magnitude in the last three decades as silicon substrate quality and process controls have advanced to permit economically acceptable yields at these chip sizes.


Figure 1.2 Integrated circuit complexity trends

Figure 1.2 shows the IC technology evolution in terms of complexity, feature size, and chip area [Burger84] [Hurst92].

Many other parameters, such as wafer size, number of chips per wafer, chip costs, gate density, and production levels, may be used to trace the progression of IC technology. Of these, IC technology is especially impressive when viewed in terms of cost decreases. The IC technology today are more impressive by using the CAD for design automation which even make it possible to produce custom, limited-production parts at low cost.

### 1.3.2 VLSI Technology Constraints

There are currently two popular semiconductor device technology: bipolar and metal-oxide semiconductor (MOS). Although new technologies combining features of both are emerging, some important differences between bipolar and MOS devices exist. While bipolar technology is faster, MOS technology, offers higher density integration and consumes less power. VLSI technology offers promising potential but creates new design constraints. The key design criteria for VLSI architectures are listed as follows [Randell82] [KungSY88b]:

- Critical design complexity and essential CAD tools
- Modularity and effective utilization of building blocks
- Simple and regular data and control paths
- Localized or reduced interconnections
- Balance between input/output (I/O) and computation
- Extensive concurrency (i.e., pipeline and/or parallel processing)
- Synchronous versus asynchronous implementation considerations
- Programmability
- Adequate reconfigurability and fault tolerance
- Balanced array and chip partitioning.

As long as communication in VLSI remains restrictive, locally interconnected arrays will be of great importance. An increase in efficiency can be expected if the algorithm arranges for a balanced distribution of work load while observing the requirement of locality, i.e., short communication paths. These properties of load distribution and information serves as a guideline to the designer of VLSI algorithms and eventually lead to new VLSI architecture design.

### 1.3.3 VLSI Architecture Design Principles

Scaling Effect. The growth of IC complexity and capability has been due to a combination of a down scaling of the feature size and up scaling of the chip size, both subject to the constraint of reasonable yield. VLSI architecture enjoys a major advantage of being very scalable technologically [Mead80]. This means that the efforts of architectural redesign will be minor when the device technology is scaled down to the submicron level.

In the scaling of geometry, it is often assumed that all the dimensions as well as the voltages and currents on the chip are scaled down by a factor $\alpha$ (an $\alpha$ greater than 1 implies that the sizes or levels are shrinking). When scaling down the linear dimensions of a transistor by $\alpha$, the number of transistors that can be placed on a chip of given size is scaled up by $\alpha^{2}$.

The switching delay of a transistor is scaled down at least by $\alpha$ due to the fact that the channel length is decreased by a factor $\alpha$. This means that gate delay decrease with scaling.

Scaling also affects the interconnections between devices. Since the cross-sectional area of the conductor is decreased by a factor of $\alpha^{2}$ and the length of the conductor is decreased by $\alpha$, the increase of resistance is in proportion to $\alpha$. At the same time, scaling down implies changes of capacitance of the interconnection. Regarding the conductor as one plate of a parallel-plate capacitor, scaling down of both linear dimensions of the plate by $\alpha$ implies that a decrease of the capacitance by $\alpha^{2}$. However, scaling down also implies a decrease by $\alpha$ of the thickness of the oxide insulating layer separating the plates of the capacitor. Hence the capacitor of a fixed interconnection scales down by $\alpha$. Therefore the RC (resistor-capacitor) time constant and the interconnection delay are unchanged because the scaling up of resistance and down of capacitance exactly cancels.

Although the scaling process has many beneficial effects, in contrast, interconnection problems become very severe due to increased chip size. Eventually, chip cost, performance, and speed are determined primarily by interconnect delay and area.

Stuffing Effect. It is clear that since gate delays decrease with scaling, whereas interconnection delays remain constant with scaling, eventually the speed at which a circuit can operate will be dominated by
interconnect delays rather than device delays. However, this situation is actually somewhat worse than the above considerations imply, due to the factor of stuffing.

Assume that circuits are consist of randomly connected logic elements. Stuffing means that the length of the interconnections do not scale down with the inverse of $\alpha$, as was assumed in scaling. In practice, as the complexity of the circuit increases, the distance over which interconnections must be maintained on a chip of fixed area may stay roughly constant. It has been argued from statistical considerations [Keyes79] that a good approximation to the maximum length of interconnection required is given by

$$
L_{\max }=\frac{A^{1 / 2}}{2}
$$

where $A$ represents the area of the chip. Therefore, if scaling occurs and chip size is also increased, the interconnection problem becomes further exacerbated. When the delay time of the circuit depends largely on the interconnection delay instead of the logic gate delay, minimal and local interconnections will become an essential factor for an effective realization of the VLSI circuits.

Regularity and Modularity. In VLSI design, the overall architecture should be as regular and modular as possible, thus reducing design error, time and cost. Any form of regularity, as derived by a careful algorithmic design, may prove useful for mapping algorithms onto architectures. The building-block concept is also instrumental and important in VLSI design. The building-block approach may be combined with high level CAD tools, such as silicon compilers, to give the VLSI designer a tremendous amount of flexibility to cope with the ever-increasing complexity of VLSI design.

I/O Constraints. With the increasing complexity of VLSI device, the number of I/O pins becomes a critical factor for VLSI design. Very often parallel signal communication wires for each node in many directions is required. A good design should take into account the constraints on I/O pins and resultant costs in terms of area and time.

Pipeline and Parallel Processing. Throughput rate is the overriding factor dictating the system performance. Extensive use of concurrency by pipeline and parallel processing is required in order to maximize the throughput.

Hierarchical Design and CAD Techniques. The growth in VLSI complexity has made hierarchical CAD techniques necessary. Such approaches are imperative due to the extremely large number of components per chip in the current VLSI technology. The development of hierarchical and structured design methodology and simplified VLSI design rules allow VLSI chips to be designed quickly.

In addition to the above mentioned aspects for VLSI design, there are other aspects required to be considered when design a VLSI device. The choice between synchronous and asynchronous clocking become very crucial for complex VLSI systems. Programmability and reconfigurability allow the designed VLSI systems to have more flexibility to meet the requirements of different applications. Fault tolerant property can improve the yield rate of VLSI devices.

### 1.4 Parallel Computer Architectures

Many scientific and technological problems demand more and more powerful computational capabilities. The history of computer technology evolution has clearly indicated the continuous performance improvement of computer systems since the invention of the first computer. Although faster and faster electronic devices with increasing switching speed and circuit density contribute very much to the performance improvement of computer systems, they cannot alone meet the everincreasing requirements in a cost effective way, especially in the case where the switch speed and circuit density are fast approaching the limits of optical resolution. Parallel processing is a natural and logical choice for further improving computer performance. The evolution of computer systems has been on evidence that increasing computer performance is due to not only using high speed electronic devices but also using parallel processing architectures. Massively using parallelism to achieve further performance improvement in addition to the improvement due to reducing switching delays has been bringing about significant breakthrough in computer technology.

The concept of parallelism breaks with the classical approach of obtaining speed by performing each operation more rapidly. In parallel computation, the speed increase comes from the simultaneous execution of operations. Once many processing elements are put together, architecture, which defines how different processing elements cooperate and how data communicate, plays a very important role in the parallel computer system design. In this section, we will have a brief overview of the computer architectures.

### 1.4.1. Flynn's Taxonomy

Flynn's taxonomy [Flynn66] classifies architectures on the presence of single or multiple streams of instruction and data. This yields the four categories below:


Figure 1.3 The four types of computer architectures (a) SISD; (b) SIMD; (c) MISD; (d) MIMD

SISD (Single Instruction Single Data stream)—defines serial computers as shown in Figure 1.3a.

- SIMD (Single Instruction Multiple Data stream)—involves multiple processors simultaneously executing the same instruction on different data as shown in Figure 1.3b.
- MISD (Multiple Instruction Single Data stream) -would involve multiple processors applying different instructions to a single datum; this hypothetical possibility is generally unrealistic but which Flynn affirms to include specialized streaming organizations as shown in Figure 1.3c.
- MIMD (Multiple Instruction Multiple Data stream)—involves multiple processors autonomously executing diverse instructions on diverse data as shown in Figure 1.3d.

Although these distinctions provide a useful shorthand for characterizing architectures, they are insufficient for classifying various modern computers. For example, pipelined vector processors merit inclusion as parallel architectures, since they exhibit substantial concurrent arithmetic execution and can manipulate hundreds of vector elements in parallel. However, they are difficult to accommodate within Flynn's taxonomy, because they lack processors executing the same instruction in SIMD and lack the asynchronous autonomy of the MIMD category. Duncan's taxonomy [Duncan90] attempted to cope with the encountered problems in Flynn's taxonomy.

### 1.4.2 Duncan's Taxonomy

Since 1980s, a wide variety of new computer architectures for parallel processing have been innovated by extending the major approaches to parallel computing developed in 1960s and 1970s. Because the diversity of parallel architectures, diverse definition have been proposed for parallel architectures. Flynn's taxonomy [Flynn66] based on instruction and data stream has been widely adopted. Recently, Duncan proposed a taxonomy which placed recent architecture innovations in the broader context of parallel architectures and in a coherent framework [Duncan90]. The proposed taxonomy satisfies the following set of imperatives:

- Exclude architectures incorporating only low-level parallel mechanism that have become commonplace features of modern computers;
- Maintain elements of Flynn's useful taxonomy based on instruction and data streams;
- Include pipelined vector processors and other architectures that intuitively seem to merit inclusion as parallel architectures, but which are difficult to gracefully accommodate within Flynn's scheme.


Figure 1.4 High-level taxonomy of parallel computer architectures

Under these conditions, a parallel architecture can be defined as an explicit, high-level framework for the development of parallel programming solutions by providing multiple processors, whether simple or complex, that cooperate to solve problems through concurrent execution. Figure 1.4 shows the Duncan's taxonomy which uses high-level categories to delineate the principle approaches and to define a coherent spectrum of architectural alternatives. Definitions for different categories are briefly provided below.

### 1.4.2.1 Synchronous Architectures

Synchronous parallel architectures coordinate concurrent operations in lockstep through global clocks, central control units or vector unit controllers.

Pipelined vector processors. Vector processors are characterized by multiple, pipelined functional units, which implement arithmetic and Boolean operations for both vectors and scalars and which can operate concurrently. Such architectures provide parallel vector processing by sequentially


Figure 1.5 Pipelined vector parallel architecture streaming vector elements through a functional unit pipeline and by streaming the output results of one unit into the pipeline of another as input. Figure 1.5 shows a representive pipelined architecture having a vector addition unit consisting of six pipeline stages. If each pipeline stage has a cycle time of 20 nanoseconds(ns), then 120 ns elapse from the time operands $a l$ and $b 1$ enter stage 1 until result $c l$ is available. When the pipe line is filled, however, a result is available every 20 ns .

SIMD architectures. SIMD architectures (Figure 1.3b) typically employ a central control unit, multiple processors, and an interconnection network (IN) for either processor-to-processor or processor-to-memory communications. The control unit broadcasts a single instruction to all processors, which execute the instruction in lockstep fashion on local data. The IN allows instruction results calculated at one processor to be communicated to another processor for use as operands in a subsequent instruction.

Under the SIMD architectures, there are two sub-categories which are namely processor array architecture and associate memory processor architecture. A processor array architecture computer is a synchronous parallel computer with multiple arithmetic logic units, called


Figure 1.6 Processor array architecture processing elements (PE), that
can operate in parallel in a lockstep fashion. A typical processor array architecture is depicted in Figure 1.6.

An associative memory processor architecture computer uses special comparison logic to access stored data in parallel according to its contents. Figure 1.7 shows the characteristic functional units of an associative memory processor. A control unit (serial computer) reads and executes instructions, invoking a specialized array controller when associative memory instructions are


Figure 1.7 Associative memory architecture encountered. Special registers enable the program controller (serial computer) and associative memory to share data.

Systolic architectures. The systolic architectures were first introduced in 1978 by Kung and Leiserson [KungHT78] as highperformance, special-purpose VLSI computer systems that are suitable for specific application requirements that must balance intensive computations with demanding I/O bandwidths [KungHT80a]


Figure 1.8 Systolic architecture [KungHT82a]. Systolic architectures (systolic arrays) are organized as networks which are composed of a large number of identical, elementary cells (or processing elements) which are locally connected. Data in systolic arrays are pulsed in rhythmic fashion from memory and through the cells before returning to memory (see Figure 1.8). A global clock and explicit timing delays synchronize the system. Modular processors united by regular and local interconnections provide basic building blocks for a variety of special-purpose systems. Systolic arrays address the performance requirements of special-purpose systems by achieving significant parallel computations and by avoiding I/O and memory bandwidth bottlenecks.

### 1.4.2.2 MIMD Architectures

MIMD architectures (Figure 1.3d) employ multiple processors that execute independent instruction streams using local data. MIMD can support parallel solutions that require processors to operate in a largely autonomous manner. Although software processes executing on MIMD architectures are synchronized by passing messages through an interconnection network or by accessing data in shared memory, MIMD architectures are asynchronous computers, characterized by decentralized hardware control. MIMD computers support high level parallelism (subprogram and task levels).

Distributed memory architectures. Distributed memory architectures (Figure 1.9) connect processing nodes, consisting of an autonomous processor and its local memory, with a processor-to-processor interconnection network. Nodes share data by explicitly passing messages through the interconnection network, since there is no shared memory. Various interconnection


Figure 1.9 Distributed memory architecture network topologies have been proposed to support architectural expendability and provide efficient performance for parallel programs with different interprocessor communication patterns. Examples of the MIMD interconnection topologies includes ring topology, mesh topology, tree topology, hypercube topology, and reconfigurable topology as shown in Figure 1.10.

## Shared memory architectures.

Shared memory architectures accomplish interprocessor


Figure 1.10 MIMD interconnection network topologies: (a) ring; (b); mesh; (c) tree; (d) hypercube; (e) tree mapped to a reconfigurable mesh
coordination by providing a global, shared memory that each processor can address. Typically, each processor in a shared memory architecture also has a local memory used as a cache. Shared memory computers do not have some of the problems encountered by message-passing architectures, such as message latency as data is queued and forwarded by intermediate nodes. However, other problems, such as data access synchronization and cache coherency, must be solved.

Figure 1.11 illustrates some shared memory architectures with different interconnections. Time shared buses offers a fairly simple way to give multiple processors access to a shared memory as shown in Figure 1.11a. A single, time shared bus can accommodates a small number of processors (from 4 to 20) since only one processor accesses the bus at any given time. Crossbar interconnections use a crossbar switch to connect processors to memories as shown in Figure 1.11b. Crossbar interconnections also limit the number of processors to be used


Figure 1.11 MIMD shared memory architectures: (a) bus interconnection; (b) crossbar; (c) multistage interconnection network.
(from 4 to 16). Multistage interconnection networks (Figure 1.1/c) strike a comprise between the price/performance alternatives offered by crossbars and buses.

### 1.4.2.3 MIMD Paradigms

MIMD/SIMD hybrids, dataflow architectures, reduction machines, and wavefront arrays all pose a similar difficulty for an orderly taxonomy of parallel architectures. Each of these architectural types is predicated on MIMD principles of asynchronous operation and concurrent manipulation of multiple instruction and data streams. However, each of these architectures is also based on a distinctive
organizing principle as fundamental to its overall design as MIMD characteristics. Therefore, these architecture are described under MIMD based architectural paradigms [Duncan90].

MIMD/SIMD architectures. MIMD/SIMD hybrid architectures allow selected portions of a MIMD architecture to be controlled in SIMD fashion. Figure 1.12 conceptually shows a MIMD/SIMD hybrid architecture.

Dataflow architectures. The fundamental feature of dataflow architectures is an execution paradigm in which instructions are enabled for execution as soon as all of their operands become available. Thus, the sequence of executed instructions is based on data dependencies, allowing dataflow architectures to exploit concurrency at the task, routine, and instruction levels. Figure 1.13 depicts a dataflow architecture executing dataflow graphs of program fragment.


Figure 1.12 MIMD/SIMD architecture


Figure 1.13 Dataflow architecture


Figure 1.14 Reduction architecture: (a) program; (b) demand token production; (c) result token production.

Reduction architectures. Reduction, or demand driven, architectures [Treleaven82] implement an execution paradigm in which an instruction is enabled for execution when its results are required as operands for another instruction already enabled for execution. Figure 1.14 illustrates a simplified reduction architecture. Figure 1.14 a shows the program to be performed by the architecture. Figure 1.14 b shows all the demand tokens produced by the program as they propagate down the tree. Figure 1.14 c shows the last two result tokens produced as then are passed to the root node.

## Wavefront array architectures.

Wavefront array processors combine systolic data pipelining with an asynchronous dataflow execution paradigm. S.Y. Kung developed wavefront array concepts in the early 1980s [KungSY82a]. Wavefront and systolic architectures are both characterized by modular processors and regular, local interconnection networks. However, wavefront arrays replace the global clock and explicit time delays used for synchronizing systolic data pipelining with asynchronous


Figure 1.15 Wavefront array architecture handshaking as the mechanism for coordinating interprocessor data movement. Thus, when a processor has performed its computations and is ready to pass data to its successor, it informs the successor, sends data when the successor indicates it is ready, and receives an acknowledgement from the successor. The handshaking mechanism makes computational wavefronts pass smoothly through the array without intersecting, as the array's processor act as a wave propagating medium. In this manner, correct sequencing of computations replaces the correct timing of systolic architectures. Figure 1.15 depicts a two dimensional wavefront architecture for matrix multiplication.

### 1.5 Organization of the Thesis

We have briefly overview the computer history which clearly indicates the ever-increasing demand of computational speed and, of course, ever increasing effort to improve the performance of computer systems. The availability of low-cost high-density, high speed VLSI devices makes it possible to build up massively parallel processors with a reasonable cost. Parallel architectures have been significantly contributing to computer speed increase. The current VLSI technology, on the other hand, put some constraints on computer architecture. Systolic array and wavefront array architectures are very amenable to VLSI implementation, since they feature the important properties of modularity, regularity, local interconnection, and highly pipelined multiprocessing.

This thesis stress the systolic array processing and its applications in scientific computation and real time signal processing. This thesis can be divided into three parts. First part is an introductory part containing both an introduction to broad areas related to systolic array processing and a detailed introduction to systolic array processing itself. The second part is concerned with design methodologies of systolic algorithms. The third part designs a number of systolic arrays for different applications. Finally a summary and a list of references end the thesis.

### 1.5.1 Part I: Introduction

Part 1 forms the introductory part of the thesis and it consists of Chap. 1 and Chap. 2. Chap. 1 gives an introduction to broader areas which systolic array processing closely relates to, including computer evolution, VLSI technology advancement and parallel architectures. Systolic array architecture is one class of parallel computer architectures and it best meets the requirements by VLSI technology. Systolic array architecture provides promising solutions to many large-scale scientific computation applications and real time signal/image processing applications.

Chap. 2 is an overview of the systolic array architecture itself. In Chapter 2, we mainly stress the following problems:

- What is a systolic array architecture?
- Why do we use systolic array architectures?
- How to classify systolic architectures?
- What problems can systolic array architectures solve?
- What kinds of systolic array systems have been implemented?
- How to design systolic array architecture?


### 1.5.2 Part 2: Systolic Array Design Methodologies

Part 2 introduces two types of the most popular systematic design methodologies for systolic arrays. Some systolic arrays for important matrix computations, including matrix multiplication and matrix triangularization, are designed as examples to illustrate the methodologies.

Chapter 3 deals with the regular iterative algorithms (RIA) approach for designing systolic arrays. The matrix-matrix multiplication serves an example for the design method.

Chapter 4 addresses the signal flow' graph (SFG) method for systolic array design. The QR and LU decomposition systolic arrays are designed by the SFG technique.

### 1.5.3 Part 3: Systolic Array Architectures and Applications

A number of systolic arrays have been designed in this thesis for different application areas. Firstly, we designed a systolic array for linear and inverse matrix system. The systolic array combines LU (or QR) decomposition, backward and forward substitution, and matrix multiplication into a trapezoidal array. The array can be used to solve a variety of linear systems and other problems such as matrix multiplication. When QR decomposition is used, the systolic array can solve linear least square problems or perform pseudo-inverse computation. Then a systolic array is designed to compute the Schur complement which often occurs in the domain decomposition method and other problems in linear algebra such as the numerical solution of elliptic partial differential equations and sparse linear system. A systolic array architecture has been designed and simulated for QR decomposition of block structured sparse systems which find applications in areas such as photogrammetry, Doppler radar positioning, geodetic problems, cluster analysis and pattern matching. A novel systolic array is derived for computing matrix inversion by Newton Iteration which has a good feature that some simper initial approximations can always guarantee a convergence. A systolic array architecture for Capon's DOA estimation in sonar and radar applications is derived based on the QR decomposition technique. Finally a novel linear systolic array architecture is proposed for conjugate gradient (CG) algorithm.

### 1.6 Summary

The systolic array architecture in parallel processing came as a product of a certain environment, that encompassed the needs (the possible applications); the means (the appropriate technology); and the background knowledge for its realization [Evans91]. The needs can be outlined as the ever-increasing tendency for faster and more complex computations, especially in areas like real-time signal processing and large-scale scientific computation. The evolution of computer history has been witnessing the increasing demands for more powerful computational facilities and the unremitting efforts to meet the demands. The means were provided by the remarkable advances in VLSI technology and CAD tools. which make it feasible to implement a large scale parallel processing system by carefully designing the architectures. Finally, the background includes the applications of parallel architectures in the form of parallel algorithms and the design of parallel computers.

Systolic array architecture, as a subclass of parallel computer architectures, features modularity, regularity, and local interconnection. Due to these features, systolic arrays are best amendable to VLSI implementation. Therefore, systolic arrays provide a promising solution to many computationally intensive problems encountered in the areas such as real-time signal and image processing and large scale scientific computations.

## Chapter 2

## OVERVIEW OF SYSTOLIC ARRAY

## ARCHITECTURES

### 2.1 What is a Systolic Array?

SYSTOLIC architectures were first introduced in 1978 by H.T. Kung and C.E. Leiserson [KungHT78] as high-performance, special-purpose VLSI computer systems that are suitable for specific application requirements or to off-load computations that are especially taxing to general purpose computer.

The term systolic is in fact taken from the Greek word systole ( $\sigma v \sigma \tau 0 \lambda \eta$ ). The physiology term systole and diastole are used to describe the rhythmic contraction of the heart as blood is pumped around the human body. The body itself is composed of approximately 1000 million million cells. Cells of the same kind are grouped into tissues which have special tasks to perform. One of the many tasks of the blood is to deliver oxygen to all the body tissues. The cells need oxygen to live and use it to perform chemical reactions which produce energy and maintain bodily functions. The reactions also produce waste matter which the blood carries away from the cells to organs that dispose of it. Blood returning from tissues to the heart passes through the lungs where it exchanges carbon dioxide and waste gas accumulated from the cells for more oxygen. Full of the oxygen, the blood returns to the heart and is pumped out to the tissues again. This simple description gives an analogy to systolic processing.

In systolic computation body, cells and tissues are replaced by simple processing elements grouped into arrays which have specialized tasks or algorithms to perform. The circulating blood is replaced by the concept of data flow and individual blood cells carrying oxygen are replaced by discrete pieces of data. The processing element perform simple computations on the data when it arrives and generate results, these results are passed on from processing element to processing element as data flows around the system. New data is entered and old data removed as the data
flow passes through some external device (called the host) made up from a more conventional computer architecture.

There are a number of "definitions" of systolic arrays [KungHT78] [Ullman84] [KungSY88b] [Megson92] [Evans91] [Johnson93]. According to Kung and Leiserson [KungHT78], we have the following definition:
"A Systolic system is a nenwork of processors which rhythmically compute and pass data through the system."

## Bellow we present a more coherent and detailed definition for systolic array.

Definition: A systolic array is a computing system which possesses the following features:

- [D1] Network. It is a computing network employing a number of processing elements (or cells) with interconnections.
- [D2] Rhythm. The data are computed and passed through the network in a rhythmic and recurrent manner.
- [D3] Synchrony. The execution of instructions and the communication of data are timed by a global clock.
- [D4] Modularity. The network consists of one or, at most, a few types of processing elements.
- [D5] Regularity. The interconnections between cells are regular and homogeneous. The numbers of interconnections for processing elements are independent on the problem size.
- [D6] Locality. The network manifests a locally communicative interconnection structure.
- [D7] Boundary. Only boundary cells in the network communicate with the outside world.
- [D8] Extensibility. The computing network may be extended indefinitely.
- [D9] Pipelinability. The network exhibits a linear rate pipelinability, i.e. it should achieve an $O(M)$ speedup, in terms of processing rate, where $M$ is the number of processing elements. Here the term speedup denotes the ratio of the processing time in a single processor to the processing time of $M$ processors for performing the same tasks.

The above definition can be exemplified with the help of Figure 2.1, which is a typical arrangement of systolic system. A linear array of processing elements (PEs) is connected to form a computing network [D1]. The processing elements perform


Figure 2.1 A typical systolic system computations on the data when they arrive and generate results; these results are passed on from one processing element to the next processing element in a similar manner to the rhythmic blood pumping of a human body [D2]. The whole array of processing element is timed by a global clock in order to coordinate the computations and communications [D3]. The systolic system consists of only a single type of PEs and hence exhibits the modularity [D4]. The interconnections between cells are homogeneous by receiving data from left neighbour and send data to right neighbour. The number of interconnections for processing elements is fixed to be 2 here no matter what is the size of problem to be solved [D5]. The PEs in the array communicate only to its nearest (left and right) neighbours [D6] except for the boundary PEs (the leftmost and the rightmost PEs) which can communicate to the outside world, i.e. the host in this case [D7]. This array can be easily expanded from one size to another size depending on the size of problem to be solved and the performance to be achieved [D8]. The array also exhibits a linear rate pipelinability. That is, if the processing time of a single PE for solving a certain problem is $T$, then the processing time of $M$ PEs for solving the same problem is reduced by $O(M)[\mathrm{D} 9]$.

The above definition for systolic array is in a very strict sense and it provides a strict framework for the systolic architecture. The features in the definition can be used to distinguish systolic array architecture from other parallel architectures. However, when implementing a systolic architecture in a hardware system, many trade-off may be made. Some items in the definition may be loosen in order to meet technology constraints and/or performance requirements. For example, some real systolic systems allow a certain degree of global communication and some are implemented in asynchronous way. In practice, as long as a computational system satisfies most of the item listed in the definition and distinguishable from other parallel architectures, we can call such a system as a systolic array architecture.

It is also noticed that qualitative definition has been employed in the above definition. We will introduce another definition of systolic array using a quantitative model (in Chap. 3).

Now let us conclude this section by the following passage quoted from the introduction of C.E. Leiserson's thesis [Leiserson81]:
"Systolic systems are an attempt to capture the concepts of parallelism, pipelining and interconnection structures in a unified framework of mathematics and engineering. They embody engineering techniques such as multiprocessing and pipelining together with the more theoretical ideas of cellular automata and algorithms, and therefore are an excellent subject of investigation from a combined standpoint. "

### 2.2 Why Systolic Architectures?

To answer this question, we can consider the following two aspects: the external causes and the internal causes. The external causes include of technology advances in VLSI, intensive uses of parallel processing and demanding scientific applications. These aspects form the environment for systolic array processing. The internal causes are due to the essential characteristics of systolic architectures themselves: simple and regular design, concurrency and communication, and balancing computation with I/O.

### 2.2.1 The External Causes

Three factors have contributed to systolic array's evolution into a leading and promising approach for handling computationally intensive applications: technology advances, parallel processing, and demanding scientific applications.

VLSI Technology Advances. Advances in VLSI technology complement the systolic array's qualification as one of the preferable architectures for many computationally intensive applications.

Density of VLSI devices has been increasing due to the photolithography technology advance. Faster and faster gate switching speed can be achieved due to scaling effect when feature size is
reduced. Larger chip area size can host more components in a single silicon package. Wafer scale integration (WSI) promises a further growth of the number of components in a single package.

Economical design and fabrication processes produce less expensive systolic chips, even in small quantities. Better design tools allow arrays to be designed more efficiently. A systolic cell can be fully simulated before fabrication, reducing the chances that it will fail to work as designed. With advances in simulation techniques, fully tested, unique cells can now be quickly copied and arranged in regular, modular arrays. As VLSI designs become more complicated, "systolicizing" them provides an efficient way to ensure fault tolerance; any fault tolerance precautions built into one cell are extensible to all cells.

Relatively new field-programmable gate array (FPGA) technology permits a reconfigurable architecture and this makes systolic array more flexible. Different systolic array topologies and different cell function can be configured on site without change hardware.

Parallel Processing. Past efforts to add concurrency to conventional von Neumann computer architecture have yield coprocessors, multiprocessing units, data pipelining, and array processors. Systolic arrays combine features from all of these architectures in a massively parallel architecture. A systolic array can act as a coprocessor, can contain multiple processors, and can act as a pipeline (maybe $n$-dimensional pipeline).

Demanding Scientific Applications. The technology growth of the last three decades has produced computing environments that make it feasible to attack demanding scientific applications on a large scale. Large-matrix computation, feature extraction, radar and sonar signal processing, image processing, weather forecasting are only a few examples. As history shows, when many computers work on a wide variety of applications, they develop new applications requiring more demanding computational performance. Many applications today require massive, repetitive and rhythmic parallel processing. Systolic arrays are right candidates for these applications.

### 2.2.2 The Internal Causes

The technology advances in VLSI/WSI make systolic arrays feasible. The evolution of parallel processing introduced many parallel architectures which provide the basic element of systolic
architecture, e.g. multiprocessors, pipelining. The demanding scientific applications, especially those requiring massive and repetitive computations, necessitate a new architecture. These aspects form the external driving forces for systolic architecture.

More importantly, systolic array themselves feature the following essential characteristics: simple and regular design, concurrency and communication, and balancing computation with l/O [KungHT82a]. These features make systolic architecture best fit the current technology advances and requirements.

### 2.2.2.1 Simple and Regular Design

In VLSI technology, the cost of components is dropping dramatically; however, the cost of design grows with the complexity of the system. By using a regular and simple design and exploiting the VLSI technology, great savings in design cost can be achieved.

Systolic architecture is based on a very few types of basic cells, as we have discussed, and this is the first advantage it has over an architecture that is based on many different types of complex cells. The second advantage is that the regular and local interconnection of cells considerably eases the arrangement of component in space and reduces the communication latency. In the simplest case, that of linear arrays as shown in Figure 2.1, we can say that every systolic algorithm leads directly to a scheme for its implementation in silicon. Furthermore, simple and regular systems are likely to be modular and therefore adjustable to various performance goals. For example, systolic arrays are adaptable to the size and nature of the problem under attack.

Finally, regular and simple design leads the testability of systolic architecture. We can make use of the regularity of the architecture to develop a sequence of tests on the basis of the characteristics of a single cells. Then this test sequence can be applied to test all the cells of the same type.

### 2.2.2.2 Concurrency and Communication

Since the technological trend clearly indicates a diminishing growth rate for component speed, any major improvement in computation speed must come from the concurrent use of many processing elements. The degree of concurrency in a VLSI computing structure is largely determined by
underlying algorithm. Massive parallelism can be achieved if the algorithm is designed to introduce high degrees of pipelining and multiprocessing. When a large number of processing elements work simultaneously, coordination and communication become significant--especially with VLSI technology where routing costs dominate the power, time, and area required to implement a computation [Hwang84].

Systolic architecture supports high degrees of concurrency, and in the mean time employs only simple, regular communication and control to allow efficient implementation. The locality of interprocessor communications is another attractive feature of systolic architecture.

### 2.2.2.3 Balancing Computation with I/O

VLSI systems are well suited to the implementation of compute-bound computations, rather than to $I / O$-bound. In a compute-bound computation, the number of elementary computations is larger than the number of I/O operations that have to be performed. The other case is I/O-bound computation which is not suited to VLSI implementation because the number of I/O ports is limited. For example, the multiplication of two matrices of size $n \times n$ requires $O\left(n^{3}\right)$ multiplications for $3 n^{2} \mathrm{I} / \mathrm{O}$ operations-this is a compute-bound computations. On the other hand, the addition of two matrices requires $n^{2}$ additions for $3 n^{2} \mathrm{I} / \mathrm{O}$ operations, and is, therefore, I/O-bound.

Bottlenecks to speed up a computation are often due to limited system memory bandwidth, so called von Neumann bottlenecks, rather than limited processing capabilities. This problem can certainly be expected for I/O bound computations, but with a conventional architectural approach, it may be present even for compute-bound computations. For every operation, at least one or two operands have to be fetched from or stored to memory, so the total amount of I/O is proportional to the number of operations rather than the number of inputs and outputs. Thus, a problem that was originally compute-bound can become I/O-bound during its execution. This unfortunate situation is the result of mismatch between the computation and the architecture. Systolic architecture, which ensures multiple computations per memory access, can speed up computebound computations without increasing I/O requirements.

The ultimate performance goal of an array processor system is a computation rate that balance the available I/O bandwidth with the host. With the relatively low bandwidth of current I/O devices,
to achieve a faster computation rate it is necessary to perform multiple computations per I/O access. For example, the systolic array in Figure 2.1 can perform multiple computations per I/O access to the host. However, the repetitive use of a data item requires it to be stored inside the system for a sufficient length of time. In other words, the I/O problem influences not only the required I/O bandwidth but also the required internal memory. Hence, it becomes very important to arrange a computation, together with an appropriate memory structure and I/O bandwidth so that computation time is balanced with I/O time.

### 2.3 Classifications of Systolic Architectures

Systolic arrays are a new class of "pipelined" array architectures, pioneered by H.T. Kung, which are becoming increasingly attractive because of continuous advances in VLSI/WSI technology. As already mentioned, a systolic system is a "network of processors which rhythmically compute and pass data through the system". Since their introduction, systolic array architectures have undergone a rapid development. Different types of systolic arrays have been designed for a very large number of applications.

In this section, we present several classification schemes for systolic architectures. These classification schemes are based on the following aspects respectively:

- Array geometry pattern;
- Application scope;
- Implementation level;
- Granularity;
- Data/Instruction Stream;
- Design Restriction.


### 2.3.1. Array Geometry Pattern

Systolic arrays can assume many different structures for different compute-bound problems. Various systolic array

Table 2.1 Array geometry pattern schemes
$\checkmark$ Linear array
$\checkmark$ Orthogonal array
$\checkmark$ Triangular array
$\checkmark$ Hexagonal array
$\checkmark$ Tree
$\checkmark$ Other complex pattern


Figure 2.2 Systolic array configuration geometries
configurations have been developed to best match the problems under attack. The main categories of systolic arrays according to their geometry is listed in Table 2.1.

Figure 2.2 shows some systolic array configurations according to their geometric patterns. Other varieties of systolic array configurations includes torus array, ring array, trapezoid array and so on. Different geometries of systolic arrays can be used to best fit specific problems in order to achieve best desired performance. For example, a square array can efficiently implement matrix-matrix multiplication while a triangular array is good at matrix triangularization. Systolic arrays with
other geometric patterns also exist (e.g. a combination of triangular and rectangular array). Systolic arrays of higher dimensions than 2 is also possible although they are not very common.

### 2.3.2 Application scope

There exist not only algorithm specific
Table 2.2 Application scope classification scheme systolic arrays, but also class specific systolic arrays, and furthermore, general purpose systolic arrays (see Table 2.2). Although systolic arrays were initially
$\checkmark$ Algorithm specific systolic array
$\checkmark$ Class specific systolic array
$\checkmark$ General purpose systolic array introduced as algorithm specific VLSI arrays, class specific and general purpose systolic arrays have received much attention recently [Johnson93].

Algorithm Specific Array. Algorithm specific arrays are specially designed for one particular algorithm in order to achieve the best performance. By their nature, such specific architectures come in a large variety of forms.

Class Specific Array. Class specific arrays can be adapted to a variety or a class of algorithms. The adaption to different algorithms is usually through programming. The array topology of a class specific systolic array is usually fixed.

General Purpose Array. General purpose systolic arrays can be adapted to broad areas of applications via programming processing elements and reconfiguring array interconnections.

### 2.3.3 Implementation Level

Systolic arrays can be implemented in different ways according to different level of hardware to be used. Many nodes of systolic arrays can be implemented in a single VLSI/WSI package. A node can be

Table 2.3 Implementation level classification scheme
$\checkmark$ Chip level systolic array
$\checkmark$ Board level systolic array
$\checkmark$ System level systolic array
implemented by using a single chip processors, e.g. transputers, TMS320 series digital signal processors (DSP). Of course, systolic array nodes can also be composed of a number of devices and components which are accommodated in a board with more complexity (and perhaps higher performance). Table 2.3 lists the classification scheme of systolic architectures based on implementation methods.

Chip Level Systolic Array. Processor elements of a chip level systolic array are many-per-chip and are compatible with custom VLSI design. Processing elements of chip level systolic arrays are usually simple and primitive. Systolic arrays of this category are dedicated and hard-wired systems which can not be easily reprogrammed to perform other tasks besides the original task. A number of chips can be connected together to form a larger size of array.

Board Level Systolic Array. Processor elements of a board level systolic array are implemented in a single chip (maybe with a small number of other supporting devices), and are usually compatible with many commercially available microprocessors. A board level systolic array usually has a certain degree of programmability which allow different tasks to be implemented in the same architecture.

System Level Systolic Array. Processor elements of a system level systolic array are implemented in one or few circuit boards (maybe even an complete computing system). The processing elements may be relative autonomous computing units. Systolic arrays of this category usually have a small number of PEs (on the order of ten) due to the limitation of the system size, power dissipation, and interconnection problem. A high degree of flexibility for implementing different types of tasks is usually provided.

### 2.3.4 Granularity

A computation task can be performed in different

Table 2.4 Granularity classification scheme
$\checkmark$ Bit level systolic array
$\checkmark$ Word level systolic array
$\checkmark$ Block level systolic array implemented in word level, where operations of individual PEs are based on words of data, e.g.,
floating point numbers. The lowest level of parallelism for a digital system is the bit level. Systolic array also can exploit the parallelism in this level (see Table 2.4).

Bit Level Systolic Array. Bit level systolic arrays [McCanny82] [McCanny84] [McCanny86] exploit bit level parallelism with the finest granularity to perform a computational task. Bit level systolic arrays were proposed to improve the pipelining rate of systolic arrays and to better utilize the current integration level of VLSI technology.

Word Level Systolic Array. Word level systolic arrays exploit word level parallelism. The PEs of such arrays carry out operations on words of operands which consist of a number of bits.

Block Level Systolic Array. Block level systolic arrays exploit subtask-based parallelism. PEs for such arrays are usually programmable processor with significant local memory [Friedlander87] [Johnson93]. Block level systolic arrays divides applications into parallel subtasks that utilize local data. Each PE executes a series of instructions on a block of local data in a sequential/concurrent way, then communicates with its neighbours by sending and receiving the result based on block data.

### 2.3.5 Data/Instruction Stream

Inside a parallel computer architecture, there are basically two type of streams: data stream and instruction stream. If data stream systolically moves through a processing array, we called this kind of array data systolic array, or simply systolic array by dropping "data" from the term. If instruction stream systolically, i.e., in a rhythmic, regular and pipelined manner, moves through a processing array, this kind of array is called instruction systolic array.

Data Systolic Array. A data systolic array features rhythmic, local, regular and pipelined communication. The input data as well as partial results flow through the array in a systolic organization.

Instruction Systolic Array (ISA). An instruction systolic array employ processors which have simple control unit but no control core. Instructions, rather than data, are pumped through a mesh connected array of processors. In an ISA, program (set of instructions) executes as it filters through the array, making it easy to execute a pipelined sequence of programs [Megson92] [Lang86].

### 2.3.6 Design Restriction

Systolic arrays were originally designed as
Table 2.6 Design restriction classification scheme special purpose, VLSI oriented, cost effective computer systems to off-load regular, compute-bound computations [KungHT82a]. However, with the developments of VLSI technology, programming languages, systolic algorithms and computer architectures, systolic array can be implemented in many ways by releasing some design restrictions as discussed in Sec. 2.1 [D1]-[D9]. Table 2.6 gives another classification schemes for systolic arrays based on design restrictions. We then define the following classes of systolic arrays [Evans91a] [Megson92] [Bekakos80].

Hard Systolic Array. Hard systolic array is characterised by a very low degree of programmability implied by the control and switching logic built into circuits of actual designs. Such designs are highly desirable from a VLSI viewpoint but make the design inherently special purpose because of limited flexibility. Arrays of this class most likely have planar geometric patterns due to VLSI technology constraints. Broadcasting to PEs is avoided as the least amount of area in a chip is required.

Hybrid Systolic Array. Hybrid systolic arrays permit a medium degree of flexibility perhaps implemented by microprogramming or simple control tags (added to the data). Programmability is allowed so that a number of cell types can be implemented by the same processor, thereby offering greater flexibilities. Broadcasting may be allowed in some extend and the planarity restriction is relaxed.

Soft Systolic Array. Soft systolic arrays provide a high degree of programmability. The interconnection structure is used as a 'harness' or data structure expressing the amount of parallelism to be exploited rather than a machine architecture. The emphasis is on retaining systolic computation as a design principle, and the mapping of the algorithm onto an available parallel architecture. Soft systolic arrays may result a decrease in speed and efficiency because of communication and processor time sharing by simulating the array architectures. In addition, they do not have to be fabricable, but must be simulatable in some appropriate programming languages, e.g., OCCAM, CONCURRENT PROLOG. The broadcast is allowed in a soft systolic array. The restrictions on original systolic design is loosen in a great extend.

It is important to recognize that hard systolic array is a subset of hybrid systolic array, and hybrid systolic array is a subset of soft systolic array, Originally, systolic array design was restricted to the first class, but a movement to the second and the third classes comes recently as a result of commercially available general-purpose parallel processors.

### 2.4 Applications of Systolic Arrays

Systolic arrays feature the important properties of modularity, regularity, local communication, a high degree of pipelining, and high synchronized multiprocessing. Since systolic principles were introduced by H.T. Kung in late 1970s, systolic arrays have found numerous applications covering many important scientific and engineering areas. Few problems have resisted attack from systolic arrays [Quinton91].

In this section, we overview the existing systolic arrays in order to show the extend to which systolic arrays have been used. We divide the systolic applications into three domains: numerical applications, signal and image processing application, and non-numerical applications. The emphasis here is what systolic arrays are able to do. Even though all the systolic arrays we are about to discuss are not implemented (or implementable) in VLSI, they clearly show the applicability of systolic arrays to broad areas.

### 2.4.1 Numerical Applications

## Matrix Arithmetic

- Matrix-vector multiplication [KungHT79] [Evans91a]
- Matrix-matrix multiplication [KungHT79] [KungHT80a]
- Multiplication of band matrices [[KungHT80a] [KungHT85] [Navarro86]
- Matrix triangularization (LU decomposition) [KungHT80a] [Gentleman81] [Onaga86]
- Solution of triangular systems [KungHT79] [KungHT80a] [Evans83]
- Solution of linear systems [Tasic93] [Wan93b]
- Cholesky decomposition [Gentlemen81] [Schreiber84] [Moonen91b]
- QR decomposition [Gentleman81] [Heller83]
- Singular value decomposition (SVD) [Finn82] [Brent83b] [Brent84] [Brent85] [Ipsen84]
- Eigenvalue problems [Brent85][Schreiber82][Schreiber84] [Scott86] [Evans89] [Lam91]
- Matrix inversion [Rote85] [Evans91b] [Quinton91]
- Matrix permutation [Rajopadhye90]
- Matrix transposition [O'Leary87] [Rajopadhye90] [Megson92]
- Matrix exponential [Evans91c]
- Solution of Toeplitz systems [Brent83a] [Delosme86] [KungSY83]
- Least Squares [Moonen91a] [Kalson85]


## Solution of Difference and Differential Equations

- Solution of linear ODE (ordinary differential equations) [Evans87]
- Romberg integration [Evans86b]
- Boundary value problem of ODE [Evans93a]
- PDE (partial differential equations) [Law85][Melhem83]
- Multigrid method for linear PDE [Hoppe86]


## Polynomial Operations

- Polynomial multiplication [KungHT82b]
- Polynomial division [Brent83a] [Zah85] [Reilly86]
- Polynomial evaluation [Hurson86]
- Polynomial greatest common divisor (GCD) [Brent81] [Brent83a]
- Polynomial equations [Margaritis92]
- Polynomial interpolation [Schroder91]


## Discrete optimization

- Dynamic programming [Guibas79] [Luoka88]
- Algebraic path problem [Lewis86] [Lewis91] [Rote85] [Quinton91]
- Assignment problem [Schwiegelshohn88] [Megson90b]


### 2.4.2 Signal and Image Processing Applications

## Signal Processing

- Discrete Fourier Transform (DFT) [KungHT80a] [KungHT85a] [Siu91]
- Fast Fourier Transform (FFT) [Kung82b] [Ahmed82] [Alia84] [Sarkar90]
- Two dimensional FFT [Gertner87]
- Convolution [KungHT82a] [Baudet83]
- Two dimensional convolution [Baudet83] [KungHT81]
- Median smoothing [Oflazer83] [Chang90]
- FIR and IIR filter [Rao84] [KungHT80b] [Shanbhag91] [Woods90]
- Inversion of finite Radon transform
- Hardmard transform [KungSY88b]
- Walsh transform
- Kalman filter [KungSY88b] [Sung87] [Mitchell90] [KungSY91]
- Cosine transform [Sikstrom87] [Ma91] [Chang91]
- Adaptive lattice filter [Lewis90]
- Nonlinear adaptive filtering [McWhirter91]
- Speech recognition [Charot86]
- Adaptive beamforming [McWhirter83][Hargrave86][Ward84] [Takao91]
- High resolution direction estimation [Robertson91] [Wan93a]
- CFAR detection [Hwang91]
- Neural network [Cong90]
- High order statistics [Manolakos91]

Image Processing [Fu85] [KungSY88b] [Kittler86]

- Convolution [KungSY88b]
- Image enhancement [Fisher82] [Oflazer83] [Nicolas85]
- Histogram computation
- Image statistics [Fisher82]
- Contrasting
- Image segmentation [Tyagi90]
- Geometrical transformations
- Image reconstruction [Zhang90]
- Stereo-matching
- Image compression [Cappello86] [Fang91] [Panchanathan91]
- Edge detection [KungSY88b]
- Contour detection
- Connected regions detection [Nudd85]
- Hough transform [Chuang85] [KungSY88b] [VanSwaaij90]


### 2.4.3 Non-Numerical Applications

## Data structures

- Stack and queue [Guibas82]
- Searching [Bentley79]
- Priority queue
- Sorting [Ericson90]


## Graph algorithms

- Transitive closure [KungSY88b] [Quinton91] [Moreno92]
- minimum spanning trees
- connected components
- Intersection of polygons [Umeo89]
- Hidden surface removal [Das90]
- Planar convex hull [Chazelle84]


## Identification

- String matching [Apostolico84] [Megson90a]
- Pattern matching [Apostolico84]
- Maximal common substring [Apostolico84]
- Detection of repetitions in a string
- Recognition of context-free languages [Cheng85]


## Computer aided design (CAD)

- Net extraction [Kane87a]
- Lattice polygon processing [Kane87b]
- Design rule checking [Kane87c]
- Electronic circuit layout [Rutenbar88]


### 2.5 Example Systolic Architectures

In Sec. 2.4 we have discussed the applications of systolic array architectures. Numerous systolic algorithms have been developed and studied. However, very few systolic algorithms are directly mapped and implemented in hardware. This may be due to the following two factors: (a) the time that has passed since the invention of the systolic principle (1978) has not been long enough for the construction of full-scale systems [Quinton91]; (b) the VLSI technology together with the CAD has not been so advanced that a custom systolic implementation can be achieved at a very low cost in small quantities although the potential of this capability is very evident. In addition to implementing systolic array in chip level which aims at special-purpose or fixed-function architecture, more flexible and versatile general purpose systolic architectures have become a new trend for systolic implementation.

### 2.5.1 Special Purpose Systolic Arrays

Below we introduce several special purpose systolic array implemented in silicon. These systolic arrays features many processing elements in a single silicon chip by using VLSI technology. They are custom designed for each application. As special purpose arrays, they have a very small or the least degree of flexibility and programmability while aiming to acquire the best performance.

Pattern Matcher. The first attempt for implementing systolic array in a silicon chip was the pattern matcher of Foster and Kung [Foster80]. The pattern matcher systolic array was consist of
$8 \times 2$ bit-level comparators and was able to handle pattern matching problems of eight two-bit patterns. The chip was implemented using NMOS technology (not VLSI technology) with a very small number of components inside the chip.

GAPP. The first parallel processor on a chip using systolic architecture and taking the form of a VLSI device has been given the name Geometric Arithmetic Parallel Processor or GAPP by the NCR company [Davis84] [KungSY88b][Quinton91]. The chip is sealed inside a 84-bin package. GAPP contains a rectangular grid of $6 \times 12$ 1-bit microprocessors. Each of the 72 processing elements contains an ALU, 128 bits of RAM and bidirectional communication lines connected to its four nearest neighbours: North, South, East and West. All basic logical and arithmetic operations are executed within one cycle. Each PE receives its instructions as 13 bit words. A power of 28 million bit additions per second can be obtained. Many GAPP chips can be placed on a board to build up arbitrarily large arrays of processors in $6 \times 12$ increments. The packages are designed to interface directly. GAPP immediately found application in signal processing, pattern recognition and image processing.

Bit Slice Correlator. The idea of using systolic arrays at the bit level was developed by McCanny and McWhirter [McCanny82], who demonstrated that many of the components required in digital signal and information processing applications can be implemented as systolic arrays of bit level processing elements based on a gated full adder function. A high-performance bit-sliced correlator based directly on this idea has been designed by [Corry83] and further developed by Marconi Electronic Devices. The systolic chip constitutes a $64 \times 10$ bit-level processing elements to compute 64 stage correlator of 4 bit input data. The chip has been designed and fabricated in CMOS technology. It comprises approximately 43,000 transistors and can handle data at sample rates up to 35 MHz . It has been designed in such a way that it can be cascaded to increase the number of correlation stages, the reference word length and the data word length.

Systolic Array Graphics Engine. Pixel processing is the most fundamental performance bottleneck in high end two or three dimensional graphics systems. Systolic array graphics engine (SAGE) has been designed to processing pixels at extremely rapid rates. In 1984 a 2D SAGE test chip with 256 Pixel Processor was implemented at Cornell University [Gharachorloo85]. Each pixel processor contains two address comparators, colour buffer and video cells. The chip contains 25,000 of 4 micron NMOS transistors and was clocked at 100 nsec . A 3D SAGE has been implemented in

1988 by Matsushita Corp. The 3D SAGE chip was implemented with 1.2 micron CMOS technology using 330,000 transistors for 256 Pixel Processors [Nishizawa88]. Another SAGE chip was implemented with one million 1.2 micron CMOS transistors and clocked at 40 nsec [Gharachorloo88].

Cellular Rotator Array. In image processing, real-time projection for an image of $1 \mathrm{~K} \times 1 \mathrm{~K}$ pixels requires a huge amount of computations (a lot of Givens rotations). A cellular rotator systolic array chip was designed by Burleson and Scharf [Burleson88] to solve this computationally intensive problem. The chip used CMOS technology and had a clock speed of 32 MHz . It contains 138,000 transistors in a $5 \times 3$ array of rotators, each of which performs rotations at 2 MHz (for 16-bit data) using bit-serial CORDIC (COordinated Rotation DIgital Computer) arithmetic. The resulting performance is 180 Mfrops (Million Fractional Operations Per Second), or a rotation rate of 30 MHz . It is worth to mention that the custom design of the chip took only 10 man-week for logic design and 8 man-week for full custom design and verification due to the use of systolic principle.

HDTV Filtering Chip Set. High Definition Television (HDTV) will require calculation intensive image processing algorithms for real-time operations. A chip set optimized for large kernel 2D transversal filters, including a programmable delay line and a filter chip has been designed [Joanblanq90]. The filter chip is made of a mono-dimensional array of 8 multiply-add cells with 8 -bit data, 12 -bit coefficients and 30 -bit full precision output. The delay line chip perform delays from 4 to 2051 clock cycles for an 8 -bit data word. By using 8 of the filter chips and 15 of the delay line chips, a systolic array architecture for $16 \times 16$ symmetric filter can be constructed. Both the chips have been implemented in the $1 \mu \mathrm{~m}$, double metal CMOS technology at 54 MHz . A single filter chip can achieve a performance of 1.7 giga MAC (Multiply-ACcumulate) per second.

### 2.5.2 General Purpose Systolic Arrays

The more specialized the hardware, the higher the performance; but cost per application also rises and flexibility decreases. Therein lies the attractiveness of general purpose systolic architectures. Table 2.7 lists most of the recent programmable and reconfigurable, general-purpose systolic arrays reported in literature. Below we will give a brief introduction to some of these general purpose systolic arrays.

| System Name Developer | Development Stage | Topology | Key Features | References |
| :---: | :---: | :---: | :---: | :---: |
| Programmable Systolic Chip Architecture Carnegie Mellon University | Prototype | Linear <br> 9 Cells | Early predecessor of Warps; 8 bit fixed-point ALUs | [Fisher84] |
| Warp Carnegic Mellon University | Commercial | Linear 10 Cells | 32-bit floating-point multiplication; block processing; 100 mflops | [Annara 87$]$ |
| iWarp Carnegie Mellon University | Prototype | $8 \times 8$ array | Warp cell without on-chip memory; expandable to 1024 cells | [Borkar88] |
| MICMACS IRISA, France | Prototype | Linear <br> 18 cells | 16-bit fixed point math; broadcast data; 90 mops | [Frison89] |
| Systolic/Cellular Architecture Hughes Research Laboratories | Prototype | $16 \times 16$ array | 32-bit fixed-function units | [Przytila88] |
| Saxpy-IM Computer Corp. | Commercial | Linear 32 cells | 32-bit floating-point capability; broadcast and global data; block processing; I gflops | [Foulser87] |
| Computer for Experimental SAR Norwegian Defense Res. Estab. | Prototype | $\begin{aligned} & \text { Four } 8 \times 16 \text { arrays } \\ & 512 \text { cells } \end{aligned}$ | Bit-serial cellular I/O; 32-bit floating point multipliers in each cell; 320 mflops | [Toverud88] |
| Cellular Array Processor Fujisu Laboratories, Japan | Commercial | $16 \times 16$ array | Block processing; floating point math; image oriented | [Ishii88] |
| PICAP3 <br> University of Paris | Prototype | $8 \times 8$ array | 16-bil word length: image orienled | [Lindscog86] |
| $\begin{aligned} & \text { PSDP } \\ & \text { Univ. of South Florida } \end{aligned}$ | Research |  | 4-bit word length; wafer-scale design | \|Landis91] |
| Cylindrical Banyan Multicomputer <br> Univ. of Texas al Austin | Research |  | Packel-switched programmable topology with programmable cells | [Malek89] |
| Splash Systolic Engine <br> Super Comp. Research Centre | Prototype | Linear <br> 32 cells | FPGA-reconfigurable cell architecture based on commercial chip | [Gokhale90] |
| Cellular Array Logic Edinburgh University, UK | Prototype | . $16 \times 16$ array | FPGA-reconfigurable cell architecture based on commercial chip | 〔Kean89] |
| Programmable Adaptive Computing Engine University of Wales, UK | Research |  | Function units embedded in each cell; reconfigurable connections in cell as well as programmable topology | [Jones89] |
| Configurable Functional Array Tsinghua University, Beijing | Research |  | Configurable array topology and cell architecture | [Wenyang91] |

Programmable Systolic Chip Architecture. The systolic model was born at Carnegie-Mellon University in 1978. To show its validity, H.T. Kung and his team developed the programmable systolic chip (PSC) architecture. The PSC was designed to be a circuit which is suitable for systolic architecture and is sufficiently flexible for use as a building block for different types of arrays. The PSC exhibited the following features:

- Three 8-bit input ports and three 8-bit output ports for data;
- Three 1-bit input and three 1-bit output control ports;
- One 8-bit ALU capable of double precision arithmetic
- Multiplier-accumulator
- Sixty-four word ( 60 -bits wide) program memory
- Sixty-four 9-bit words of internal register storage
- Three 9-bit buses

A demonstration system was constructed around a SUN workstation host. In May 1984, a linear array of 9 PSCs could perform a bidirectional convolution of size $512 \times 512$ every 1.8 seconds using 8 -bit data.

WARP. Warp is a systolic machine composed of 10 cells, each of which has a power of 10 Mflops. Warp is connected to a UNIX workstation and is accessed via procedure calls. A highlevel language, called W 2 , is used to program the machine. The principal application domains for Warp are scientific computation, signal and image processing.

The design of the Warp machine began in 1984 and the first prototype was operational in 1986. The machine has been commercialised by the General Electric Company since 1987.

Figure 2.3 shows the structure of Warp. It contains three elements: the Warp itself, the interface and the host. The Warp processors perform the


Figure 2.3 The Warp machine architecture
computation. The interface controls inputs and outputs between Warp and the host, and generates addresses and data, and handles the results returned by Warp. Furthermore, the host executes those parts of the application program that cannot be handled by the Warp processor, for example taking global decisions, or the control of certain scientific routines.

Each cell is a microprogrammable processor with horizontal microcode. It contains its own sequencer as well as 8 K microcode instruction memory. There are two communication channels X and Y for each cell. Addresses and control signals pass along the dedicated address channel Adr. The direction of the flow along the Y channel can be reconfigured for the specific algorithm being executed. The interface handles all communication between the host and the Warp processor and provides all the control signals necessary to drive the Warp array. The host consists of a Sun workstation which acts as the master controller and of a microprocessor called the external host whose role is to control peripherals and to manage the data required by the computations performed by Warp.

The performance of the Warp is impressive when one takes its cost into account. The speed-up with respect to a VAX 11/780 varies between a few dozen and several hundred times, according to the type of algorithm. Statistics obtained from 72 different program have shown that Warp achieves at least 28 Mflops, with its maximum speed being around 100 Mflops.
iWarp. The iWarp system is a distributed memory, multiple-instruction-multiple-data (MIMD) multicomputer. A 2D toroidal array of cells (Figure 2.4) characterizes the system architecture. This architecture supports both message-passing and systolic communications efficiently [Peterson91].


Figure 2.4 iWarp architecture

Each cell holds one iWarp microprocessor and its local memory. The iWarp microprocessor contains an integrated communication agent, allowing an iWarp cell to connect to each of its four nearest neighbours via eight $40 \mathrm{Mbytes} / \mathrm{s}$ buses (four input and four output). The total communication bandwidth is $320 \mathrm{Mbytes} / \mathrm{s}$. Each processor connects to its local memory via a 64bit bus with $180 \mathrm{Mbytes} / \mathrm{s}$ bandwidth. The iWarp microprocessor is a very powerful processor which is capable of 100 million-operations-per-second (MOPS) per cell, including 20 Mflops of single precision, IEEE floating-point operations. The chip holds more than 650,000 transistors and is fabricated in a $1 \mu \mathrm{~m}$ CHMOS IV, two-layer metal process. The chip measures $14.55 \mathrm{~mm} \times 13.46 \mathrm{~mm}$, packaged in a 271 -pin PGA. The first such chip was fabricated in December of 1989 .

The iWarp consists of $8 \times 8$ iWarp cells with a peak performance of 1.28 Gflops. The iWarp can be expandable to 1024 cells with a peak performance of 20 Gflops.

The iWarp machine can be programmed to operate as a pool of individual communicating processors using standard high-level language C and Fortran or as an array using parallel program generators (PPG). The PPGs use knowledge of a specific application area, such as image processing or signal processing, to automate the detailed implementation of programming parallel machines.

Cellular Array Processor (CAP). Cellular array processor developed by Fujitsu Laboratory has a standard configuration of 256 cells in a two dimensional $16 \times 16$ array. The array topology resembles the surface of a torus. Each cell is connected to four adjacent neighbours; cells at the boundaries are connected to cells at the opposite sides. In addition to intercell connections,


Figure 2.5 CAP architecture
the common command bus links all cells to the host computer, and the common video bus links all cells to the video interface (Figure 2.5).

Each cell in CAP consists of a general purpose microprocessor, local memory, and a CAP-VLSI chip. The CAP-VLSI chip is a key processor component. It has two independent common bus interfaces for data broadcasting and six serial communication ports for local data communication. The chip also has realtime image data handling capabilities [Ishii88].

CAP can perform ray tracing very efficiently, and the performance increases linearly with the number of cells. In addition to image generation, CAP can also find applications in VLSI logic simulation, routing, and placement.

### 2.6 Systolic Array Design Methodologies

One of the important design problems in systolic array processing is the development of a systematic methodology for transforming an algorithm represented in some high-level constructs into a systolic architecture, or systolic algorithm, specified by the timing of data movements and the interconnection of processing elements such that the design requirements are satisfied.

Different design methodologies have been proposed for the systolic architecture design. Among other issues, these methods differ in the way that the original algorithms are represented. Nineteen systematic methods for synthesizing systolic arrays were reviewed by Fortes et al. in [Fortes88]; new ones have been proposed since then. The original algorithm can be represented in the forms of algebraic expressions, graphical descriptions, and high-level languages. Accordingly, various methodologies can be grouped into the following classes:

Algebraic Expressions. Algebraic descriptions of algorithms include vector operators [Gannon82], a canonical algebraic representation [KungHT83], recurrence equations [Miranker84] [Li83] [Cappello83] [Quinton84] [Delosme86], regular iterative expressions [Jagadish87] [Rao88], indexes of nested loops [Moldovan83] [Moldovan86] [Fortes85] [Lee88], space-time transformations [Cappello84], and affine recurrence equations [Yaacoby88].

Graph-based Description. Graphical notations to describe an algorithm include signal flow graphs (SFG) [KungSY88b], dataflow graphs [Koren83], multimesh graphs [Moreno92], and other types of graphs [Schwartz84] [Ramakrishnan83].

High Level Languages. The use of high level languages as description tool has been reported in [Lam85] [Chen86] [Chapman86].

the use of a Figure 2.6 The stages in a transformational design methodology transformational approach: systolic architectures are derived by transforming the original algorithm description that is unsuitable for direct VLSI or systolic implementation. Starting from a representation of an original algorithm, a transformational design methodology consists of two stages [Moreno92]: algorithm regularization and array mapping (see Figure 2.6).

The regularization stage corresponds to the derivation of a regularized representation of an algorithm from an original admissible form. The regularized representation has as its objective the description of the algorithm in a manner suitable for manipulation in the remaining steps of a particular method. Consequently, different approaches are characterized by different regularized representations.

On the other hand, the mapping stage uses the regularized description to determine the topology and structure of the array, the characteristics of cells, the allocation of data and operations to cell, the dataflow, the I/O, the control and so on.

The two most popular types of algorithm representation are algebraic expressions and graphical descriptions. In algebraic based methods the regularized descriptions given as a set of algebraic expressions, and transformations are applied to these expressions to obtain an implementation. Common types of algebraic descriptions are regular iterative algorithms (RIA) [Jagadish87]
[Rao88], uniform recurrence equations (URE) [Quinton84], affine recurrence equations (ARE) [Yaacoby88]. We will discuss the RIA method in Chap. 3 in detail.

A different line of research uses graphical notations to describe an algorithm. Examples are the signal flow graph (SFG) method [Kung88b], and the multimesh graph (MMG) [Moreno92]. Graph-based methods represent an algorithm as a graph, and apply transformation to the graph to render it more suitable for later steps. The regularized graph is then mapped onto an array either directly or through other intermediate representations. Chap. 4 will discuss the SFG method.

The above mentioned design methodologies allow systematic synthesizing for systolic arrays. Another important issue for systolic array design is the automation of these transformational design methodologies. A number of ongoing attempts to develop synthesis software tools have been made. Some most notable attempts among these are ADVIS [Moldovan87], DIASTOL [Gachet86], SDEF [Engstrom87] and VACS [KungSY88c]. Systolic synthesis tools are still at early stages of development and a fully integrated system is still some way off although some progress towards usable software tools is gaining momentum [Megson92].

### 2.7 Some Related Issues to Systolic Architecture

### 2.7.1 Performance of Systolic Architecture Versus Other Parallel Architectures

There are a number of measures for the performance of a parallel architecture. Some important measures include performance, cost, and performance-cost ratio. The performance-cost measure may be the most important consideration when comparing different architectures. When considering the cost-performance measure of a parallel architecture, speedup factor and efficiency of the architecture are usually used to measure the performance of a parallel architecture. Below we will mainly concern with the speedup and efficiency of a parallel architecture. This by no means indicates that other measures are not important.

The speedup of a parallel architecture can be defined as the ratio of its processing time to that of a sequential, single-processor computer for a given problem:

$$
\begin{equation*}
S=T / T_{p} \tag{2.1}
\end{equation*}
$$

where $T_{s}$ is the time on a sequential computer and $T_{p}$ is the time on a paraliel computer.

The efficiency of a parallel computer can be defined as the ratio of its speedup to the number of processors employed in the parallel computer:

$$
\begin{equation*}
E=\frac{S}{N}=\frac{T_{s}}{T_{p} \times N} \tag{2.2}
\end{equation*}
$$

where $N$ is the number of processors employed in the parallel computer.

The ideal performance characteristics for an $N$ identical processor system, on which a given problem could be partitioned into $N$ similar tasks, would be the linear relationship between speedup versus the number of processors. The efficiency of such an ideal parallel computing system is $100 \%$.

The most straightforward approach to parallel architectures may be to connect a number of CPUs to a common bus. However, speedup of such a parallel system improves linearly with the number of processors only up to the point that bus-contention problems become the limitation. Minsky's famous conjecture is that, for a broad range of algorithms, the conflict between $N$ processors for access to shared resources along the common bus limits the speedup improvement to $\log _{2} N$.

Modern supercomputer designers have utilized a number of parallel processing stratagems to improve on this state of affairs and are achieving performance improvements commensurate with Amdahl's law [Whitehouse85]. Consider a computing problem, which can be executed by a uniprocessor in unit time, $T_{s}=1$. Let $f_{i}$ be the probability of assigning the same problem to $i$ processors working equally with an average load $d_{i}=1 / i$ per processor. Furthermore, assume equal probability of each operating mode using $i$ processors, that is $f_{i}=1 / N$, for $N$ operating modes: $i=1,2, \ldots, N$. The average time required to solve the problem on an $N$-processor system is given below:

$$
\begin{equation*}
T_{p}=\sum_{i=1}^{N} f_{i} \cdot d_{i}=\frac{1}{N} \sum_{i=1}^{N} \frac{1}{i} \tag{2.3}
\end{equation*}
$$

The average speedup is obtained as the ratio of $T_{s}=1$ to $T_{p}$; that is

$$
\begin{equation*}
S=\frac{T_{s}}{T_{p}}=\frac{N}{\sum_{i=1}^{N} \frac{1}{i}} \leqslant \frac{N}{\ln N} . \tag{2.4}
\end{equation*}
$$

Systolic architecture, however, yields a perfectly speedup improvement factor of $N$, or a perfect constant efficiency regardless of the number of processors. We will exemplify this attractive characteristics of systolic architecture in Chap. 3 when we analyse the performance of various systolic arrays for matrix multiplication. Figure 2.6 graphically depicts the speedup achieved through parallelism for the three cases of Minsky's conjecture, Amdahl's law, and systolic arrays.

### 2.7.2 Area-Time Complexity Theory



Figure 2.6 Speedup of $N$-processor parallel system over a single processor

Complexity theory attempts to provide systematic information about algorithmic complexity in order to estimate the algorithm complexity quantitatively. A model for VLSI computation is often based on a "grid model" [Preparata84]. Area-time complexity measures have received special attention. They depend on two factors, computation time ( $T$ ) and chip area ( $A$ ) . Generally, areatime measures $f(A, T)$, can be defined as:

$$
\begin{equation*}
f(A, T)=A^{\propto} T^{\beta} \tag{2.5}
\end{equation*}
$$

where $\alpha$ and. $\beta$ are nonnegative constants. The above area-time measure is a monotonic and quasi-homogeneous function of $A$ and $T$ [Frumkin92]. The monotonic property indicates that the area-time measure satisfies:

$$
\begin{equation*}
f\left(A_{1}, T_{1}\right) \geq f(A, T) . \quad \text { if } A_{1} \geq A \text { and } T_{1} \geq T \tag{2.6}
\end{equation*}
$$

The quasi-homogeneous property indicates that the complexity measure is independent on units of the scale we measure $A$ and $T$, i.e.,

$$
\begin{equation*}
\frac{U\left(\xi A_{1}, \zeta T_{1}\right)}{U\left(A_{1}, T_{1}\right)}=\frac{U\left(\xi A_{2}, \zeta T_{2}\right)}{U\left(A_{2}, T_{2}\right)} \tag{2.7}
\end{equation*}
$$

If $\alpha=1$ and $\beta=0$, we have $f(T, A)=A$. This measure can be used in the design of calculators because we do not worry about time but to reduce the size and weight as much as possible. If $\alpha=0$ and $\beta=1$, we have $f(T, A)=T$. This measure can be used in the design of devices of a supercomputer where we want to reduce processing time at any cost. If $\alpha=1$ and $\beta=1$, we have $f(T, A)=A T$. This measure takes into account the area and the processing time to the equal extent.

In the literature on VLSI complexity, the following complexity measure has been accepted as a reasonable performance measure for VLSI algorithms [McEvoy91]:

$$
\begin{equation*}
f(A, T)=A T^{2} \tag{2.8}
\end{equation*}
$$

Systolic arrays and systolic algorithms are also based on "grid model". Area-time measures, especially the $A T^{2}$ measures, are useful for VLSI implementation of systolic designs. These measures can be also used to measure the lower and upper bounds of many computation algorithms.

### 2.7.3 Systolic Arrays Versus Wavefront Arrays

As mentioned in Chap. 1, wavefront arrays combine systolic data pipelining with an asynchronous dataflow execution paradigm. Both systolic arrays and wavefront arrays are characterized by regularity, modularity, locality and pipelinability. A simple way to compare systolic arrays and wavefront arrays is [KungSY88b]:

Wavefront array $=$ systolic array + data flow computing
Systolic arrays are defined in a strict sense as a synchronous architecture, although many implementation of systolic arrays (especially those general purpose systolic arrays) have adopted asynchronous control mechanisms. Since clocking scheme is a critical factor for large-scale systems, the global synchronization often incurs severe hardware burdens in terms of clock skew, that is, each PE in the array may not receive the clock signal at the same time. This may be due to the different path lengths from the clock generator to each PE, or other reasons, such as process variations for different clock paths. To overcome clock path problems, special clock schemes have been developed. For example, an H -tree scheme can be used to distribute the clock signal to
regular arrays such that every PE has the same distance from the clock generator as illustrated in Figure 2.7.

Though H-tree schemes can solve the clock path length problem, the issue of the clock skew problem is not completely resolved. The timing analysis of H -tree clocking distribution synchronous system showed [KungSY82b] that the clock skew time are of $O\left(N^{3}\right)$ where $N$ is number of PEs in the array. This will degrade systolic array efficiency very significantly when $N$ is large.

Wavefront arrays adopt asynchronous scheme without involving global clock. Information


Figure 2.7 H-tree layout for clocking a square array transfer in wavefront arrays is by mutual convenience and agreement between each processing element and its immediate neighbours. Whenever the data is available, the transmitting PE informs the receiver of that fact, and the receiver accepts the data whenever it is convenient for it to do so. This scheme can be implemented by means of a handshaking protocol. Wavefront arrays may be extended indefinitely without degrading performance.

In general, a systolic array is useful when the PEs are simple and primitive modules, since the handshaking hardware in a wavefront array would represent a non-negligible overhead for such applications. On the other hand, a wavefront array is more applicable when the modules of the PEs are more complex (such as floating-point multiply-and-add), when synchronization of a large array becomes impractical.

### 2.8 Summary

Systolic arrays were first introduced by H.T. Kung and Leiserson as high-performance, specialpurpose VLSI computer systems. A systolic array is a network of processors which rhythmically
compute and pass data through the system. A systolic array is characterised by the combination of the following features: synchrony, modularity, regularity, locality, and pipelinability.

Systolic arrays adopt simple and regular design and are best amendable to VLSI implementation. Systolic arrays employ massive parallelism of multiple processing elements to achieve a linear rate of speedup. They also use regular and local communication and can be expanded to different sizes for different performance. Systolic arrays balance computation with I/O by performing multiple computations per I/O access and are applicable to many compute-bound problems.

Various systolic arrays have been designed and studied in the last two decades. Systolic arrays have found numerous applications in many important scientific and engineering areas such as matrix computations, real-time signal processing and image processing.

Some systolic arrays have been implemented in dedicated VLSI devices in order to achieve the best performance possible. Programmable systolic arrays and reconfigurable systolic arrays are very appealing when system flexibilities are considered. Systolic principles can also be applied to general purpose computers and some commercially available systems of such have been developed. While implementing systolic arrays on silicon can maximally utilizing the hardware resources to achieve the best performance for a specific problem, programmable and reconfigurable general purpose systolic arrays provide flexibility and are capable of solving various kinds of computationally intensive problems.

Systematic synthesizing systolic array has received much attention and different kinds of synthesizing methods have been developed with different kind of capabilities. Most of the popular systolic array design methodologies use transformational approach. Automation of synthesizing procedure is still a challenging task and some attempts to develop synthesis software tools are ongoing.

Featuring modularity, regularity and local communication, systolic arrays are amendable to VLSI implementation. Systolic arrays combine pipelining, array-processing and mutiprocessing to produce a high-performance parallel computer system. Systolic arrays provide an encouraging solution for many scientific and engineering problems such as real-time signal and image processing, artificial intelligence, and large scale scientific computations.

## Chapter 3

## SYSTOLIC ARRAY SYNTHESIS USING RIA REPRESENTATIONS

### 3.1 Introduction

THE evolution in VLSI technology has had a great impact on computer architecture. Many existing algorithms in matrix computations, pattern recognition, signal and image processing can be implemented on a VLSI chip using multiple, regularly connected processing elements (PEs) to exploit the great potential of pipelining and multiprocessing. This type of array processor is referred to as systolic array. In addition to implementing systolic arrays in VLSI devices, systolic approach is in fact has a wider applicability. There exists a large number of systolic algorithms that is not practical to map directly onto hardware in order to produce a special-purpose device, but they perform very efficiently when implemented on appropriate parallel computers. Therefore, systolic array or systolic algorithm design plays an important role in VLSI implementation of an algorithm as well as in efficient mapping of an algorithm to an existing parallel computing architecture.

One of the important design problems in systolic processing is the development of a systematic methodology for transforming an algorithm represented in some high-level constructs into a systolic architecture specified by the timing of data movement and the interconnection of processing elements such that the design requirements are satisfied [Fortes88].

The common characteristic of most proposed methodologies for systolic array design is the use of transformational approach-i.e., systolic architectures are derived by transforming the original algorithm descriptions that are unsuitable for direct VLSI or systolic implementation. Distinct transformational systems for systolic architecture design can be characterized by how algorithms are described, what formal models are used, how systolic architectures are specified, and what types of transformation are used on and between these representations.

The transformational systolic architecture design systems can be visualized as a three-dimensional space, where dimensions (or axes) are associated with the algorithm representation, algorithm model, and architecture specification. To the axis of algorithm representation, we associate different forms or levels to present an algorithm to the transformational system. The axis of algorithm model shows different levels of abstraction used to represent relevant features of the algorithm. The axis of architecture specification is associated with the hardware model or level of design in which the systolic array is described [Fortes88].

This three dimensional space can be graphically depicted as a Y chart as shown in Figure 3.1, where directed arcs can be drawn to illustrate transformations that map a given representation into another representation in the same axis and level (a self loop), in the same axis and different level, or between distinct dimensions.

The systolic array design Figure 3.1 Y chart for transformational systems. methodology of using regular iterative algorithm (RIA) were first introduced by S.K. Rao [Rao85]. This methodology starts with the so-called RIA and then extracts an index space and a reduced dependence graph (RDG). Next, an iteration space is selected. A schedule can be obtained by finding the solution of a linear programming problem. The compatibility between a schedule and iteration space can validate if a feasible design can be achieved. The array topology can be obtained by mapping the points on the index space into the processor space which is a complementary space to iteration space. The interconnections between processors can be determined by mapping the index displacement vector to the processor space. Figure 3.2 shows the Y chart of this methodology.

The RIA design methodology for systolic arrays are similar to the methodologies independently derived by several authors in [Moldovan83] [Quinton84] [Li85]. The RIA methodology generalizes and extends this class of methodology by introducing the iteration space and the dependencies of different variables in the iteration.


Figure 3.2 Overview of RIA design methodology

Because a wide range of engineering problems are shown to be members of regular iterative algorithms, the RIA methodology can be applied to systematically designing systolic architectures for many applications. The uniform recurrence equations (UREs) of Quinton's method, another wellknown algebraic transformational approach for systolic array synthesis [Quinton84] [Quinton91], are a subclass of RIAs.

The RIA design methodology can produce multiple choices of architectures each with its latency, iteration interval, and schedule for a given algorithm. The design process of RIA methodology can be fully automated.

In the following section, we first define the regular iterative algorithm (RIA) and introduce some other basic concepts. Algorithms to solve a wide range ofengineering problems are shown to be member of this class. In Sec. 3.3, a systematic procedure is described to obtain a variety of systolic array architectures suitable for implementing a given regular iterative algorithm. In Sec. 3.4, an example of the use of this design methodology to the matrix-matrix multiplication problem is presented and different systolic designs are produced by this methodology. Sec 3.5 presents the performance analysis on the various systolic designs for the matrix-matrix multiplication algorithm. Sec. 3.6 gives a formal definition of a systolic array using the RIA concepts. Sec. 3.7 summarizes the chapter.

### 3.2 Basic Concepts and Definitions

### 3.2.1 On the Exploitation of Parallelism in Algorithms

An Algorithm takes concrete form only through the language expressing it. It has been recognized for some time that for the purpose of extracting the parallelism in an algorithm, standard sequential programming languages such as Fortran and Pascal are ill-suited vehicles for expressing the algorithm. An algorithm written in these languages has built-in ordering of computations which most often obscures any parallelism present in the algorithm. Furthermore, ever since the days when core memory was a costly resource to be sparingly used, one has been conditioned to think in terms of minimizing the storage required by the program, and hence encouraged to overwrite on variables as much as possible. Such overwriting further compounds the problem of extracting the parallelism from the program.

Single Assignment Algorithm. The so-called Single Assignment Language, for example, provides the means for overcoming the difficulties mentioned above by requiring that every variable defined in the program takes on a unique value during the course of computation. Thus assignment statement of the form " $a:=a+b$ " is not allowed since the variable $a$ appears on both sides of the statement. If an algorithm is expressed as a Single Assignmem Algorithm, viz., as a program in the single assignment language, then one can conceive of automated procedures for extracting the parallelism in the algorithm, with no further effort required of the user.

Dependence Graph. Given a single assignment algorithm, it is possible to capture the information regarding the parallelism in the algorithm by means of a dependence graph. This graph has one node for each of the variables in the algorithm and a directed arc from node $x$ to node $y$ if and only if variable $y$ is computed using the value of $x$ in the algorithm. The dependence graph of a single assignment algorithm specifies a partial ordering among the computations in the algorithm; that is, if there is a directed path in the dependence graph form node $x$ to node $y$, then the computation represented by node $y$ must be executed after the computation represented by node $x$ is completed, no matter how many processors are brought to bear upon the problem. In such a case, one would say that $y$ is dependent upon $x$, and if a path from $x$ to $y$ is an edge, this dependence is direct. From this observation, one can infer that the length of the longest path, also referred as the critical path
[Jagadish87] in the dependence graph is a lower bound for the total time required for executing the algorithm, independent of the number of processors used in this execution.

Suppose that one wishes to obtain an implementation of the algorithm that is optimal with respect to the total time required for executing the algorithm. One simple and brute-force method for achieving this object is to use a distinct processor for executing the computation represented by every node in the dependence graph. This, in general, leads to a very inefficient use of the computational resources, since each processor is active only for a constant period of time, which could be a minute fraction of the time required for completing the algorithm. To achieve a better utilization of these resources, it is necessary to reuse the same processor for handling a large number of computations. In general, the set of computations can be arbitrarily partitioned and assigned to different processors.

Schedule. In determining an implementation for the algorithm, one must not only specify the processor at which each computation is to be performed, but also assign a time at which it is to be executed by the processor. This mapping of computations into time slots is referred to as the construction of a schedule for the computations. A schedule must satisfy the precedence constraints imposed by the dependence graph of the algorithm and must also be such that no two computations assigned to the same processor are expected to be executed at the same time. A schedule must also take into account the communication constraints among the processors. That is, if variable $x$ is computed by processor $p_{x}$ and if $x$ is required as an input to the computation of variable $y$ at processor $p_{y}$, then the schedule for execution of $y$ must include the time required to communicate the value of $x$ from $p_{x}$ to $p_{y}$. Clearly, for different partitions of the nodes in the dependence graph, the interprocessor communications required will differ in general.

The problem of determining an optimal schedule, i.e., one that minimizes the total time for the execution of the algorithm, is extremely hard even if the interprocessor communication is assumed to be instantaneous. Indeed, it has been proved to be NP-complete even in the presence of many simplifying assumptions. If communication constraints have to be taken into account as well, then the problem becomes even more intractable, thereby forcing one to seek ways of exploiting any available structures in the algorithm.

Regular Iterative Algorithms. Regular iterative algorithms (RIAs) are a special subclass of single assignment algorithms for which many of the difficulties discussed above can be successfully
overcome. Indeed, for an RIA, one can ensure that all computations assigned to the same processor can be described by the same simple instruction. For instance, if this instruction is a multiply operation, then one can replace this processor by a simple serial multiplier element. Furthermore, for an RIA, one can ensure that the interprocessor communication required is fixed and can be implemented using a few dedicated links. A further attraction of this class is that the schedule for the algorithm can be constructed to be "periodic" so that the necessary delays on the interprocessor links can be implemented using shift registers and Last-In-First-Out buffers alone, without any additional control circuitry. Finally, RIAs form an extremely useful subclass of single assignment algorithms.

### 3.2.2 Definition of Regular Iterative Algorithm

Below, we will present a formal definition of Regular Iterative Algorithm and then provide some examples of RIAs.

Definition: A regular iterative algorithm is defined by the triple $\{\boldsymbol{I}, \boldsymbol{X}, \boldsymbol{F}\}$ where
$I$ is the index space which is the set of all lattice points enclosed within a specified region in $S$-dimensional Euclidean space,
$X$ is the set of $V$ variables that are defined at every point in the index space, where the variable $x_{j}$ defined at the index point $\boldsymbol{k}$ will be denoted as $x_{j}(\boldsymbol{k})$ and takes on a unique value in any particular instance of algorithm, and
$F$ is the set of functional relations among the variables, restricted to be such that if $x_{i}(k)$ is computed using $x_{j}\left(\boldsymbol{k}-\boldsymbol{d}_{j i}\right)$, then
$\boldsymbol{d}_{j i}$ is a constant vector independent of $\boldsymbol{k}$ and the extent of the index space, and for every $/$ contained in the index space, $x_{i}(\boldsymbol{l})$ is computed using $x_{i}\left(\boldsymbol{l}-\boldsymbol{d}_{j i}\right)$ (if $x_{j}\left(\boldsymbol{l}-\boldsymbol{d}_{j i}\right)$ falls outside the index space, then this is an external input to the algorithm).

Central to the concept of a regular iterative algorithm is the notion of the Index Space. It is specified by a set of constraints such that any integer vector $i=\left[i_{1}, i_{2}, \cdots, i_{s}\right]^{T}$ belongs to the index space if it is a "feasible" vector for the set of constraints. Such a vector is said to represent an index point, and is also referred to as an index vector. For example:

$$
\begin{equation*}
i=\left[i_{1}, i_{2}, \cdots, i_{s}\right]^{T} \in I, \quad \text { if and only if } 0 \leq i_{j} \leq N \text { for all } j \tag{3.1}
\end{equation*}
$$

defines an index space that is the $S$-dimensional hypercube of length $N$.

At any particular point in the index space, every variable assumes a unique value that is either an input to the algorithm, or is computed as a function of the values of variables at other index points in the index space. Thus the defining equation for a variable, say $x_{j}$, at the index point $\boldsymbol{k}$ can be expressed as

$$
\begin{equation*}
x_{j}(k)=f_{k, j}\left(x_{1}\left(k-d_{11}\right), x_{1}\left(k-d_{12}\right), \cdots, \quad x_{\nu}\left(k-d_{V 1}\right), x_{V}\left(k-d_{V 2}\right), \cdots\right) \tag{3.2}
\end{equation*}
$$

where the constraint of regularity requires that the vectors $d_{j i}$ are independent of the index point $k$ and the extent of the index space $I$. These vectors $\boldsymbol{d}_{j i}$ will be referred to as index displacement vectors in what follows.

The set of equations for evaluating each of the variables at a particular index point is referred to as an iteration umit. It should be noted that the function $f_{k, j}$ in Eq. (3.2) could vary with $k$ and thus the regularity constraint that we impose upon the algorithm is only on the dependence relations and not on the functional relations. Furthermore, the function $f_{k j}$ could involve conditional, provided the differences between branches are not in the dependencies. For instance

$$
\begin{gather*}
x(i, k)=x(i-1, k)+k^{2} x(i, k-1)^{y(i+1, k-1)} \\
y(i, k)= \begin{cases}i^{k} v(i-2, k+1)+x(i, k), & \text { for even } i \\
-i^{k} y(i-2, k+1)+x(i, k), & \text { for odd } i\end{cases} \tag{3.3}
\end{gather*}
$$

is a regular iterative algorithm whereas

$$
\begin{equation*}
x(i, k)=x(i / 2, k / 2)+x(i-1, k) \tag{3.4}
\end{equation*}
$$

is not. Following are a few more examples of RIAs.

Example 1. An urn contains $N$ red balls and $N$ green balls. The following experiment is conducted repeatedly until the urn is empty or exactly one ball remains.

Two balls are picked at random from the urm. If they are of the same colour, then one of these is replaced in the urn. If they are of different colours, then both are discarded.

To determine the probability that the urn is empty at the end of the experiment, one can derive a recursive algorithm, using elementary counting arguments. Let $p(i, j)$ denote the probability that the urn becomes empty if there are $i$ red balls and $j$ green balls to begin with. Then

$$
\begin{equation*}
p(i, j)=\frac{i(i-1) p(i-1, j)+j(j-1) p(i, j-1)+2 i j p(i-1, j-1)}{(i+j)(i+j-1)}, \quad 1 \leq i, j \leq N \tag{3.5}
\end{equation*}
$$

with $p(0,0)=1, p(i, 0)=0$ for all $i>0$ and $p(0, j)=0$ for all $\rho>0$.

Example 2. Consider the following simple sorting algorithm referred to as selection sort. Given a list of $N$ numbers $\{x(i)\}$, first determine the largest number in the list and delete it from the list. Then from the ( $N-1$ ) numbers in the remaining list, delete the largest number and so on iteratively until the list is empty.

To write this algorithm in single assignment form, let the $j$ th instance of the list be given by $\{x(i, j)\}$, where i ranges from 1 to $(N-j)$. In addition, let $m(i, j)$ be the largest number in the segment $\{x(k, j), k=1$ to $i\}$ of the list. Then

$$
m(i, j)=\left\{\begin{array}{ll}
x(i, j), & \text { if } i=1  \tag{3.6}\\
\max \{m(i-1, j), x(i, j)\}, & \text { otherwise }
\end{array}\right]=\left\{\begin{array}{ll}
\text { if } i=1
\end{array}\right] \begin{array}{ll}
\operatorname{undefined}, & \text { otherwise } \\
\min \{m(i-1, j), x(i, j)\},
\end{array}
$$

The calculations in Eq. (3.6) must be carried out for $j=1$ to $N-1$, and $i=1$ to $N-j$.

Example 3. Consider the multiplication of two matrices $A\left(N_{1} \times N_{3}\right)$ and $B\left(N_{3} \times N_{2}\right)$ to give a resulting matrix $C\left(N_{1} \times N_{2}\right)$. In a fairly straightforward way one can obtain a regular iterative algorithm that performs the desired computation:

$$
\begin{align*}
& \text { for } i=1 \text { to } N_{1}, j=1 \text { to } N_{2}, k=1 \text { to } N_{3} \\
& \qquad \begin{array}{l}
a(i, j, k)=a(i, j-1, k) \\
b(i, j, k)=b(i-1, j, k) \\
c(i, j, k)=c(i, j, k-1)+a(i, j, k-1) b(i, j, k-1) \\
\text { end } \\
\text { with } \\
\quad a(i, 0, k)=a_{i k} \quad b(0, j, k)=b_{k j} \quad c(i, j, 0)=0
\end{array}
\end{align*}
$$

The desired elements $c_{i j}$ are obtained as $c\left(i, j, N_{3}\right)$, with $c(i, j, n)$ representing the partial sum of the first $n$ terms of the summation for $c_{i j}$. Notice that the $a$ and $b$ terms have to be "propagated" through the algorithms so that they would be available at the appropriate places.

The regular iterative algorithms, as shown in the above examples, have the following features:

- They are in the single assignment format.
- Each variable in RIAs is identified by a label ( $p$ in Example 1, for instance) and an index vector ( $k=[i, j]^{T}$ in Example 1). The range of the index vector, which in general can be $S$-dimensional with $S_{21}$, forms the index space. For instance, in Example 1, the index space is twodimensional and is described by an ( $N \times N$ ) square grid (Figure 3.3a) whereas in Example 2 it is a triangular (Figure 3.3b) and in Example 3 it is a cuboid (Figure 3.3c). At each integer point in the index space, a set of $V$ labels is used to denote the distinct variables ( $V=1,2$ and 3 for Examples 1, 2 and 3 respectively).
- The main feature of these algorithms is the regularity of the direct dependencies among the variables with respect to the index points. That is, if $x(k)$ is computed using the value of $y(\boldsymbol{k}-\boldsymbol{d})$, then the index displacement vector $\boldsymbol{I}$, corresponding to this direct dependence, is the same regardless of the index point $\mathbf{k}$. In Example 1 for instance, $p(i, j)$ is directly dependent on, say, $p(i-1, j)$ irrespective of the particular value of $i$ and $j$. As a consequence of this regularity, the dependence graph of an RIA has an iterative structure, which can be clearly demonstrated by drawing the dependence graph (DG) within the index space (Figure 3.4).


Figure 3.3 The index spaces of the RIAs in Examples 1-3

Reduced Dependence Graph. The fine structure in the dependence graph of an RIA is concisely captured in the concept of a Reduced Dependence Graph (RDG) [KungHT79]. In general, the RDG of an RIA has $V$ nodes, one for each of the indexed variables in the RIA; it has a directed arc from node $x$ to node $y$, if $y(k)$ is computed using the value of $\boldsymbol{x}(\boldsymbol{k}-\boldsymbol{d})$ for some $\boldsymbol{d}$; finally, each directed arc is assigned a vector weight representing the displacement of the index point across the direct


Figure 3.4 The DGs of the RIAs in Examples 1-3 dependence. The RDGs for the RIAs in Examples 1-3 are shown in Figure 3.5. The RDG together with the specification of the index space, will combine to form a complete description of the dependence graph. Given the RDG and a specification of the index space, one has all the information necessary to determine a systolic implementation for the RIA.


Figure 3.5 The RDGs for RIAs in Examples 1-3

Algorithms that at first glance may appear not to be an RIA can sometimes be transformed into an RIA
if additional variables are included whose sole purpose is to "propagate" certain values from one part of the index space to another. For some algorithms, one may need to transform the index space, even possibly in some nonlinear way for one or more of the variables. Using such simple manipulations, many interesting problems can be solved by means of regular iterative algorithms.

There are two points with regard to RIAs that worth remark. The first is that the concept of an iteration is naturally introduced, and this concept is different from the concept of time. Computations belonging to the same iteration unit would, in general, not all occur simultaneously (and could, in fact, be displaced from one another by arbitrarily large amounts) in time. On the other hand, certain computations belonging to distinct iterations could be performed at the same time. Iteration is a property of the algorithm while time is a property of its implementation. The mapping from the iteration to time will be discussed in the next section.

The other point worth noticing is fairly obvious. The dependencies between neighbouring iteration units are different for different variables. This corresponds to the fact that index displacements have been obtained for each pair of variables separately rather than only between iteration units.

In this section, we have introduced the basic concepts of the exploitation of parallelism and regular iterative algorithms. In the next section, we will investigate how systolic architectures can systematically be derived from given RIAs.

### 3.3 A Systolic Array Architecture Synthesis Procedure

### 3.3.1 Processor Space and Iteration Space

We wish to implement an RIA defined over an $S$-dimensional hyper-cubic index space of size $N \times N \times \ldots \times N$ on a $t$-dimensional systolic array (mesh-connected array of processors). (Extension to the case of a rectangular index space is conceptually easy but notationally cumbersome.) Along each coordinate axis of this array, there are $O(N)$ processors, though the exact number required has not yet been determined. We can call this $t$-dimensional Cartesian space the processor space ( $P$ ) for an implementation of the algorithm ( $P$ also denotes an $S \times t$ matrix when we project the index points to the processor space).

Each index point can now be projected on to the processor space by means of some "many-to-one" mapping that assigns a unique processor to the iteration unit corresponding to each index point. Each processor is responsible for the execution of several iteration units. We shall restrict the projections to be linear.

Let the subspace of the index space complementary to the processor space be called the iteration space $(U)(U$ is also referred to as an $S \times u$ matrix where $t+u=S$ when we project the index points into the iteration space). Every index point can then be expressed uniquely as a linear combination of two points, one in each subspace. These points are the projections of the index point on to the respective subspaces.

Even though we started out by defining a processor space first, the choice of iteration space is more fundamental. The reason is that if two index points $\boldsymbol{k}_{\mathbf{1}}$ and $\boldsymbol{k}_{\mathbf{2}}$ are such that their vector difference lies entirely within the iteration space ( $\boldsymbol{k}_{\mathbf{1}}-\boldsymbol{k}_{\mathbf{2}}=\boldsymbol{U} \boldsymbol{a}$ for some vector $\boldsymbol{a}$ ), then the corresponding computations are mapped to the same processor. Thus the mere choice of the iteration space dictates which index points shall be computed by which processors, through the order in which these computations are performed (or a time schedule) has not yet been specified. Given the iteration space, one may pick any complementary processor space, the choice affecting only the way in which the processors are numbered. For convenience, we shall always choose the processor space to be orthogonal to the iteration space.

To visualize this process, consider a three-dimensional index space for some algorithm. Choose a vector for the (one-dimensional) iteration space. Now project the cube along the direction of the vector on to a plane that is perpendicular to it. The projected images of the index points on this plane represent the processors in the corresponding implementation. The images of several index points will overlap indicating that the associated iteration units must all be executed on the same processor.

It is easy to see that the number of image points does not change if the plane is tilted about so long as the plane is not rendered parallel to the iteration vector. Think of a three-dimensional grid of point objects suspended in space. Shine light on these object from a source at infinity so that all the light rays are parallel to the desired iteration vector. Capture the shadow of the grid points on a screen which represents the processor space. Clearly, a different shadow will form if the light shines from a different directions, but the same shadow will merely be compressed or enlarged along one axis or
the other as the screen is tilted about. In other words, given the iteration vector, any complementary processor space yields the same mapping.

Given a decomposition of the index space into the processor space and the iteration space, one can determine the necessary interconnections among the processors by projecting on to the processor space interconnections between index points that already exist in the index space. The interconnections in the index space are specified by the algorithm. If in iteration unit $\boldsymbol{k}$, the variable $\boldsymbol{x}_{i}(\boldsymbol{k})$ is computed using the value of $\boldsymbol{x}_{\boldsymbol{j}}(\boldsymbol{k}-\boldsymbol{d})$, then there is an arc from index point $\boldsymbol{k}-\boldsymbol{d}$ to index point $\boldsymbol{k}$. If these index points map on to different processors then the variable $\boldsymbol{x}_{j}(\boldsymbol{k}-\boldsymbol{d})$ must be transferred from the processor corresponding to the first variable to the processor corresponding to the second. We require that this data transfer occur directly without hopping from processor to processor. Since, for an RIA, the index displacements for the variables participating in each iteration unit are similar, their projection on to the processor space results in a regular interconnection pattern of processors and this satisfies the important property of systolic array: regularity.

We shall follow the convention of describing all vector spaces by means of integer basis vectors each with 1 as the greatest common divisor of all elements.

The partitioning of the index space into the processor space and the iteration space provides a complete description of the directions of data flow in the implementation. One must next determine a schedule for the computation performed at each processor such that the available computational resources are utilized to the fullest extent possible.

### 3.3.2 Obtaining a Schedule

The most important requirement of a schedule of computation is obvious-no computation must be scheduled before all its inputs have been computed. In another word, if variable $x_{i}(k)$ is computed using variable $x_{j}\left(\boldsymbol{k}-\boldsymbol{d}_{j i}\right)$, then the time $t\left(x_{i}(\boldsymbol{k})\right)$ at which $x_{i}(\boldsymbol{k})$ is computed and the time $t\left(x_{j}\left(\boldsymbol{k}-\boldsymbol{d}_{j i}\right)\right)$ at which $x_{j}\left(k-d_{j i}\right)$ is computed must satisfy the following relation:

$$
\begin{equation*}
t\left(x_{i}(k)\right)-t\left(x_{j}\left(k-d_{j i}\right)\right) \geq h_{j i} \tag{3.8}
\end{equation*}
$$

where $h_{j i}$ is the time required to compute $x_{i}(k)$ given $x_{j}\left(k-d_{j i}\right)$. If we let the unit time for a systolic
clock as 1 , the time $h_{j i}$ should be a positive integer, indicating the number of clock clicks required to complete the computation.

The time at which each variable is computed can be assumed as a sum of two terms, one representing the time at which this particular iteration unit begins computing, and the other representing the time at which this particular variable is computed within the iteration unit. Consider the first term first. As variables are propagated through the array, they suffer a delay at step of propagation. Correspondingly, one would expect a skew between the times at which corresponding computations are initiated in different processors, this skew being a linear function of the processor coordinates. Similarly, within a processor, one would hope that every iteration unit is similarly computed and therefore required the same amount of time. Putting these together, one can expect the time at which each iteration unit commences to be a linear function of its index point. The following discussion will based on this linear function assumption.

If a linear schedule is considered, we can let $\lambda=\left(\lambda_{1}, \lambda_{2}, \cdots, \lambda_{s}\right)^{T}$ be an $S \times 1$ constant vector and let quantity $\gamma_{j}$ be a scalar constant that is specific to each variable $x_{j}$. Then we can write

$$
\begin{equation*}
t\left(x_{j}(k)\right)=\lambda^{T} k+\gamma_{j} \tag{3.9}
\end{equation*}
$$

The above scheduling strategy is called a wiform affine schedule. By substituting Eq. (3.9) into Eq. (3.8), we have

$$
\begin{equation*}
\gamma_{i}-\gamma_{j}+\lambda^{T} d_{j i} \geq h_{j i} \tag{3.10}
\end{equation*}
$$

If we combine all such inequalities for every dependence in the algorithm, we obtain a single matrix inequality, called the Schedule Constraimt Inequality

$$
\begin{equation*}
\gamma^{T} C+\lambda^{T} D \geq h^{T} \tag{3.11}
\end{equation*}
$$

where
(a) $C$ is the familiar $V \times E$ edge-vertex incidence matrix or the comection matrix, commonly used in circuit analysis [Hayt78]. It has $E$ columns (where $E$ is the number of edges in the RDG), one for each of the edges in the RDG, and $V$ rows, one for each of the nodes in the RDG. The $(m, n)$ th element of $C, c_{m n}$, is +1 if edge $n$ terminates in node $m$, is -1 if edge $n$ originates from node $n$, and is zero otherwise (if edge $n$ both originates and terminates at node $m$, then also $c_{m n}$ is zero).
(b) $D$ is the $S \times E$ index displacement matrix, in which the $m$ th column is the vector weight, or the index displacement, on the $m$ th edge in the RDG.
(c) $\gamma$ is the $V \times 1$ vector obtained by stacking all $\gamma_{i}$ corresponding to all variables in the appropriate order, consistent with the arrangement of the rows in the connection matrix.
(d) $\boldsymbol{h}$ is the $E \times 1$ vector obtained by stacking all $h_{j i}$ corresponding to all the edges in the RDG.

Eq. (3.11) can be further expressed as

$$
\begin{equation*}
y^{T} G \geq h^{T} \quad \text { or } \quad G^{T} y \geq h \tag{3.12}
\end{equation*}
$$

where $\boldsymbol{y}^{T}=\left(\gamma_{1}, \gamma_{2}, \cdots, \gamma_{V}, \lambda_{1}, \lambda_{2}, \cdots, \lambda_{s}\right)$ and $G=\left[\begin{array}{l}C \\ D\end{array}\right]$.

Thus from RDG of the given algorithm, one can obtain the $G$ matrix (combination of the $C$ and $D$ matrices), and from the hardware that is to be used one can obtain the $\boldsymbol{h}$ vector. Given these, one has simply to find a value of $y$ that satisfy Eq. (3.12), and the problem for finding a schedule becomes simply one of obtaining feasible solution to a linear programming problem. If such a solution exists, it can be found by well-known techniques, and there are also techniques to determine whether there is a solution or not.

One characteristic of any algorithm is its critical path, or the longest sequence of operations that must succeed one another. No matter how many processors are employed to solve a problem, it takes time at least proportional to the length of the critical path. For a given RIA, with the index space being an $S$-dimensional hypercube of length of $N$, let the critical path be $O\left(N^{\circ}\right)$. If all the elements of $y$ were to be constants independent of $N$, then the schedule, being a linear function of index vector as expressed in Eq. (3.9), would be only $O(N)$. Therefore, such a constant vector $y$ can be only found for problems that have a critical path that is $O(N)$. In other words, an RIA with a uniform affine schedule must have a critical path of $O(N)$. Most interesting algorithms do indeed have such a critical path of $O(N)$. However, there are some that do not, and for these, obviously a solution cannot be found to Eq. (3.12). If no solution.exists to the schedule constraint inequality, one may still be able to obtain a valid schedule for the algorithm. The tactic is to split the iteration unit into several pieces, with no cyclic dependence between pieces, and separately obtain a schedule for each piece by solving
a smaller schedule constraint inequality. These schedule can then be knit together to obtain a global schedule. Further details on this tactic can be found in [Rao85].

### 3.3.3 Compatibility of Schedule and Iteration Space

Having found one or more schedules for the algorithm, one must choose one that results in a "good" implementation. Specially, we require that the computations to be performed well all evenly distributed so that we need as few processors as possible. Remember that if the critical path of an algorithm is $O(N)$, the schedule achieved will be at least $O(N)$.

Compatibility Theorem. If $\boldsymbol{U}$, the basis for iteration space, be of dimension $u \geq c$, and be such that

$$
\begin{equation*}
\lambda^{T} U \neq 0 \tag{3.13}
\end{equation*}
$$

then the RIA can be executed in time $O\left(N^{u}\right)$ for the schedule and iteration space chosen.

Proof: Consider the time required for computing the values of the variable $x_{j}$ alone on the resulting systolic array. From Eq. (3.9), the values of $x_{j}$ will be computed at $O(N)$ distinct steps. For any two
index points $\boldsymbol{k}_{\mathbf{1}}$ and $\boldsymbol{k}_{2}$, the values of $\boldsymbol{x}_{j}\left(\boldsymbol{k}_{\mathbf{1}}\right)$ and $\boldsymbol{x}_{j}\left(\boldsymbol{k}_{2}\right)$ will be assigned to the same time step provided

$$
\begin{equation*}
\lambda^{T}\left(k_{1}-k_{2}\right)=0 \tag{3.14}
\end{equation*}
$$

Further, $x_{j}\left(k_{1}\right)$ and $x_{j}\left(k_{2}\right)$ will be computed by the same processor if

$$
\begin{equation*}
\left(k_{1}-k_{2}\right)=U a_{j} \tag{3.15}
\end{equation*}
$$

for some constant vector $a_{j}$ of appropriate dimensions. Thus $x_{j}\left(\boldsymbol{k}_{1}\right)$ and $x_{j}\left(\boldsymbol{k}_{2}\right)$ will be assigned to the same processor and will be computed at the same time step provided

$$
\begin{equation*}
\lambda^{T} \boldsymbol{U} a_{j}=0 \tag{3.16}
\end{equation*}
$$

If $\lambda$ and $\boldsymbol{U}$ meet the requirement of Eq. (3.13), then there can at most be $O\left(N^{u-1}\right)$ difference linear independent choices for $a_{j}$ within the index space that satisfy Eq. (3.16). Therefore, for the computation of $x_{j}$, at each distinct computational step, each processor will require $O\left(N^{u-1}\right)$ time. Since there are $O(N)$ such steps, the total time required by the systolic array for computing $x_{j}$ at all
index points is $O\left(N^{4}\right)$. Using similar argument for each of other variables in turn, it can be seen that the total time required for completing the algorithm is $O\left(N^{4}\right)$ as desired.

If $c=u=1$, the critical path is $O(N)$ and the iteration space becomes a single vector. According to the compatibility theorem, the RIA can be executed in time $O(N)$. At each distinct processing step, each processor will require $O(1)$ time which is independent on the problem size $N$. If $x_{j}\left(\boldsymbol{k}_{\mathbf{1}}\right)$ and $x_{j}\left(k_{2}\right)$ are computed by same processor, i.e.,

$$
\begin{equation*}
k_{1}-k_{2}=\alpha U \tag{3.17}
\end{equation*}
$$

then we have

$$
\begin{equation*}
t\left(x_{j}\left(k_{1}\right)\right)-t\left(x_{j}\left(k_{2}\right)\right)=\lambda^{T}\left(k_{1}-k_{2}\right)=\alpha \lambda^{T} U \neq 0 \tag{3.18}
\end{equation*}
$$

Thus the compatibility theorem in the case of $u=c=1$ implies that if $x_{j}\left(\boldsymbol{k}_{1}\right)$ and $x_{j}\left(\boldsymbol{k}_{2}\right)$ are computed by the same processor, then they must not be assigned to the same step in the schedule.

When we allow the dimension of the iteration basis to be greater than 1 , i.e., $1>1$, at a distinct computation step, each processor will require $O\left(N^{\mu-1}\right)$ time which is dependent on the size of the problem. That means, there are some loop operations inside an individual PE which require to be completed within a systolic clock tick. This choice of $u$ thus results a trade-off between the number of processing elements and the complexity of the processing elements. In common situations or in a strict sense, a systolic array should use a single vector as their iteration space in order to obtain the simplest PEs possible. The choice of iteration space with $\|>1$ is actually a type of partitioning of the systolic array with $\pi=1$. Therefore, in a wide sense, the derived array with iteration space $u>1$ is also referred as to a systolic array. In a strict sense, systolic array should be the array derived by choosing $u=1$.

### 3.3.4 A Step-by-Step Synthesis Procedure

We now summarize the synthesis procedure discussed above:

Step 1. Given an RIA with a critical path of $O\left(N^{c}\right)$ (where $c=1$ ) defined over an $S$-dimensional index space, choose $t$ and $u$ such that $t+u=S$ and $u \geq c$. For such a choice of $u$, the algorithm could potentially be completed in $O\left(N^{u}\right)$ time with $O\left(N^{\prime}\right)$ processors.

Step 2. Obtain a schedule in the manner suggested in the previous subsection, and choose a $1 \mathbf{1}$ dimensional subspace of the index space to be iteration space, ensuring that the requirement of Compatibility Theorem is satisfied.

Step 3. Map the points on the index space into the processor space by means of the matrix $U$ of vectors that form the basis for the iteration space and the matrix $\boldsymbol{P}$ of vectors that form the basis for the processor space (complementary to the iteration space). For simplicity, we assume that the choice of spaces has been made such that each vector in $\boldsymbol{U}$ is orthogonal to each vector in $\boldsymbol{P}$, i.e., $\boldsymbol{U}^{\boldsymbol{T}} \boldsymbol{P}=\mathbf{0}$. The computations at index point $k$ will be performed at the processor whose location is defined by

$$
\begin{equation*}
\alpha=P^{T} k \tag{3.19}
\end{equation*}
$$

where $\alpha$ is the vector representing the coordinates of the processor in the processor space.

Step 4. Determine the interconnections between the processors. The index displacement vectors of an RIA contain full information about the flow of data from one index point in the algorithm to another. An index displacement is simply the difference between the coordinates of two index points that have a dependence between them. Therefore, an index displacement vector $d$ can be mapped into the iteration space and processor space respectively by following relations

$$
\begin{equation*}
\alpha_{d}=P^{T} d \tag{3.20}
\end{equation*}
$$

and

$$
\begin{equation*}
\beta_{d}=U^{T} d \tag{3.21}
\end{equation*}
$$

The coefficients $\alpha_{d}$ and $\beta_{d}$ obtained in Eq. (3.20) and Eq. (3.21) explicitly represent the processor and iteration components of the index displacement $\boldsymbol{d}$, and hence give a complete description of the data flow through the systolic array.

Step 5. Various properties of interest can be measured for each array obtained through the procedure above. These properties could include, for example, the throughput, the exact number of processors required, the communication requirement, the kind of initialization required, and so forth. Thus a designer could select the array best suited for his particular application.

### 3.4. Systolic Array Design for Matrix Multiplication

In this section, we shall apply the synthesis procedure presented above to the very fundamental matrix multiplication algorithm. By using the methodology, many different designs of systolic arrays for the matrix multiplication problem can be systematically obtained.

### 3.4.1 Dense Matrix Multiplication

Consider the multiplication of two matrices $A\left(N_{1} \times N_{3}\right)$ and $B\left(N_{3} \times N_{2}\right)$ to give a resulting matrix $C\left(N_{1} \times N_{2}\right)$. In a fairly straightforward way one can obtain a regular iterative algorithm that performs the desired computation:

$$
\begin{align*}
& \text { for } i=1 \text { to } N_{1}, j=1 \text { to } N_{2}, k=1 \text { to } N_{3} \\
& a(i, j, k)=a(i, j-1, k) \\
& b(i, j, k)=b(i-1, j, k) \\
& c(i, j, k)=c(i, j, k-1)+a(i, j, k-1) b(i, j, k-1) \tag{3.22}
\end{align*}
$$

end
with

$$
a(i, 0, k)=a_{i k} \quad b(0, j, k)=b_{k j} \quad c(i, j, 0)=0
$$

The desired elements $c_{i j}$ are obtained as $c\left(i, j, N_{3}\right)$, with $c(i, j, n)$ representing the partial sum of the first $n$ terms of the summation for $c_{i j}$. Figure 3.6 depicts the index space, the dependence graph (DG) and the reduced dependence graph (RDG) for the matrix multiplication problem.


Figure 3.6 Matrix multiplication: (a) Index space; (b) DG; (c) RDG

Now we use the step-by-step procedure as discussed in the previous section to design the systolic arrays for the matrix multiplication algorithm.

Step 1. As mentioned earlier, the matrix multiplication problem is of dimension of $S=3$. The algorithm has a minimum I/O latency of $O\left(N_{3}\right)$, or a critical path of $O\left(N_{3}\right)$ because each product element $c_{i j}$ is obtained by summing up $O\left(N_{3}\right)$ terms (hence $c=1$ ). We can choose $t=2$ and $u=1$. Then the choice of iteration space reduces to the choice of a single three-dimensional vector. For each choice of this iteration vector, a different geometry of implementation is obtained.

Step 2. From the RDG, we can obtain the connection matrix

$$
C=\left[\begin{array}{rrrrr}
0 & 0 & 0 & -1 & 0  \tag{3.23}\\
0 & 0 & 0 & 0 & -1 \\
0 & 0 & 0 & 1 & 1
\end{array}\right]
$$

where the first three columns capture the connections of each $a, b, c$, upon themselves, and the last two columns show the two edges, one from the node $a$ to $c$, the other from $b$ to $c$. The dependence matrix corresponding to this edge ordering can be given by

$$
D=\left[\begin{array}{lllll}
0 & 1 & 0 & 0 & 0  \tag{3.24}\\
1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 1
\end{array}\right]
$$

If we assume that it takes exactly one time unit to evaluate any of three equations in the iteration unit, then we have

$$
h=\left[\begin{array}{lllll}
1 & 1 & 1 & 1 & 1 \tag{3.25}
\end{array}\right]^{T}
$$

Then we can formulate the schedule constraint inequality as follows

$$
\left[\begin{array}{llllll}
\gamma_{a} & \gamma_{b} & \gamma_{c} & \lambda_{i} & \lambda_{j} & \lambda_{k}
\end{array}\right]\left[\begin{array}{ccccc}
0 & 0 & 0 & -1 & 0  \tag{3.26}\\
0 & 0 & 0 & 0 & -1 \\
0 & 0 & 0 & 1 & 1 \\
0 & 1 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 1
\end{array}\right] \geq\left[\begin{array}{lllll}
1 & 1 & 1 & 1 & 1
\end{array}\right]
$$

Since the critical path is of length $O\left(N_{3}\right)$, the $\gamma$ quantities have been assumed to be constants independent of the index.

A solution to this system of inequalities is given by $\gamma_{a}=\gamma_{b}=\gamma_{c}=0$ and $\lambda_{i}=\lambda_{j}=\lambda_{k}=1$. This leads to $t(a(k))=t(b(k))=t(c(k))=i+j+k$. In other words, all variables associated with the index point $k=[i j k]^{T}$ complete evaluation at time $i+j+k$. Therefore, any iteration vector not orthogonal to $\left[\begin{array}{lll}1 & 1 & 1\end{array}\right]^{T}$ will satisfy the requirement of the compatibility theorem.

We can choose the iteration space to be the vector $U=\left[\begin{array}{lll}0 & 0 & 1\end{array}\right]^{T}$ and it is obvious that the iteration vector satisfies the condition of the compatibility theorem. The processor space which is complementary to the processor space can then be given by

$$
P=\left[\begin{array}{ll}
1 & 0  \tag{3.27}\\
0 & 1 \\
0 & 0
\end{array}\right]
$$

Step 3. The index points $k=[i j k]^{T}$ in the index space can be mapped into the processor space as follows

$$
\alpha=\boldsymbol{P}^{T} k=\left[\begin{array}{lll}
1 & 0 & 0  \tag{3.28}\\
0 & 1 & 0
\end{array}\right]\left[\begin{array}{l}
i \\
j \\
k
\end{array}\right]=\left[\begin{array}{l}
i \\
i \\
j
\end{array}\right]
$$

This means that the computation at index points $k=[i j k]^{T}$ will be performed at the processor located at $\alpha=\left[\begin{array}{l}i j\end{array}\right]^{T}$. Hence a $N_{1} \times N_{2}$ rectangular array of processing elements is formed by this projection.

Step 4. The interconnection of the array can be obtained by mapping the index displacement vector into the processor space. For the matrix multiplication algorithm above, the index displacements are all unit vectors along the coordinate axes of the index space, that is, $d_{a a}=\left[\begin{array}{lll}0 & 1 & 0\end{array}\right]^{T}, d_{b b}=\left[\begin{array}{lll}1 & 0 & 0\end{array}\right]^{T}$ and $\boldsymbol{d}_{a c}=\boldsymbol{d}_{b c}=\boldsymbol{d}_{c c}=\left[\begin{array}{lll}0 & 0 & 1\end{array}\right]^{T}$. For the iteration space and processor space we have chosen, the index displacement vectors can be mapped into the processor space and iteration space by Eqs. (3.20) and (3.21). Mapping the index displacements into the processor space results $\alpha_{a a}=\left[\begin{array}{ll}0 & 1\end{array}\right]^{T}, \alpha_{b b}=\left[\begin{array}{ll}1 & 0\end{array}\right]^{T}$ and $\alpha_{a c}=\alpha_{b c}=\alpha_{c c}=\left[\begin{array}{ll}0 & 0\end{array}\right]^{T}$ and mapping the index displacements into the iteration space results $\beta_{a a}=\beta_{b b}=0$ and $\beta_{a c}=\beta_{b c}=\beta_{c c}=1$. Thus $a$ and $b$, respectively, are transmitted vertically and horizontally in the processor array, while $c$ is transmitted along the iteration axis within the same processor. The
resulting matrix multiplication systolic array is the well-known systolic array first presented by H.T. Kung in [KungHT78] (see Figure 3.7).

Similarly through picking different choices of the iteration space, we can obtain the different types of systolic arrays. It is clear that there are infinite number of different choices for the iteration space. Each of this will correspond to a different array. In practical, however, we wish to limit the arrays generated to have not just local but nearest communication.

Table 3.1 Summary of different systolic arrays for matrix multiplication

|  | Iteration Space $\boldsymbol{U}^{T}$ | Processor Space $\boldsymbol{P}^{T}$ | Number of Processors $\left\|N_{1}=N_{2}=N_{3}=N\right\|$ | Schedule $t(a(k))=t(b(k))=t(c(k))$ |
| :---: | :---: | :---: | :---: | :---: |
| 1 | $\left[\begin{array}{llll}0 & 0 & 1\end{array}\right],\left[\begin{array}{llll}0 & 0 & -1\end{array}\right]$ | [100;010] | $N_{1} \times N_{2} \quad\left[N^{2}\right\}$ | $i+j+k$ |
| 2 | [0110], [0-1 0] | [100;001] | $N_{1} \times N_{3} \quad\left[N^{2}\right]$ |  |
| 3 | $\left[\begin{array}{llll}1 & 0 & 0\end{array}\right],\left[\begin{array}{llll}-1 & 0 & 0\end{array}\right]$ |  | $N_{2} \times N_{3} \quad\left[N^{2}\right]$ |  |
| 4 | [ 01111$],\left[\begin{array}{llll}0 & -1 & -1\end{array}\right]$ | [100;0.11] | $\begin{aligned} & N_{1}\left(N_{2}+N_{3}-1\right) \\ & {\left[2 N^{2}-N\right]} \end{aligned}$ | $i+j+k$ |
| 5 | $\left[\begin{array}{llll}0 & 1 & -1\end{array}\right],\left[\begin{array}{llll}0 & -1 & 1\end{array}\right]$ | [100;011] |  | $i+2 j+k$ |
| 6 | $\left[\begin{array}{llll}1 & 1\end{array}\right],\left[\begin{array}{llll}-1 & 0 & -1\end{array}\right]$ | $1010,10-1 \mid$ | $\begin{aligned} & N_{2}\left(N_{1}+N_{3}-1\right) \\ & \left\|2 N^{2}-N\right\| \end{aligned}$ | $i+j+k$ |
| 7 | [10-1], [-1 001$]$ | [010; 1011$]$ |  | $2 i+j+k$ |
| 8 | $\left[\begin{array}{llll}1 & 1 & 0\end{array}\right],\left[\begin{array}{ccc}-1 & -1 & 0\end{array}\right]$ | [001; 1-10] | $\begin{aligned} & N_{3}\left(N_{1}+N_{2}-1\right) \\ & {\left[2 N^{2}-N\right]} \end{aligned}$ | $i+j+k$ |
| 9 | $\left[\begin{array}{lll}1 & -1 & 0\end{array}\right],\left[\begin{array}{lll}-1 & 1 & 0\end{array}\right]$ | [001; 1110] |  | $2 i+j+k$ |
| 10 | $\left[\begin{array}{llll}1 & 1\end{array}\right],\left[\begin{array}{lll}-1 & -1\end{array}\right]$ | $[01-1 ; 2-1-1]$ | $\begin{gathered} N_{1} N_{2}+N_{2} N_{3}+N_{3} N_{1} \\ -N_{1}-N_{2}-N_{3}+1 \\ \\ \left\{3 N^{2}-3 N+1\right] \end{gathered}$ | $i+j+k$ |
| 11 | $\left[\begin{array}{llll}1 & -1],\left[\begin{array}{llll}-1 & -1 & 1\end{array}\right]\end{array}\right.$ | [011:2-11] |  |  |
| 12 | [1-1 1], [-1 1-1] | $[011 ; 21-1]$ |  |  |
| 13 | $[-111] ;\left[\begin{array}{lll}1 & -1 & -1\end{array}\right]$ | [01-1;211] |  |  |
| 14 | [2111]; $[-2-1-1]$ | [01-1; - 1 11] | $\begin{aligned} & 2 N_{2} N_{3}+\left(N_{1}-2\right)\left(N_{2}+N_{3}-1\right) \\ & {\left[4 N^{2}-5 N+2\right]} \end{aligned}$ | $i+j+k$ |
| 15 | [2 1-1]; $\left[\begin{array}{llll}-2 & -1 & 1\end{array}\right]$ | $1011 ; 1-111$ |  |  |
| 16 | $\left[\begin{array}{lllll}0 & 1 & 1\end{array}\right],\left[\begin{array}{llll}0 & -1 & -1\end{array}\right]$ | $[100 ; 0-11]$ | $w^{\prime} N_{1}$ | $i+j+k$ |
| 17 | $\left[\begin{array}{llll}0 & 1 & -1\end{array}\right],\left[\begin{array}{llll}0 & -1 & 1\end{array}\right]$ | [100;011] |  | $i+2 j+k$ |
| 18 | $\left[\begin{array}{llll}1 & 1 & 1\end{array}\right],\left[\begin{array}{lll}-1 & -1 & -1\end{array}\right]$ | \{01-1;2-1-1] | $w_{1}^{\prime} w_{2}^{\prime}$ | $i+j+k$ |
| 19 | $[1 \mid-1],[-1-11]$ | [011:2-11] |  |  |

If we restrict the iteration vector to be $U=\left[\begin{array}{lll}u_{1} & u_{2} & u_{3}\end{array}\right]^{T}$ where $u_{i} \in\{-1,0,1\}$ for $i=1,2,3$, that is, the element of the iteration vector is either one of the three values $-1,0$ and 1 . This constraint can meet the nearest communication requirement. It is then a simple combinatorial exercise to show that there are 26 different choices of an iteration vector (noticing that zero length vector is not valid to be an iteration vector). Because two iteration vectors pointing to the opposite directions generate the same systolic array, we can obtain 13 different systolic arrays for the matrix multiplication algorithm.

In addition to the above choices of iteration space, there are still other candidates that meet the nearest neighbour communication requirements. The choices of $\boldsymbol{U}=\left[\begin{array}{lll}2 & 1 & 1\end{array}\right]^{T}$ and $U=\left[\begin{array}{lll}2 & 1 & -1\end{array}\right]^{T}$ are two examples which also satisfy the requirement of nearest neighbour communication. Actually any iteration vector with one element being $\pm 2$ and the other two elements being $\pm 1$ will result a systolic array of nearest communication. This leads to 24 choices of iteration vector, or 12 systolic arrays, under the nearest communication constraint, although only 2 systolic arrays are listed in Table 3.1.

Table 3.1 summarizes the obtained 15 different systolic arrays as well as another 4 arrays for band matrix multiplication to be discussed later. It lists the iteration space, processor space in the 2 nd and 3 rd columns respectively, where the processor space matrix $P$ is written in a compact format with the semicolon separating the different rows of the matrix (a convention used in the computer language MATLAB). The number of processor used by each systolic array design is provided in the 4 th column, and the numbers inside the brackets are for the case of $N_{1}=N_{2}=N_{3}=N$. The feasible schedule for each design is given in the last column in the table.

Notice that in Table 3.1 some designs have different schedules from $t(a(k))=t(b(k))=t(c(k))=i+j+k$ in order to satisfy the compatibility theorem. In this case of the matrix multiplication algorithm, a schedule with $\lambda$ being not orthogonal to the iteration vector $U$ will be a feasible schedule according to the compatibility theorem. For example, the systolic array design of version 9 has a schedule $t(a(k))=t(b(k))=t(c(k))=2 i+j+k$ where $\lambda=\left[\begin{array}{ll}2 & 1 \\ 1\end{array}\right]^{T}$ is not orthogonal to the iteration vector $\boldsymbol{U}=\left[\begin{array}{lll}1 & -1 & 0\end{array}\right]^{T}$.

Step 5. Figures 3.7-3.21 (as given at the end of this chapter) show the different versions of systolic array designs for the matrix multiplication algorithm. Version 1 in Figure 3.7 is the well-known array first presented by H.T. Kung in [KungHT78]. In this array, the $c_{i j}$ values are stationary while the $a_{i j}$ move from bottom to top and the $b_{i j}$ move from left to right. At the end of the computation, the $c_{i j}$
values are held in the array's cells and therefore some mechanism should be adopted to drive the result out of the array [Quinton91].

There are another two types of orthogonal systolic arrays. In Version 2 systolic array (in Figure 3.8), the values of matrix $B$ move from left to right and the values of $C$ move form bottom to top in a pipeline way. The values of matrix $A$ are required to be preloaded in the array cell. Version 3 (in Figure 3.9) is similar to Version 2, with $A$ and $C$ moving and $B$ stationary.

Version 4 (in Figure 3.10), Version 6 (in Figure 3.12) and Version 8 (in Figure 3.14) are rectangular arrays. These arrays feature that all the input data and output data are circulating. One matrix moves horizontally from left to right, while the other two matrices move vertically in opposite directions. The circulation of the input and output data is an obvious advantage because they allow the extraction of results in an immediate way. The cost for this advantage is that more PEs are required to complete the same computations and less efficiency can be achieved. This will be investigated in more details in the next section.

Version 5 (in Figure 3.11), Version 7 (in Figure 3.13) and Version 9 (in Figure 3.15) are another group of rectangular arrays. These arrays also feature that all the input data and output data are circulating. One matrix moves horizontally from left to right, while the other two matrices move vertically in the same direction from bottom to top but in different speed. Notice that the small black dots on the arcs of these arrays represent a unit time delay. The schedule for these arrays are also different from the other versions of systolic arrays as listed in Table 3.1.

Version 10 (in Figure 3.16) is a hexagonal systolic array. In this systolic array, the three matrices $\boldsymbol{A}$, $B$ and $C$ are input to the array diagonal by diagonal in three different directions. Versions 11-13 (in Figures 3.17-3.19 respectively) also take a hexagonal geometry. In these three systolic arrays, one of the three matrices $\boldsymbol{A}, \boldsymbol{B}$ and $C$ moves through the arrays diagonal by diagonal, while the other two matrices move through the arrays anti-diagonal by anti-diagonal.

Versions 14 and 15 (in Figures 3.20 and 3.21 respectively) can be also regarded as hexagonal arrays although they have different geometrical form from Versions 10-13. The data movement in these arrays is more complicated. These arrays also have a larger number of PEs than the rest of the systolic arrays in Table 3.1 when $N_{1}=N_{2}=N_{3}=N$.

More details on the performance analysis of these array will be treated in the next section before we discuss the band matrix multiplication.

### 3.4.2 Band Matrix Multiplication

As discussed above, the number of PEs in the systolic arrays for dense matrix multiplication depends on the dimension of the input matrices $\boldsymbol{A}$ and $\boldsymbol{B}$. When the matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ are banded, especially when the bandwidth is far less than the matrix dimension, it is not efficient for the systolic array discussed above to perform the banded matrix multiplication. In this sub-section, we will design several band matrix multiplication systolic arrays which can be directly derived from some of the dense matrix multiplication array discussed earlier.

We say that a matrix $A \in \mathbb{R}^{m \times n}$ is a band matrix if $a_{i j}=0$ whenever $i ン j+p$ and $j ン i+q$, where $p$ is called lower bancwidth and $q$ is called upper bandwidth. The total bandwidth of such a matrix is $w=p+q+1$. Below is an example of a band matrix with $p=2$ and $q=3$ :

$$
\boldsymbol{A}=\left[\begin{array}{llllllll}
\times & \times & \times & \times & & & &  \tag{3.29}\\
\times & & & \\
\times & \times & \times & \times & \times & & & \\
\times & \times & \times & \times & \times & \times & & \\
& & \\
& \times & \times & \times & \times & \times & \times & \\
& & \\
& & \times & \times & \times & \times & \times & \times \\
& & \\
& & & \times & \times & \times & \times & \times \\
& \times & \\
& & & & \times & \times & \times & \times \\
& \times & \times \\
& & & & & & \times & \times \\
\times & \times & \times & \times \\
& & & & & & & \times \\
\times & \times & \times \\
& & & \times & \times
\end{array}\right]
$$

where $\times$ 's designates arbitrary nonzero entries.

Version 16. We first consider the case where $A$ is a dense matrix and $B$ is a band matrix (similar designs can be obtained when $A$ is a band matrix and $B$ is a dense matrix). Observing Version 4 systolic array discussed earlier, we find that the matrix $B$ enters into the systolic array in a diagonal by diagonal fashion. Therefore it is straightforward for us to obtain a systolic array with less PEs for the case of $B$ being a band matrix by removing the rows of the PEs corresponding to those diagonals with zero elements. Figure 3.22 shows the obtained systolic array where $N_{1}=N_{2}=N_{3}=N=4$ and $p=q=1$.

Version 17. When we examine Version 5 systolic array, we can find that the matrix $B$ enters the systolic array in an anti-diagonal by anti-diagonal fashion. Therefore it is not straightforward to use the procedure as used in Version 16 systolic array to obtain a new array for the band matrix case. However, we know that matrix multiplication can be expressed as

$$
C=\boldsymbol{A B}=\left[\begin{array}{cccc}
a_{11} & a_{12} & \cdots & a_{1 N}  \tag{3.30}\\
a_{21} & a_{22} & \cdots & a_{2 N} \\
\vdots & \vdots & \ddots & \vdots \\
a_{N 1} & a_{N 2} & \cdots & a_{N N}
\end{array}\right]\left[\begin{array}{cccc}
b_{11} & b_{12} & \cdots & b_{1 N} \\
b_{21} & b_{22} & \cdots & b_{2 N} \\
\vdots & \vdots & \ddots & \vdots \\
b_{N 1} & b_{N 2} & \cdots & b_{N N}
\end{array}\right]=\left[\begin{array}{cccc}
a_{1 N} & \cdots & a_{12} & a_{11} \\
a_{2 N} & \cdots & a_{22} & a_{21} \\
\vdots & \vdots & \vdots & \vdots \\
a_{N 1} & a_{N 2} & \cdots & a_{N N}
\end{array}\right]\left[\begin{array}{cccc}
b_{N 1} & b_{N 2} & \cdots & b_{N N} \\
\vdots & \vdots & \vdots & \vdots \\
b_{21} & b_{22} & \cdots & b_{2 N} \\
b_{11} & b_{12} & \cdots & b_{1 N}
\end{array}\right]
$$

Eq. (3.30) shows that reordering the columns of $A$ while in the same time reordering the rows of $B$ correspondingly will result the same matrix product. Notice that the elements of matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ in the right-hand-side of Eq. (3.30) are arranged in anti-diagonal way. Thus we can reorder the inputs of Version 5 systolic array and then remove the PEs in the rows corresponding to those zero diagonals of $\boldsymbol{B}$. The resulting systolic array is shown in Figure 3.23 and we will show in the next section that this version has a higher efficiency than Version 16.

Version.18. Now we consider the situation when both $A$ and $B$ are band matrices. Assume that $A$ and $B$ have bandwidths of $w_{1}$ and $w_{2}$ respectively. Version 10 systolic array can be used in this case because both $A$ and $B$ flow into the array in a diagonal by diagonal fashion. The obtained systolic array for band matrix multiplication after removing those redundant PEs in Version 10 systolic array is shown in Figure 3.24. This array was first proposed by Kung and Leiserson [KungHT79] [Leiserson81].

Version 19. By applying the technique as shown in Eq. (3.30), we can derive another systolic array for the matrix multiplication of two band matrices from Version 11 systolic array. The resulting systolic array is shown in Figure 3.25. This array was first proposed by Weiser and Davis [Weiser81]. We will show in the next section that this array performs better than Version 18 systolic array.

In this section we have together designed 19 different types of systolic arrays for the matrix multiplication algorithm. The first 15 systolic arrays are for the dense matrix multiplication; the 16 th and 17th are for the multiplication of a dense matrix and a band matrix; the last 2 designs are for the multiplication of two band matrices. Using the design procedure, we can also design one-dimensional
array by selecting a 2 -dimensional iteration space. More details on the one dimensional systolic array design for matrix multiplication are referred to [Jagadish87]

### 3.5 Systolic Array Performance Analysis

### 3.5.1 The Definition of Systolic Array Performance Measures

There are different performance measures for a systolic array. The most basic and widely used performance measures for a systolic array are listed in Table 3.2.

Array size ( $A$ ). The number of processing elements (PEs) in the array. The array size obviously determines the basic hardware cost for implementing the systolic array. The array size can be determined when projecting the

Table 3.2 Systolic array performance measures index points to processor space.

| $A$ | Array size |
| :--- | :--- |
| $T$ | Computation time |
| $\alpha$ | Pipelining period |
| $\beta$ | Block pipelining period |
| $S$ | Speedup |
| $E$ | Efficiency of the array |
| $I O$ | I/O Channels |
| $A T^{2}$ | Area-time complexity |

Computation time (T). The computation time for a given algorithm can be divided into two terms. The first term is the time interval between starting the first computation and finishing the last computation of a problem instance by the systolic array. This time can be expressed as

$$
\begin{equation*}
T_{c}=\max _{k_{1}, k_{2} \in l, x_{i}, x_{j} \in X} t\left(x_{j}\left(k_{1}\right)\right)-t\left(x_{i}\left(k_{2}\right)\right)_{1}+1 \tag{3.31}
\end{equation*}
$$

The second term, denoted by $T_{d}$, is the sum of the time for the input data required by the first computation to propagate to the appropriate PE from certain boundaries of the systolic array, and the time for the result generated by the last computation to propagate to the boundary of the systolic array.

Pipelining period ( $\alpha$ ). The time interval between two successive computations in a PE. In other words, the processor is busy for one out of $\alpha$ time intervals. Given a systolic schedule

$$
\begin{equation*}
t\left(x_{j}(k)\right)=\lambda^{T} k+\gamma_{j} \tag{3.32}
\end{equation*}
$$

and a projection vector $\boldsymbol{u}$, the pipelining period can be calculated as

$$
\begin{equation*}
\alpha=\lambda^{T_{u}} \tag{3.33}
\end{equation*}
$$

To verify Eq.(3.33), we note that the pipelining period equals the time between two consecutive computations for a processor [Rao85] [KungSY88b]. If the iteration vector (or projection vector) is $\boldsymbol{u}$ (with 1 as the greatest common divisor of its element) and $\boldsymbol{k}$ is a point in the index space, then $\boldsymbol{k}$ and $\boldsymbol{k}+\boldsymbol{u}$ represent the indices of two consecutive nodes that are projected to the same processor. Therefore, the pipelining period $\alpha$ is equal to the separation between the computation times of these two nodes, i.e.,

$$
\begin{equation*}
\alpha=t\left(x_{j}(k+u)-x_{j}(k)\right)=\lambda^{T}(k+u)-\lambda^{T} k=\lambda^{T} u \tag{3.34}
\end{equation*}
$$

Block pipelining period ( $\beta$ ). The time interval between the initiations of two successive problem instances by the processor array. The block pipelining period is actually the largest time span of any PE in the array, where the time span indicates the difference between the last time step and the first step a PE is busy for the given algorithm.

Speedup (S). When a given problem is processed by a systolic array, the speedup of the systolic array can be defined as the ratio of its processing time to that of a single processor, i.e.

$$
\begin{equation*}
S=T / T \tag{3.35}
\end{equation*}
$$

where $T_{s}$ is the time on a single processor for a given algorithm. When $M$ instances of problems are fed into the systolic array to be processed in a pipeline way, the total time required for a single process is $M T_{s}$ and the total time required for the systolic array is $T+(M-1) \beta$. Thus the speedup of the systolic array becomes

$$
\begin{equation*}
S_{M}=\frac{M T_{s}}{T+(M-1) \beta}, \quad S_{\infty}=\lim _{M-\infty} S_{M}=\frac{T_{s}}{\beta} \tag{3.36}
\end{equation*}
$$

Efficiency ( $E$ ). The efficiency of the array is defined as the ratio of its speedup to the number of PEs in the array

$$
\begin{equation*}
E=\frac{S}{A}=\frac{T_{s}}{T \times A} \tag{3.37}
\end{equation*}
$$

The efficiency for multiple problem instances can be given by

$$
\begin{equation*}
E_{M}=\frac{S_{M}}{A}=\frac{M T_{s}}{A[T+(M-1) \beta]}, \quad E_{\infty}=\frac{S_{\infty}}{A}=\frac{T_{s}}{A \beta} \tag{3.38}
\end{equation*}
$$

I/O channels (IO). The number of I/O lines to communicate with the outside world (the host computer or other devices) is another important performance measure. Input/Output channels are directly tied to hardware cost in terms of I/O pins of a VLSI chip or I/O wires of a circuit board. The number of I/O channels can be derived from the projected systolic array directly.

Area-time Complexity $\left(A 7^{2}\right)$. This is a combination of two factors $A$ and $T$. There are different kinds of area-time complexity measures (see Sec. 2.7.2 for more details). The $A T^{2}$ is the most useful areatime complexity measure for array processors. It provides a useful measure for the hardware costeffectiveness. In the case of many problem instances to be processed, the average time for a single problem instance is approaching to the block pipelining period $\beta$. Thus, the area-time complexity measure becomes $A \beta^{2}$.

It is worth pointing out that some of the measures introduced above may be of special interest for some cases, while other measures may be of interest for other cases. For example, when we consider a single problem with finite input data, the computation time $T$ is perhaps a more important criteria than the pipelining period. When we consider a single problem instance with indefinite input data, such as filtering in DSP applications, the pipelining period $\alpha$, instead of the computation time $T$, may become the focal point of the array performance. When many problem instances are to be processed by the same systolic array, we should be more concerned with the block pipelining period $\beta$.

### 3.5.2 Performance Analysis of the Matrix Multiplication Systolic Arrays

We consider Version I systolic array. The array has $A=N_{1} \times N_{2}$ PEs. The computation time can be obtained by

$$
T_{c}=t\left(c\left(k_{1}\right)\right)-t\left(c\left(k_{2}\right)\right)+1=N_{1}+N_{2}+N_{3}-2 \quad \text { where } k_{1}=\left[\begin{array}{lll}
N_{1} & N_{2} & N_{3}
\end{array}\right]^{T} \quad k_{2}=\left[\begin{array}{lll}
1 & 1 & 1 \tag{3.39}
\end{array}\right]^{T}
$$

The time for unloading the result from the systolic array is at least

$$
\begin{equation*}
T_{d}=\min \left\{N_{1}, N_{2}\right\} \tag{3.40}
\end{equation*}
$$

The pipelining rate $\alpha$ can be calculated by plugging $\lambda=\left[\begin{array}{lll}1 & 1 & 1\end{array}\right]^{T}$ and $u=\left[\begin{array}{lll}0 & 0 & 1\end{array}\right]^{T}$ into Eq. (3.34)

$$
\alpha=\left[\begin{array}{lll}
1 & 1 & 1
\end{array}\right]^{T}\left[\begin{array}{lll}
0 & 0 & 1 \tag{3.41}
\end{array}\right]=1
$$

The block pipelining period $\beta$ is $N_{3}$ and it is the time period one row of $A$ or one column of $B$ passes through a corresponding PE in the array.

Because a single processor requires $T_{s}=N_{1} \times N_{2} \times N_{3}$ time for the matrix multiplication algorithm, the speedup (if we ignore the time $T_{d}$ ) is

$$
\begin{equation*}
S=\frac{N_{1} N_{2} N_{3}}{N_{1}+N_{2}+N_{3}-2}, \quad S_{m}=\frac{N_{1} N_{2} N_{3}}{N_{3}}=N_{1} N_{2} \tag{3.42}
\end{equation*}
$$

The efficiency of the systolic array is given by

$$
\begin{equation*}
E=\frac{N_{3}}{N_{1}+N_{2}+N_{3}}, \quad E_{\infty}=1 \tag{3.43}
\end{equation*}
$$

The array has $N_{1}+N_{2}$ input channels and $N_{1}+N_{2}$ output channel. The area-time complexity becomes

$$
\begin{equation*}
A T^{2}=N_{1} N_{2}\left(N_{1}+N_{2}+N_{3}-2\right)^{2}, \quad A \beta^{2}=N_{1} N_{2} N_{3}^{2} \tag{3.44}
\end{equation*}
$$

Similarly, we can obtain the performance measures of all the systolic arrays for the matrix multiplication algorithm. Table 3.3 summarizes the obtained results for the four main measures $A, T$, $\alpha$ and $\beta$. Table 3.4 presents more performance measures of the obtained systolic arrays with $N_{1}=N_{2}=N_{3}=N$ for the convenience of comparison. With these performance measures, one can find out the best choice for his specific application. For instance, if one wish to achieve the best efficiency for the case of multiple problem instances, the choices will be Versions 1,2 and 3 for dense matrix multiplication. From Table 3.4, it is clear that Version 17 is superior to Version 16 in most
performance measures, and that Version 19 is better than or at least same as Version 18 in all the performance measures listed in the table.

### 3.6 Definition of Systolic Array

In Sec 3.2, we introduced the definition of a regular iterative algorithm (RIA). Using the RIA definition as a vehicle, now we turn to a formal definition of a systolic array [Ra088]. The following formal definition is consistent with the definition given in Sec. 2.1.

Definition of a Systolic Array: A systolic array (in a strict sense with $u=1$ ) is characterized by the sets $\left\{P, \tau, X, D_{p}, F\right\}$ where
$P$ is the processor space which is the set of all lattice points enclosed within a specified region in $t$ dimensional Euclidean space;
$\tau$ represents the beats of the systolic clock;
$\boldsymbol{X}$ is the set-of $V$ variables that is computed by every processor in the processor space and at every beat of the systolic clock during the execution of the array;
$\boldsymbol{D}_{\boldsymbol{p}}$ is the set of processor displacements that defines the interconnection links in the processor array so that
if $d$ is a member of $D_{p}$, then there is an interconnection link from the processor at location $p$ to the processor at location $(p+d)$ irrespective of the particular value of $p$,
if variable $x$ computed at beat $\tau$ by the processor at location $p$ is transferred across the link to the processor at location $(p+d)$, then this data transfer occurs regardless of the particular values of $\tau$ and $p$;
$F$ is the set of functional dependencies that relate the computation of a variable $x$ at processor $p$ during beat $\tau$, as a function of the variables computed during the previous beat at the
neighbouring processors. (As mentioned in Sec. 3.2, the function dependencies in $\boldsymbol{F}$ can involve conditional branches.)

With these formalizations in place, any algorithm executed by a systolic array can be characterized as follows.

Theorem. A systolic array executes a Regula Iterative Algorithm which has a uniform affine schedule. Conversely, every Regular Iterative Algorithm with a uniform affine schedule can be implemented on a systolic array.

Proof: To show that a systolic array executes an RIA, we define the index space to be

$$
I=\left\{k=\left[\begin{array}{l}
p  \tag{3.45}\\
\tau
\end{array}\right], p \in P, \tau=\text { systolic beat }\right\}
$$

Next, let the variable $x$ computed by the processor at location $p$ at beat $\tau$ be denoted as $x(\boldsymbol{k})$. Then, by the definition of systolic array, if $x(\boldsymbol{k})$ is computed using $y(l)$, then

$$
k-l=\left[\begin{array}{l}
d  \tag{3.46}\\
1
\end{array}\right]
$$

which is independent of $k$ and the extent of the index space.

To prove the converse statement, let $\{\gamma, \lambda\}$ constitute the parameters of a uniform affine schedule for the RIA. Then, if $C$ is the connection matrix of the RDG of the RIA and $D$ is its index displacement matrix, one must have

$$
\gamma^{T} C+\lambda^{T} D \geq\left[\begin{array}{llll}
1 & 1 & \cdots & 1 \tag{3.47}
\end{array}\right]
$$

Therefore, it must be possible to determine $\lambda$ such that the greatest common divisor of its elements is 1 . This implies that there exists a vector $\boldsymbol{u}$ such that

$$
\begin{equation*}
\lambda^{T} u=1 \tag{3.48}
\end{equation*}
$$

Next we redefine each indexed variable $x$ to be

$$
\begin{equation*}
\bar{x}(k)=x\left(k-\gamma_{x} u\right) \tag{3.49}
\end{equation*}
$$

Then the displacement matrix in the new variable domain can be written as

$$
\begin{equation*}
\bar{D}=D-u \gamma^{T} C \tag{3.50}
\end{equation*}
$$

Choose $\boldsymbol{u}$ to be the iteration vector and define the processor space according to

$$
\begin{equation*}
P=\{p: p=P k\} \tag{3.51}
\end{equation*}
$$

where $\boldsymbol{P}$ is orthogonal to the iteration vector. To complete the systolic array implementation, we define

$$
\begin{equation*}
\tau=t(\bar{x}(k))=t\left(x\left(k-\gamma_{x} u\right)\right)=\lambda^{T}\left(k-\gamma_{x} u\right)+\gamma_{x}=\lambda^{T} k \tag{3.52}
\end{equation*}
$$

so that $\bar{x}(k)$ is computed by the processor at location $P k$ during the $\tau$ th beat of the systolic clock.

### 3.7 Summary

Given a regular iterative algorithm, we can systematically generate a variety of systolic array designs according to the procedures we have described. These systolic designs can be compared in order to obtain one that is best under the appropriate objective functions or performance measures. As an example to show the power of the RIA design methodology, we presented 19 different systolic designs for the very fundamental and important matrix-matrix multiplication algorithm. The performance of all these designs have been given in details in order to select the best design for a specific application.

The design methodology using RIA representations has the following benefits:

- It can be fully automated as the procedure of design is quantitively described.
- From an algorithm it produces multiple choices of designs. Then we can obtain the performance measures such as the computation time, block pipeling rate, pipeling rate for each of these design. Thus we can choose the best one under specific requirements.
- It allows all regular iterative algorithm. with a uniform affine schedule to be mapped into a systolic implementation.
- It can handle a large class of algorithms that arise in a broad range of engineering problems.

Table 3.3 Summary of different systolic arrays for matrix multiplication

|  | A | $T$ |  | $\alpha$ | $\beta$ |
| :---: | :---: | :---: | :---: | :---: | :---: |
|  |  | $T_{\text {c }}$ | $T_{d}$ |  |  |
| 1 | $N_{1} \times N_{2}$ | $N_{1}+N_{2}+N_{3}-2$ | $\min \left\{N_{0}, N_{2}\right\}-1$ | 1 | $N_{3}$ |
| 2 | $N_{1} \wedge N_{3}$ |  | $\min \left\{N_{1}, N_{3}\right\}-1$ | 1 | $N_{2}$ |
| 3 | $N_{2} \times{ }^{\text {N }}$ |  | $\min \left\{N_{2}, N_{3}\right\}-1$ | 1 | $N_{1}$ |
| 4 | $N_{1}\left(N_{2}+N_{3}-1\right)$ |  | max $\left\{N_{2}, N_{3}\right\}+N_{3}-2$ | 2 | $2 \times \max \left\{N_{2} N_{3}\right\}-1$ |
| 5 |  | $N_{1}+2 N_{2}+N_{3}-3$ | $N_{3}-1$ | 1 | $\max \left\{\mathrm{N}_{2} \mathrm{~N}_{3}\right\}$ |
| 6 | $N_{2}\left(N_{1}+N_{3}-1\right)$ | $N_{1}+N_{2}+N_{3}-2$ | $\max \left\{N_{1}, N_{3}\right\}+N_{3}-2$ | 2 | $2 \cdot \max \left\{N_{1}, N_{3}\right\}-1$ |
| 7 |  | $2 N_{1}+N_{2}+N_{3}-3$ | $N_{3}-1$ | 1 | $\max \left\{N_{1}, N_{3}\right\}$ |
| 8 | $N_{3}\left(N_{1}+N_{2}-1\right)$ | $N_{1}+N_{2}+N_{3}-2$ | $\boldsymbol{m a x}\left\{N_{1}, N_{2}\right\}-1$ | 2 | $2 \times \max \left\{N_{1}, N_{2}\right\}-1$ |
| 9 |  | $2 N_{1}+N_{2}+N_{3}-3$ | $N_{2}-1$ | 1 | $\max \left\{N_{1} N_{2}\right\}$ |
| 10 | $\begin{gathered} N_{1} N_{2}+N_{2} N_{3}+N_{3} N_{1} \\ -N_{1}-N_{2}-N_{3}+1 \end{gathered}$ | $N_{1}+N_{2}+N_{3}-2$ | 2N-2 (1) | 3 | $3 N-2 \quad\left({ }^{(1)}\right.$ |
| 11 |  |  | 0 | 1 | $N \quad$ (1) |
| 12 |  |  | 0 | 1 | $N$ (1) |
| 13 |  |  | 0 | 1 | $N \quad$ (1) |
| 14 | $\begin{gathered} 2 N_{2} N_{3}+ \\ \left(N_{1}-2\right)\left(N_{2}+N_{3}-1\right) \end{gathered}$ |  | int\{SN/2\}-2 (1) | 4 | 4N-3 (1) |
| 15 |  |  | 0 | 2 | 2N-1 (1) |
| 16 | $w N_{1}$ |  | $\underline{\max \{p, q\}+p}$ (z) | 2 | $2 N-1 \quad$ (1) |
| 17 |  | $N_{1}+2 N_{2}+N_{3}-3$ | $2 q+p$ | 1 | $N \quad(1)$ |
| 18 | $w_{1} w_{2}$ | $N_{1}+N_{2}+N_{3}-2$ | w-1 (3) | 3 | $3 N-2 \quad{ }^{(1)}$ |
| 19 |  |  | w-1 (3) | 1 | $N \quad(1)$ |

Note: (1) $N_{1}=N_{2}=N_{3}=N$; (2) $p$ and $q$ are lower and upper bandwidth of $B$; (3) $w_{1}=w_{2}=w$

Table 3.4 Performance of different systolic arrays for matrix multiplication

| Ver. | A | $T_{\text {c }}$ | $T_{d}$ | $\alpha$ | $\beta$ | $S$ | $S_{m}$ | $E$ | $E_{\sim}$ | $1 / O$ | $A \beta^{2}$ |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 1 | $N^{2}$ | 3 N | $N$ | 1 | $N$ | $N / 4$ | $N^{2}$ | 1/4 | 1 | $4 N$ | $N^{*}$ |
| 2 | $N^{2}$ | $3 N$ | $N$ | 1 | $N$ | $N^{2} / 4$ | $N^{2}$ | $1 / 4$ | I | $4 N$ | $N^{*}$ |
| 3 | $N^{2}$ | $3 N$ | $N$ | 1 | $N$ | $N 2 / 4$ | $N{ }^{2}$ | 1/4 | 1 | $4 N$ | $N^{*}$ |
| 4 | $2 N^{2}$ | 3 N | 2 N | 2 | $2 N$ | $N / 5$ | $N^{2} / 2$ | 1/10 | $1 / 4$ | $8 N$ | $8 N^{4}$ |
| 5 | $2 N^{2}$ | $4 N$ | $N$ | 1 | $N$ | $N^{2} / 5$ | $N^{2}$ | $1 / 10$ | $1 / 2$ | $8 N$ | $2 N^{4}$ |
| 6 | $2 N^{2}$ | $3 N$ | 2 N | 2 | $2 N$ | $N 2 / 5$ | $N^{12} 2$ | $1 / 10$ | $1 / 4$ | $8 N$ | $8 N^{\prime}$ |
| 7 | $2 N^{2}$ | $4 N$ | $N$ | 1 | $N$ | $N^{2} / 5$ | $N$ | 1/10 | 112 | $8 N$ | $2 N^{4}$ |
| 8 | $2 N^{2}$ | $3 N$ | $N$ | 2 | $2 N$ | $N^{2} / 4$ | $N^{2} / 2$ | $1 / 8$ | $1 / 4$ | $8 N$ | $8 N^{*}$ |
| 9 | $2 N^{2}$ | $4 N$ | $N$ | 1 | $N$ | $N \% / 5$ | $N^{2}$ | $1 / 10$ | $1 / 2$ | $8 N$ | $2 N^{\prime \prime}$ |
| 10 | $3 N^{2}$ | $3 N$ | 2 N | 3 | $3 N$ | N2/5 | $N^{2} / 3$ | $1 / 15$ | 1/9 | 12 N | $27 N^{4}$ |
| 11 | $3 N^{2}$ | $3 N$ | 0 | 1 | $N$ | $N^{2 / 3}$ | $N^{2}$ | $1 / 9$ | 1/3 | 12 N | $3 N^{4}$ |
| 12 | $3 N^{2}$ | $3 N$ | 0 | 1 | $N$ | $N: 3$ | $N$ | 1/9 | $1 / 3$ | 12 N | $3 N^{4}$ |
| 13 | $3 N^{2}$ | 3 N | 0 | 1 | $N$ | $N^{2} / 3$ | $N^{2}$ | 1/9 | 1/3 | 12 N | $3 N^{4}$ |
| 14 | $4 N^{2}$ | $3 N$ | $5 N^{\prime} / 2$ | 4 | $4 N$ | $2 N^{2} / 1 /$ | $N^{2 / 4}$ | $1 / 22$ | 1/16 | 18 N | $64 N^{4}$ |
| 15 | $4 N^{2}$ | 3 N | 0 | 2 | $2 N$ | $N^{2 / 3}$ | $N / 2$ | $1 / 12$ | 1/8 | 18 N | $16 N^{*}$ |
| 16 | $w N$ | $3 N$ | $w$ | 2 | $2 N$ | $w N / 3$ | $w N / 2$ | $1 / 3$ | $1 / 2$ | $\pm N$ | $4 w N^{3}$ |
| 17 | $w N$ | $4 N$ | $3 w / 2$ | 1 | $N$ | wN/ 4 | $w N$ | $1 / 4$ | 1 | $4 N$ | $w N^{3}$ |
| 18 | $w^{2}$ | $3 N$ | w | 3 | 3 N | w2/3 | $w^{2 / 3}$ | $1 / 3$ | //3 | $8 w$ | $9 w^{2} N^{2}$ |
| 19 | $w^{\prime 2}$ | $3 N$ | $w$ | 1 | $N$ | $w^{2 / 3}$ | $w^{2}$ | $1 / 3$ | 1 | $8 w$ | $w^{2} N^{2}$ |

Note: (1) $N_{1}=N_{2}=N_{3}=N$; (2) $w_{1}=w_{2}=w$; (3) $N=-l$; (4) $N \leq-=w$;
(5) $T_{s}=N^{3}$ for versions 1-15; $T_{s}=w N^{2}$ for versions 16-17; $T_{s}=w^{2} N$ for versions 18-19.


Figure 3.7 Systolic array for matrix multiplication: Version I, $U^{\top}=\left[\begin{array}{lll}0 & 0 & 1\end{array}\right]$


Figure 3.9 Systolic array for matrix multiplication: Version 3, $\boldsymbol{U}^{\mathrm{T}}=\left[\begin{array}{lll}1 & 0 & 0\end{array}\right]$


Figure 3.8 Systolic array for matrix multiplication: Version 2, $U^{\mathrm{T}}=\left[\begin{array}{lll}0 & 1 & 0\end{array}\right]$


Figure 3.10 Systolic array for matrix multiplication: Version $4, U^{\mathrm{T}}=\left[\begin{array}{lll}0 & 1 & 1\end{array}\right]$


Figure 3.11 Systolic array for matrix multiplication: Version $5, U^{\mathrm{T}}=\left[\begin{array}{lll}0 & 1 & -1\end{array}\right]$


Figure 3.14 Systolic array for matrix multiplication: Version 8, $\boldsymbol{U}^{\mathrm{T}}=\left[\begin{array}{lll}1 & 1 & 0\end{array}\right]$


Figure 3.12 Systolic array for matrix multiplication: Version $6, U^{\mathrm{T}}=\left[\begin{array}{lll}1 & 0 & 1\end{array}\right]$


Figure 3.13 Systolic array for matrix multiplication: Version $7, \boldsymbol{U}^{\mathrm{T}}=\left[\begin{array}{lll}1 & 0 & -1\end{array}\right]$


Figure 3.15 Systolic array for matrix multiplication: Version 9, $\boldsymbol{U}^{\mathrm{r}}=\left[\begin{array}{lll}1 & -1 & 0\end{array}\right]$


Figure 3.16 Systolic array for matrix multiplication: Version $10, U^{\mathrm{T}}=\left[\begin{array}{lll}1 & 1 & 1\end{array}\right]$


Figure 3.17 Systolic array for matrix multiplication: Version 11, $U^{\mathrm{T}}=\left[\begin{array}{lll}1 & 1 & -1\end{array}\right]$


Figure 3.18 Systolic array for matrix multiplication: Version $12, U^{\mathrm{T}}=\left[\begin{array}{lll}1 & -1 & 1\end{array}\right]$


Figure 3.19 Systolic array for matrix multiplication: Version $13, U^{\mathrm{T}}=\left[\begin{array}{lll}-1 & 1 & 1\end{array}\right]$


Figure 3.20 Systolic array for matrix multiplication: Version 14, $U^{\text {T }}$
$=\left[\begin{array}{lll}2 & 1 & 1\end{array}\right]$


Figure 3.21 Systolic array for matrix multiplication: Version $15, U^{\top}$ $=\left[\begin{array}{lll}2 & 1 & -1\end{array}\right]$


Figure 3.22 Systolic array for matrix multiplication: Version 16, $U^{\text {「 }}$


Figure 3.23 Systolic array for matrix multiplication: Version $17, U^{\top}$
$=\left[\begin{array}{lll}0 & -1 & 1\end{array}\right]$


Figure 3.24 Systolic array for matrix multiplication: Version $18, U^{\mathrm{T}}=\left[\begin{array}{lll}1 & 1 & 1\end{array}\right]$


Figure 3.25 Systolic array for matrix
multiplication: Version 19, $U^{\mathrm{T}}=\left[\begin{array}{lll}1 & 1 & -1\end{array}\right]$

## Chapter 4

## SYSTOLIC ARRAY SYNTHESIS USING SFG REPRESENTATIONS

### 4.1 Introduction

T HE systolic design flow graph (SFG) was introduced by S.Y. Kung [KungSY84]. Given an algorithm, the method starts from the dependence graph (DG) of the algorithm. Then the DG is mapped to an SFG. Using the obtained SFG representing the algorithm, this method proceeds by choosing basic operational modules that correspond to the functional description of PEs of the architecture. Localization


Figure 4.1 Y chart of SFG methodology rules are then applied to derive a regular and temporally localized SFG. The localization procedure consists of selecting cut-sets of the SFG and reallocating scaled delays to edges "leaving" and "entering" the cut-set in such a way that at least one unit of time is allowed for communicating a signal between two nodes. Delays are combined with operational modules to obtain a full description of the operation of a basic systolic module. The resulting SFG maps straightforwardly into the systolic array by mapping basic modules into PEs and edges into interconnections. Timing and data movement can be derived from the basic modules due to the localized spatial and temporal characteristics of the SFG. Figure 4.1 shows the Y chart for the SFG design methodology.

In this chapter, we will introduce this design methodology along with an example for matrix triangularization which find many applications in a variety of scientific and engineering problems. The rest of this chapter is organised as follows. Sec. 4.2 introduces the systematic procedure of mapping DGs to SFGs. Sec 4.3 presents the methodology of mapping SFGs to systolic arrays. Sec. 4.4 summarizes the design procedure of the SFG methodology. Sec. 4.5 designs systolic arrays for matrix triangularization using the SFG methodology. Sec. 4.6 discusses some aspects of optimal design for systolic arrays. Finally, Sec. 4.7 summarizes the chapter.

### 4.2 Mapping DGs onto SFGs

### 4.2.1 DG Design

As discussed in Chap. 3, in order to achieve the maximal parallelism in a (single assignment) algorithm, we must carefully study the data dependencies in the computations. The dependence graph (DG) of an algorithm can capture the information regarding the parallelism in the algorithm. We again give the definition of a dependence graph (DG) as follows.

Dependence Graph: A dependence graph (DG) is a directed graph, where a node with index $\boldsymbol{k}$ represents computations of variables associated with index $\boldsymbol{k}$ in a single assigument form, and an arc from node $\boldsymbol{i}$ to $\boldsymbol{j}$ denote a data dependency from a variable with index $\boldsymbol{i}$ to a variable with index $\boldsymbol{j}$.

A DG can be considered as the graphical representation of a single assignment algorithm. For example, we consider matrix-vector multiplication algorithm given in a single assignment form as follows:

$$
\begin{align*}
& c=A b \quad A \in \mathbf{R}^{N \bullet N}, \quad b, c \in \mathbf{R}^{N \times 1} \\
& c(i, j)=c(i, j-1)+a(i, j) \times b(j) \quad \text { for all } i \text { and } j, \text { with } c(i, 0)=0 \tag{4.1}
\end{align*}
$$

The dependence graph of the above algorithm is shown in Figure 4.2a. The operations inside each node are ignored for simplicity. It is very straightforward to extend the DG concept to include the operations inside each node. The DG shown in Figure 4.2 is called a complete $D G$, which specifies all the dependencies between all variables in the index space. An algorithm is computable if and only if its complete DG contains no loops or cycles.


Figure 4.2 DG for matrix-vector multiplication (a) with global communication and (b) with only local communication

Localized Dependence Graph. In Figure 4.2a, the value $b(j)$ of each element of vector $\boldsymbol{b}$ should be "broadcast" to all the index points having the same $j$-index. This kind of data is termed broadcast data. In general, this means that global communication is involved in array processor design. In many cases, such broadcasting can be avoided and replaced by local communication. An algorithm is localized if all variables are directly dependent upon the variables of neighbouring nodes only. As an example, a localized DG is shown in Figure 4.2 b , where $b(j)$ is "propagated" step by step, without being modified, to all the nodes with the same $j$-index. This kind of data, which is propagated without being modified, is called transmittent data. Otherwise, it is called nontransmittent data. The corresponding localized single assignment algorithm for the matrix-vector multiplication is shown next:

$$
\begin{align*}
& b(i, j)=b(i-1, j) \\
& c(i, j)=c(i, j-1)+a(i, j) \times b(i, j) \tag{4.2}
\end{align*} \quad \text { for all } i \text { and } j, \text { with } c(i, 0)=0
$$

Shift-Invariance of DG. A DG is shift-invariamt if the dependence arcs corresponding to all nodes in the index space remain unchanged with respect to their positions. Formally, this means that if a variable at $\boldsymbol{k}_{1}$ depends on a variable at $\boldsymbol{k}_{1}-\boldsymbol{d}$, then a variable at $\boldsymbol{k}_{2}$ will dependent on a variable at $\boldsymbol{k}_{\mathbf{2}} \boldsymbol{-} \boldsymbol{d}$ in the same manner. Note that the node functions can be different and the border I/O nodes are exempted from such a condition (c.f. Sec. 3.2 for definition of RIA).

### 4.2.2 Signal Flow Graph (SFG)

The SFG offers a powerful abstraction and graphical representation for problems in scientific and signal processing computations. Below we present a formal definition of SFG.

Signal Flow Graph (SFG). A signal flow graph (SFG) is a directed graph defined by a triple

$$
\begin{equation*}
S F G \triangle\{V, E, D(e)\} \tag{4.3}
\end{equation*}
$$

where nodes in $V$ model computations and edges in $E$ model one-way (directed) communications. Each edge, $e$, has an edge-delay, represented by $D(e)$.

In general, a node is often denoted by a circle representing an arithmetic or logic function performed with zero delay. An edge, on the other hand, denotes either a dependence relation or a delay. When an edge is labelled with a capital letter $D$ (or $2 D, \ldots$ ), it represents a time delay operator with delay time $D$ (or $2 D, \ldots$ ). Examples of the SFGs will be given in the next subsection.

A complete SFG description should include both functional and structural description parts. The functional description defines the behaviour within a node, whereas the structural description specifies the interconnection (edges and delays) between the nodes.

As compared with the DG, the SFG has the following properties:

- The SFG can be viewed as a simplified graph which has less nodes than the corresponding DG. That means the SFG is a more concise representation of an algorithm than the DG.
- The SFG is more specific and is closer to hardware level design. Therefore, the SFG also dictates the type of arrays that will be obtained.
- While there are no loops in any DG, the SFG can have loops, as long as there is at least one delay $D$ on each loop.


### 4.2.3 Mapping DG to $S F G$

There are two basic considerations for mapping from a DG to an SFG:

- To which processors should operations be assigned?
- In what ordering should the operations be assigned to a processor?

Therefore, two steps are involved in mapping a DG to an SFG array. The first step is the processor assignmem. Once the processor assignment is fixed, the second step is the scheduling. Similar to the RIA design method, we usually use a linear projection for processor assignment, in which nodes of the DG in a certain straight line are projected to a PE in the SFG array, and a linear scheduling for schedule assignment, in which nodes on a parallel hyperplane in the DG are scheduled to be processed at the same time step.

Processor Assignment. As a simple example, a projection method may be applied, in which nodes of the DG along a straight line are assigned to a common PE. Mathematically, a linear projection is often represented by a projection vector $d$. The results of this projection are represented by an SFG (see Figure 4.3a). As an example, the 2-D index space of matrix-vector multiplication as shown in Figure 4.2 may be decomposed into a direct sum of a 1-D processor space and 1-D delay space (cf. iteration space in the RIA method). The delay space is related to the scheduling as explained below.


Figure 4.3 The SFG of matrix-vector multiplication: (a) a linear projection with projection vector $\boldsymbol{d}$; (b) a linear schedule $s$ and its hyperplanes

Scheduling. The projection should be accompanied by a scheduling scheme, which specifies the sequence of the operations in all the PEs. A linear schedule is based on a set of parallel and uniformly spaced hyperplanes in the DG. These hyperplanes are called equitemporal hyperplanes, all the nodes on the same hyperplane must be processed at the same time. Mathematically, the schedule can be represented by a schedule vector $s$, pointing to the normal direction of the hyperplane. For any index point $\boldsymbol{k}$ in the DG, its time step is $\boldsymbol{s}^{\mathrm{T}} \boldsymbol{k}$. A set of linear schedule hyperplanes and the schedule vector are illustrated in Figure 4.3b.

Given a DG and a project direction $d$, we note that not all hyperplanes qualify to be a valid schedule for the DG. Some of them violate the precedence relation of computation specified by the dependence arcs. The allowable directions of the hyperplanes actually define the class of permissible linear schedules. In order for the given hyperplanes to represent a permissible linear schedule, it is necessary and sufficient that the normal vector $s$ satisfies the following two conditions:

$$
\begin{align*}
& \text { (1) } s^{T} e \geq 0, \quad \text { for any dependence arc } e \\
& \text { (2) } s^{T} d>0 \text {. } \tag{4.4}
\end{align*}
$$

In other words, the schedule is permissible if and only if (1) all the dependency arcs flow in the same direction across the hyperplanes; and (2) the hyperplane are not parallel with projection vector $\boldsymbol{d}$.

Given a DG and a projection vector $d$, the most likely used schedules for the SFG projection are the following:

- Default schedule. The corresponding hyperplanes are orthogonal to the projection direction $\boldsymbol{d}$; or the normal direction of hyperplanes $\boldsymbol{s} \boldsymbol{s}$ is parallel to the projection direction $\boldsymbol{d}$.
- Recursion schedule. The schedule vector $s$ is parallel to one of the axes in the index space of the DG. Usually, the one corresponding to the recursion numbering is used.
- Systolic schedule. The systolic schedule means that there is at least one delay on each edge of the resulting SFG (this schedule has been treated in Chap. 3 in an algebraic form).

In this chapter, only the first two schedules are discussed. We will use a retiming technique to systolize the obtained SFG.

Algebraic Approach for SFG Projection. In addition to the graph-based projection shown in Figure 4.3, we describe a formal algebraic approach below. Given a DG of dimension $N$, a projection vector $\boldsymbol{d}$, and a permissible linear schedule $s$, an SFG may be derived based on the following mappings.

- Node mapping. This mapping assigns the node activity in the DG to PEs. The index space of dimension $N$ is mapping to the processor space of dimension $N-1$. The mapping of a computation at index $\boldsymbol{k}$ in the DG onto a node $v$ in the SFG is found by

$$
\begin{equation*}
\nu=P^{T} k \tag{4.5}
\end{equation*}
$$

where the processor basis $P$, denoted by an $N \times(N-1)$ matrix, is orthogonal to $\boldsymbol{d}$, i.e.,

$$
\begin{equation*}
P^{T} d=0 \tag{4.6}
\end{equation*}
$$

- Arc mapping. This mapping maps the arcs of the DG to the edges of the SFG. The set of edges $\boldsymbol{c}$ into each node of the SFG and the number of delays $D(\boldsymbol{c})$ on every edge are derived from the set of dependence edge $b$ at each point in the DG by

$$
\left[\begin{array}{c}
D(e)  \tag{4.7}\\
e
\end{array}\right]=\left[\begin{array}{l}
s^{T} \\
P^{T}
\end{array}\right] b
$$

- I/O mapping. The SFG node position $v$ and time $t(\boldsymbol{k})$ of an input of the DG computation at index $\boldsymbol{k}$ is derived by

$$
\left[\begin{array}{c}
t(k)  \tag{4.8}\\
v
\end{array}\right]=\left[\begin{array}{c}
s^{r} \\
P^{r}
\end{array}\right] k
$$

A similar mapping applies to output nodes.

### 4.3 Mapping SFGs to Systolic Arrays

In the last chapter, we described the algebraic mapping method from the DGs to systolic arrays. There are several reasons that one might want to derive an SFG array and then convert it into a systolic array: (1) the SFG offers a concise expression for parallel algorithms, (2) the SFG defines the structure of the array with minimum constraints on timing, and (3) formal transformations from an SFG to a systolic array can be developed.

Obviously, in the mapping form DGs to SFGs, not all SFG schedules $s$ complying with Eq. (4.4) satisfy the conditions of the systolic schedule, which requires that

$$
\begin{equation*}
s^{T} e>0 \quad \text { and } \quad s^{T} d>0 \tag{4.9}
\end{equation*}
$$

An SFG array is in fact very close to a systolic array. The major gap between an SFG array and a systolic array is that most SFGs are not temporally localized because an SFG can have zero-delay edges which means that data from one node may be immediately available for other nodes and broadcast data may be involved. This argument can be described in other words

$$
\text { systolic array }=\text { SFG array + pipeline retiming }
$$

Here, retiming is the procedure to transform an SFG to an equivalent and temporal localized form. In this section we present a cut-set retiming procedure [KungSY84] [McEvoy91] to systolize SFGs.

### 4.3.1 Cut-Set Retiming Procedure

The objective of the cut-set retiming procedure is to convert an SFG into a temporally localized form so that all the edges between modular sections have at least one delay element.

Definition. A cut-set in an SFG is a minimal set of edges which partition the SFG into two parts.

A more formal definition of the cut-set can be found in [McEvoy91]. By removing a cut-set from its corresponding SFG, the SFG can be divided into two disconnected parts. By the minimal set, it means that no strict subset of the cut-set can form another cut-set. The cut-set retiming procedure is based on two simple rules:

Rule 1: Time Scaling. All delays $D$ may be scaled by a single positive integer $\alpha$, i.e., $D=\alpha D^{\prime}$. The integer $\alpha$ is also known as the pipelining period of the SFG . Correspondingly, the input and output rates also have to be scaled by a factor of $\alpha$ (with respect to the new time unit $D^{\prime}$ ).

Rule 2: Delay-Transfer. Given any cut-set of the SFG, which partitions the graph into two components, we can group the edges of the cutset into inbound edges and outbound edges, as shown in Figure 4.4. Delay-transfer rule allows advancing $k D^{\prime}$ on all the outbound edges and delaying $k D^{\prime}$ on the inbound edges, or vice versa. It is clear that, for an SFG, the general system


Figure 4.4 Illustration of delay-transfer rule behaviour is not affected, because the effects of delays and advances cancel each other in the overall timing. Note that the input-input and input-output timing relations also remain exactly the same only if they are located on the same side of the cut-set. Otherwise, they should be adjusted by a delay of $k D^{\prime}$ or an advance of $k D^{\prime}$. In other words, if there is more than one cut-set involved and if the input and the output are separated by more than one cutset, then such adjustment factors should be accumulated.

These two basic rules are referred to as the cut-set retiming rules. Below we will prove that any computable SFG (one without zero-delay loops or cycles) can be systolized by the cut set procedure.

Retiming Theorem. All computable SFGs can be made temporally local by applying the two cut-set rules. Consequently, a spatially local and regular SFG is always systolizable.

Proof: We claim that the two retiming rules can be used to localize any targeted zero-delay edge, i.e., to convert it into à nonzero-delay edge. This is done by choosing a good cut-set and applying the rules to it. A good cut-set, including the target edge, should not include any bad edges, i.e., those zerodelay edges in the opposite direction of the target edge. This means that the good cut-set will include only (1) the target edge, (2) nonzero-delay edges going in either direction, and (3) zero-delay edges going in the same direction. Then according to Rule 2, the nonzero delays of the opposite-direction edges can give one or more spare delays to the target edge in order to localize it. If there are no spare delays to give away, all delays in the SFG are simply scaled according to Rule I to create enough delays for the transfer needed.

Therefore, the only thing need to prove is that such a good cut-set always exist for a computable SFG. For this, we refer to Figure 4.5 , where only the zero-delay successor edges and the zero-delay predecessor edges connected to the target edge have been kept, and all the other edges have been removed from the graph. Between the two sets of the edges, there exist no bad edges which have zero-delay in the opposite direction of the target edge. In other words, if there is any zero-delay edge (as shown by the


Figure 4.5 Proof of cut-set procedure dash line in Figure 4.5) between the two sets of edges other than the target edge, they must not be in the opposite direction of the target edge. Otherwise, some set of zero delay edges would form a zero-delay cycle, and thus the SFG would not be computable. Therefore, there always exists a good cut-set in a computable SFG. It is clear that repeatedly applying the two retiming rules will eventually lead to a temporally localized SFG.

### 4.3.2 Systolization Procedure

A regular SFG array can be easily systolized to become a systolic array by the following systolization procedure which is based essentially on the cut-set retiming rules:

1. Selection of Basic Operation Modules. The choice may not be unique. Generally, the finer the granularity of the basic modules, the faster a systolic array is.
2. Applying Retiming Rules. If the given SFG is regular, i.e., modular and spatially local, then regular cut-sets can be selected and the above retiming rules can be applied to derive a regular and temporally localized SFG. In order to preserve the modular structure of the SFG, the cut-set retiming should be applied uniformly across the network.
3. Combination of Delay and Operation Modules. To convert the obtained temporally-localized SFG into a systolic array, it is only necessary to successfully introduce a delay into each of the operation modules. The delay can then be combined with the module operation to form a basic systolic element. All the extra delays are modeled as pure delays without operations. Since self-loops are implemented as registers in the PE , they are also combined into the PE. Figure 4.6 shows this procedure.


Figure 4.6 Illustration of combing delays into module operations. (a) Module operation with delays in the circle; (b) The corresponding systolic array processing elements

### 4.4 A Step-by-Step Mapping Procedure

Step 1: DG Design. For a given problem, one first identifies a suitable algorithm. Then a suitable algorithm expression, e.g., single assignment form, should be developed. Based on the appropriate algorithm expression, a DG is designed. Note that although many methods have been proposed to construct a DG from sequential code, a formal and automatic methodology remains a major open
research problem. Since the structure of a DG greatly affects the final array design, further modification on the DG are often desirable in order to achieve better design [KungSY88b]. In order to implement the algorithm in a systolic array, a regular (or shift-invariant) and localized DG is desired.

Step 2: SFG Design. Based on different mappings of the DG onto array structure, a number of SFGs can be derived form the DG. The permissible linear schedules should be used when mapping the DGs to SFGs. A defanlt schedule will be usually selected. Algebraic approach for SFG projection can be used to describe the mapping procedure formally.

Step 3: Systolic Array Design. The SFG obtained in Step 2 can be systolized by applying the cut-set retiming rules. Firstly, a basic operation module is selected. Then the two cut-set rules can be used to derive a regular and temporally localized SFG. By combining delay and operation modules, it is straightforward to convert the retimed SFG into a systolic array.

The above design procedure is illustrated in Figure 4.7. In the next section, we will exemplify this mapping procedure by designing systolic arrays for matrix triangularization


Figure 4.7 Illustration of the step-by-step mapping procedure for systolic array design
problems: LU decomposition and QR decomposition..

### 4.5 Systolic Array for Matrix Triangularization

Matrix triangularization is one of the fundamental matrix operations. It has found its application in solving linear equations, solving least square problems, calculating eigenvalues, and so on. There are many methods to triangularize a matrix. Gaussian elimination, LU decomposition and QR decomposition are the most commonly used methods. In this section, we will design systolic arrays for LU decomposition and QR decomposition algorithm.

### 4.5.1 Mapping Algorithms to DGs

### 4.5.1.1 LU Decomposition

A factorization of a non-singular square matrix $\boldsymbol{A}$ as $\boldsymbol{A}=\boldsymbol{L} \boldsymbol{U}$, where $L$ is lower triangular with diagonal entries being l's and $\boldsymbol{U}$ is upper triangular, is called an $L U$ decomposition of $\boldsymbol{A}$. If $\operatorname{det}(A(1: k, 1: k)) \neq 0$ for $k=1: N$, then there exists a unique LU decomposition of $\boldsymbol{A}$ [Golub89]. The LU decomposition of $\boldsymbol{A}$ can be expressed as follows

$$
\left[\begin{array}{cccc}
a_{11} & a_{12} & \cdots & a_{1 N}  \tag{4.10}\\
a_{21} & a_{22} & \cdots & a_{2 N} \\
\vdots & \vdots & \ddots & \vdots \\
a_{N 1} & a_{N 2} & \cdots & a_{N N}
\end{array}\right]=\left[\begin{array}{cccc}
1 & & & \\
l_{21} & 1 & & \\
\vdots & \vdots & \ddots & \\
l_{N 1} & l_{N 2} & \cdots & 1
\end{array}\right]\left[\begin{array}{cccc}
u_{11} & u_{12} & \cdots & u_{1 N} \\
& u_{22} & \cdots & u_{2 N} \\
& & \ddots & \vdots \\
& & & u_{N N}
\end{array}\right]
$$

A recursive algorithm for finding the LU decomposition of matrix $\boldsymbol{A}$ can be given by

$$
\begin{array}{cc}
\text { for } k=1: N \\
& \text { for } \quad i=k+1: N \\
& a_{i k}=a_{i k} / a_{k k} \\
& \text { for } j=k+1: N  \tag{4.11}\\
& \text { for } i=k+1: N \\
& a_{i j}=a_{i j}-a_{i k} a_{k j} \\
\text { with } \quad l_{i j}= & =a_{i j} \text { for } i>j, \quad u_{i j}=a_{i j} \text { for } i \leq j
\end{array}
$$

A single assignment formation of the proceeding recursive algorithm has the following form

```
for \(k=1: N\), for \(i=k: N\), for \(j=1: N\)
    \(u(i, j, k)= \begin{cases}a(i, j, k-1) & \text { if } i=k \\ u(i-1, j, k) & \text { otherwise }\end{cases}\)
    \(l(i, j, k)= \begin{cases}a(i, j, k-1) / u(i, j, k) & \text { if } j=k \\ l(i, j-1, k) & \text { otherwise }\end{cases}\)
    \(a(i, j, k)=a(i, j, k-1)-l(i, j, k) * u(i, j, k) \quad\) if \(j \neq k\)
with \(a(i, j, 0)=a_{i j}\)
```

Thus we can obtain the DG for the LU decomposition algorithm, which is shown in Figure 4.8. In each horizontal plane, the points that serve as the source of the row and column values are represented as dark dots. (The algorithm in Eq. (4.12) is slightly different from the algorithm given by S.Y. Kung in [KungSY88a] in computing $a(i, j, k)$ only when $j \neq k$. The benefit due to this modification will be shown later.)


Figure 4.8 DG for LU decomposition

### 4.5.1.2 QR Decomposition

An $M \times N$ matrix $A$ with $M \geq N$ can be written as $A=Q R$ where $Q$ is an $M \times M$ matrix with orthonormal columns (i.e. $Q^{\mathrm{T}} Q=I$ ) and $\boldsymbol{R}$ is an $M \times N$ upper triangular matrix. This decomposition is called QR decomposition. The QR decomposition can be obtained by a sequence of Givens rotations (GRs). The GR is a numerically stable orthogonal operator that performs a plane rotation of the matrix $A$. The purpose of these rotations is to annihilate the subdiagonal elements of the matrix $\boldsymbol{A}$ and reduce it to upper triangular form. In the Givens algorithm, the subdiagonal elements of the first column are nullified first, then the elements of the second column, and so forth until an upper triangular form is eventually reached. Below we briefly discuss how the QR decomposition can be obtained by using GRs. The upper triangular matrix $R$ is obtained as follows:

$$
\begin{align*}
& Q^{T} A=R \\
& Q^{T}=Q_{N} Q_{N-1} \cdots Q_{1}  \tag{4.13}\\
& Q_{p}=Q_{(p, p)} Q_{(p+1, p)} \cdots \varrho_{(M-1, p)}
\end{align*}
$$

where $Q_{(i j)}$ is the Givens rotation operator used to annihilate the matrix element located at the $(i+1)$ st row and $j$ th column and has the following form

where $c=\cos \theta, s=\sin \theta$, and $\theta=\tan ^{-1}\left[a_{i+1, j} / a_{i, j}\right]$. (Notice that if $M=N$, then the annihilation operations are not necessary for the last column of the matrix.) The operation to compute the matrix product $\boldsymbol{A}^{\prime}=\boldsymbol{Q}_{(i, j)} \boldsymbol{A}$ is named Givens rotation (GR) and the operation of creating $\cos \theta$ and $\sin \theta$ is named Givens generation (GG). The matrix product $\boldsymbol{A}^{\prime}=\boldsymbol{Q}_{(i, j)} \boldsymbol{A}$ differs from $\boldsymbol{A}$ only in $i$ th and $(i+1)$ st rows and this can be expressed by

$$
\left.\begin{array}{l}
a_{i, k}=a_{i, k} \cos \theta+a_{i+1, k} \sin \theta  \tag{4.15}\\
a_{i+1, k}=-a_{i, k} \sin \theta+a_{i+1, k} \cos \theta
\end{array}\right\} \quad \text { for all } k=1, \ldots, N
$$

The full procedure of the $Q R$ decomposition is illustrated below using a $4 \times 3$ example:

$$
\begin{align*}
& A=\left[\begin{array}{llll}
x & x & x & x \\
x & x & x & x \\
x & x & x & x \\
x & x & x & x \\
x & x & x & x
\end{array}\right] \Longrightarrow\left[\begin{array}{llll}
x & x & x & x \\
x & x & x & x \\
Q_{4,1} \\
x & x & x & x \\
x & x & x & x \\
0 & x & x & x
\end{array}\right] \Longrightarrow\left[\begin{array}{llll}
x & x & x & x \\
Q_{3,1} & x & x & x \\
x & x & x & x \\
0 & x & x & x \\
0 & x & x & x
\end{array}\right] \Rightarrow\left[\begin{array}{llll}
x & x & x & x \\
x & x & x & x \\
Q_{2,1}
\end{array} \Rightarrow\left[\begin{array}{llll}
x & x & x & x \\
0 & x & x & x \\
0 & x & x & x \\
0 & x & x & x
\end{array}\right] \Rightarrow\left[\begin{array}{lll} 
\\
Q_{1,1} \\
0 & x & x
\end{array}\right]\right. \\
& \begin{array}{l}
Q_{4,2}\left[\begin{array}{llll}
x & x & x & x \\
0 & x & x & x \\
0 & x & x & x \\
0 & x & x & x \\
0 & 0 & x & x
\end{array}\right] \Longrightarrow\left[\begin{array}{llll}
x & x & x & x \\
0 & x & x & x \\
0 & x & x & x \\
0 & 0 & x & x \\
0 & 0 & x & x
\end{array}\right] \Longrightarrow\left[\begin{array}{llll}
x & x & x & x \\
0 & x & x & x \\
Q_{2,2} & 0 & x & x \\
0 & 0 & x & x \\
0 & 0 & x & x
\end{array}\right] \stackrel{Q_{4,3}}{\Longrightarrow}\left[\begin{array}{llll}
x & x & x & x \\
0 & x & x & x \\
0 & 0 & x & x \\
0 & 0 & x & x \\
0 & 0 & 0 & x
\end{array}\right]\left[\begin{array}{llll}
x & x & x & x \\
Q_{3,3} \\
0 & x & x & x \\
0 & 0 & x & x \\
0 & 0 & 0 & x \\
0 & 0 & 0 & x
\end{array}\right] \Rightarrow\left[\begin{array}{llll}
x & x & x & x \\
Q_{4,4} & x & x & x \\
0 & 0 & x & x \\
0 & 0 & 0 & x \\
0 & 0 & 0 & 0
\end{array}\right]
\end{array} \tag{4.16}
\end{align*}
$$

The QR decomposition described above can be written in a sequential algorithm as follows

$$
\begin{align*}
& \text { for } k=1: N \\
& \qquad \begin{array}{l}
\text { for } i=(M-1): k \\
\qquad=\frac{a_{i, k}}{\sqrt{\left(a_{i+1, k}^{2}+a_{i, k}^{2}\right)}}, \quad s=\frac{a_{i+1, k}}{\sqrt{\left(a_{i+1, k}^{2}+a_{i, k}^{2}\right)}} \\
\text { for } j=k: N \\
t l=a_{i, j} c+a_{i+1, j} s \\
t 2
\end{array} \\
& a_{i, j}=t 1 \\
& a_{i, j}=t 2 \tag{4.17}
\end{align*}
$$

where $t l$ and $t 2$ are temporary variables to store the new values of $a_{i, j}$ and $a_{i+1, j}$. The matrix $\boldsymbol{A}$ is then updated to yield the $\boldsymbol{R}$ matrix as illustrated in Eq. (4.16).

The sequential algorithm in Eq. (4.17) can be converted to single assignment formulation; however, extra care has to be taken. Some new variables are introduced in order to convert the sequential algorithm into a single assignment format. We denote the "old" value of two elements in a rotation as $o x(i, j, k)$ and $o y(i, j, k)$ and the "new" values of the two elements as $m x(i, j, k)$ and $m y(i, j, k)$. The single assignment form of the QR decomposition is:

```
for \(k=1: N\)
    for \(i=(M-1): k\)
        for \(j=k: N\)
                \(o x(i, j, k)=o x(i, j, k-1)\)
                \(o y(i, j, k)= \begin{cases}n y(i, j, k-1) & \text { if } i=M-1 \\ n x(i+1, j, k) & \text { if } i \neq M-1\end{cases}\)
            \(c(i, j, k)= \begin{cases}o x(i, j, k) / \sqrt{o x^{2}(i, j, k)+o y^{2}(i, j, k)} & \text { if } j=k \\ c(i, j-1, k) & \text { if } j \neq k\end{cases}\)
\(s(i, j, k)= \begin{cases}o y(i, j, k), \sqrt{o x^{2}(i, j, k)+o y^{2}(i-1, j, k)} & \text { if } j=k \\ s(i, j-1, k) & \text { if } j \neq k\end{cases}\)
            \(n x(i, j, k)=c(i, j, k) * o x(i, j, k)+s(i, j, k) * o y(i, j, k)\)
            \(n y(i, j, k)=-s(i, j, k) * o x(i, j, k)+c(i, j, k) * o y(i, j, k)\)
with \(n x(i, j, 0)=a_{i j} n y(M-1, j, 0)=a_{M j} r_{i j}=n x(i, j, i)\)
```

From Eq. (4.18), we can easily obtain the DG for the QR decomposition algorithm by comparing the index differences between each assignment statement. The obtained DG for QR decomposition algorithm given in Eq. (4.18) is depicted in Figure 4.9 where an example of $5 \times 4$ matrix is used. The dark nodes in the DG represent the GGs while the rest nodes represent the GRs.

There exist different annihilation orders for the QR decomposition. Now we consider another commonly used annihilation scheme for QR decomposition. Instead of applying Givens' rotation to the neighbouring rows, we can also adopt an annihilation order similar to that of LU decomposition discussed before. Firstly, the first column entries of the input matrix $A$ below the main diagonal are annihilated by applying GRs between the first row and the


Figure 4.9 DG for QR decomposition rest rows. Then the $k$ th column of the updated matrix below main diagonal is annihilated by applying GRs between $k$ th row and the rest rows. Finally the original matrix can be triangularized. The algorithm for the QR decomposition using new annihilation scheme can be expressed as

$$
\begin{align*}
& \text { for } k=1: N \\
& \text { for } i=(k+1): M \\
& \qquad=a_{k, k} / \sqrt{\left(a_{k, k}^{2}+a_{i, k}^{2}\right)}, \quad s=a_{i, k} / \sqrt{\left(a_{k, k}^{2}+a_{i, k}^{2}\right)} \\
& \text { for } j=k: N \\
& t l=a_{k j} c+a_{i, j} s  \tag{4.19}\\
& t 2=-a_{k, j} s+a_{i, j} \\
& a_{k j}=t l \\
& a_{i, j}=t 2
\end{align*}
$$

Now we propose a new single assignment format QR decomposition algorithm as follows

$$
\begin{align*}
& \text { for } k=1: N, \text { for } i=k \cdot M, \text { for } j=k \cdot N \\
& m(i, j, k)=a(i, j, k-1) \text { if } i=k \\
& c(i, j, k)= \begin{cases}m(i-1, j, k) / \sqrt{m^{2}(i-1, j, k)+a^{2}(i, j, k-1)} & \text { if } j=k \\
c(i, j-1, k) & \text { if } j \neq k\end{cases} \\
& s(i, j, k)=\left\{\begin{array}{ll}
a(i, j, k-1) / \sqrt{m^{2}(i-1, j, k)+a^{2}(i, j, k-1)} & \text { if } j=k \\
s(i, j-1, k) & \text { if } j \neq k
\end{array}\right\} \text { if } i \neq k  \tag{4.20}\\
& m(i, j, k)=c(i, j, k) m(i-1, j, k)+s(i, j, k) a(i, j, k-1) \\
& a(i, j, k)=-s(i, j, k) m(i-1, j, k)+c(i, j, k) a(i, j, k-1)
\end{align*}
$$

Based on Eq. (4.20), we can obtain another DG for the $Q R$ decomposition which is illustrated in Figure 4.10. Notice that the dark nodes represent the GGs and the rest represent the GRs.

### 4.5.2 Mapping DGs to SFGs

Based upon the DGs obtained in Sec. 4.5.1, we will develop some SFGs for the matrix triangularization problem in this subsection.


Figure 4.10 A new DG for QR decomposition

### 4.5.2.1 LU Decomposition

Version 1. Referring to Figure 4.8, we choose the projection vector $d=\left[\begin{array}{ll}1 & 1\end{array} 1\right]^{T}$ and choose the processor basis $\boldsymbol{P}$ which is orthogonal to $\boldsymbol{d}$. The schedule is chosen to be the default one, i.e., $\boldsymbol{s}=\boldsymbol{d}$. The mappings are

$$
d^{T}=\left[\begin{array}{lll}
1 & 1 & 1
\end{array}\right], \quad P^{T}=\left[\begin{array}{lll}
1 & 0 & -1  \tag{4.21}\\
0 & 1 & -1
\end{array}\right]
$$

By using the mapping method discussed in Sec. 4.2, we can obtain the SFGs for the LU decomposition algorithm. The node mapping, the arc mapping and the I/O mapping for the given project vector and the processor basis can be given as follows.

$$
\text { Node mapping: } \quad\left[\begin{array}{ccc}
1 & 0 & -1  \tag{4.22}\\
0 & 1 & -1
\end{array}\right]\left[\begin{array}{l}
i \\
j \\
k
\end{array}\right]=\left[\begin{array}{l}
i-k \\
j-k
\end{array}\right]
$$

Arc mapping: $\quad\left[\begin{array}{ccc}1 & 1 & 1 \\ 1 & 0 & -1 \\ 0 & 1 & -1\end{array}\right]\left[\begin{array}{lll}1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1\end{array}\right]=\left[\begin{array}{ccc}1 & 1 & 1 \\ 1 & 0 & -1 \\ 0 & 1 & -1\end{array}\right]$

$$
\begin{array}{ll}
\text { I/O mapping: } & \text { Input: }\left[\begin{array}{ccc}
1 & 1 & 1 \\
1 & 0 & -1 \\
0 & 1 & -1
\end{array}\right]\left[\begin{array}{l}
i \\
j \\
1
\end{array}\right]=\left[\begin{array}{c}
i+j+1 \\
i-1 \\
j-1
\end{array}\right] \\
\text { Output: }\left[\begin{array}{ccc}
1 & 1 & 1 \\
1 & 0 & -1 \\
0 & 1 & -1
\end{array}\right]\left[\begin{array}{cc}
i & k \\
k & j \\
k & k
\end{array}\right]=\left[\begin{array}{cc}
i+2 k & j+2 k \\
i-k & 0 \\
0 & j-k
\end{array}\right] \tag{4.24}
\end{array}
$$

The SFG array derived from this projection is shown in Figure 4.11. The SFG is a hexagonal array with the pipelining period $\alpha=3$. The PEs in the first column perform division and the first row PEs only pass data, and the rest PEs perform multiply-and-add operation. Output matrices, $\boldsymbol{L}$ and $\boldsymbol{U}$ can be obtained from the first column PEs and first row PEs respectively.


Version 2. Now we choose another projection direction as $\boldsymbol{d}=\left[\begin{array}{lll}1 & 0 & 0\end{array}\right]^{T}$ and a corresponding orthogonal processor basis $\boldsymbol{P}$. The default schedule is again selected. The mappings can be described as follows:

$$
d^{T}=\left[\begin{array}{lll}
1 & 0 & 0
\end{array}\right], \quad P^{T}=\left[\begin{array}{lll}
0 & 1 & 0  \tag{4.25}\\
0 & 0 & 1
\end{array}\right]
$$

Similarly, we can derive the corresponding SFG as shown in Figure 4.12. Version 2 SFG is a
triangular array whose diagonal PEs perform divisions and the rest PEs perform multiply-and-add operations.

### 4.5.2.2 QR Decomposition

Version 1. We select a projection vector $\boldsymbol{d}=\left[\begin{array}{lll}0 & 1 & 1\end{array}\right]^{T}$ for the DG for QR decomposition in Figure 4.9. The mappings are then given by

$$
d^{T}=\left[\begin{array}{lll}
0 & 1 & 1
\end{array}\right], \quad P^{T}=\left[\begin{array}{ccc}
1 & 0 & 0  \tag{4.26}\\
0 & 1 & -1
\end{array}\right]
$$

The SFG array derived from this projection is shown in Figure 4.13. The PEs in the first column (dark ones) perform GGs and the rest PEs perform GRs. Output matrix $R$ can be obtained from the top of the SFG array. Notice that the bottom row PEs will receive two rows of input data as indicated by the QR decomposition algorithm given in Eq. (4.18).


Figure 4.13 Ver. I SFG for QR decomposition


Figure 4.14 Ver. 2 SFG for QR decomposition

Version 2. Now we choose another projection direction as $d=\left[\begin{array}{lll}1 & 0 & 0\end{array}\right]^{T}$ for the DG in Figure 4.9 and a corresponding orthogonal processor basis $\boldsymbol{P}$. The default schedule is again selected. The mappings can be described as follows:

$$
\boldsymbol{d}^{T}=\left[\begin{array}{lll}
1 & 0 & 0
\end{array}\right], \quad \boldsymbol{P}^{T}=\left[\begin{array}{lll}
0 & 1 & 0  \tag{4.27}\\
0 & 0 & 1
\end{array}\right]
$$

This obtained Version 2 SFG is a triangular array whose diagonal PEs perform GGs and the rest PEs perform GRs as shown in Figure 4.14.

Version 3. Now we apply the projection vector $\boldsymbol{d}=\left[\begin{array}{lll}1 & 0 & 0\end{array}\right]^{T}$ to the DGs in Figure 4.10. The mappings are same as those in Eq. (4.27). The obtained SFG is shown in Figure 4.15.

There are other possible projection directions; however, they will result that the PEs in the corresponding SFGs need perform different functions at different time. All the SFGs shown above do not need reprogramming (no change of processor functions) and this will simplify the processor array implementation.

### 4.5.3 Mapping SFGs Systolic Arrays

### 4.5.3.1 LU Decomposition

We can systolize the two SFGs for the LU decomposition corresponding to the projections in the directions of $d=\left[\begin{array}{lll}1 & 1 & 1\end{array}\right]^{T}$ and $d=\left[\begin{array}{lll}1 & 0 & 0\end{array}\right]^{T}$, as shown in Figures 4.11 and 4.12, by applying the cut-set retiming procedure.

Note that the SFG in Figure 4.11 is already systolized as each edge has at least one delay. Therefore, the corresponding systolic array can be derived by simply combing the delay and operation module. Figure 4.16 shows Version 1 systolic array for the LU decomposition.


Figure 4.15 Ver. 3 SFG for QR decomposition


Figure 4.16 Systolic array for LU decomposition (Version 1)

Version 1 systolic array for LU decomposition in Figure 4.16 is equivalent to the hexagonal systolic array proposed in [KungHT78], [Leiserson81] and [KungHT82a]. The detailed operation of this systolic array can be found in [Quinton91] where the snapshots of the systolic array are given and analysed.

Version 2 SFG for the LU decomposition in Figure 4.12 is required to apply retiming procedure in order to be systolized. We can apply the retiming rule 2 to the cut-sets as shown in Figure 4.17a to obtain the corresponding systolic array as shown in Figure 4.17b.

The localized SFG in Figure
4.17a has one delay assigned to each edge and thus represents a temporally localized network. According to the retiming rule 2, the inputs from different column of $A$ have to be adjusted by a certain number of delays before arriving at the array. By counting the cut sets involved in Figure 4.17a, it is clear that the first column of $A$ needs no extra delay, the second column needs one


Figure 4.17 Systolic Array for LU decomposition (Version 2): (a) cut-sets; (b) systolic array delay, the third two (because of the two cut-sets separating the third column input and the adjacent top-row processor) and so forth. Therefore, the input matrix $A$ will be skewed as shown in Figure 4.17b.

The systolic array in Figure 4.17b was first proposed by Gentleman and Kung in [Gentleman81] and it has found numerous applications. This systolic array uses the triangular array of processors connected orthogonally. The Occam (a parallel programming language) implementation of this systolic array is appended in this chapter [Wan90a]. A detailed performance analysis of the array can be found in [Quiton91] and [Wan90a]. The neighbouring pivot scheme to ensure a stable solution of LU
decomposition was introduced in [Gentleman81] and the Occam implementation adopt this new scheme although we will not give the details here.

Now we can compare the two versions of systolic arrays for the LU decomposition algorithm. Both arrays has fixed function PEs. The triangular array in Figure 4.17 is different from the one in [KungSY88a] which has function-changing (from division to multiply-and-add operation) PEs due to a slight different DGs is used as mentioned earlier in Sec. 4.5.1.1. Version 1 systolic array has two advantages: (1) it is easily adaptable to the band matrix LU decomposition problem; and (2) both the lower triangular matrix $L$ and the upper triangular matrix $U$ are available at the boundary PEs. On the other hand, Version 2 systolic array offers very attractive advantages: (1) There is no need for diagonal connection, thus simplifying the required hardware; (2) less PEs (approximately $50 \%$ of Version 1) are used while completing the same problem with equal computation time and thus it is twice as efficient as the version 1; and (3) Version 2 systolic array enjoys a pipelining period of 1, compared with a value of 3 for Version 1 .

### 4.5.3.2 QR Decomposition

From Version I SFG for QR decomposition in Figure 4.13, we can obtain a corresponding systolic array. The cut-sets for systolizing the SFG is shown in Figure 4.18. It is noticed that the input matrix $\boldsymbol{A}$ is required to be skewed. This systolic array can be further modified to have $M$ rows of PEs instead of ( $M-1$ ) rows by introducing a new row to pass the last row of the input matrix.


Figure 4.18 Systolic Array for QR decomposition (Version 1): (a) cut-sets; (b) systolic array

Versions 2 and 3 SFGs for QR decomposition in Figures 4.14 and 4.15 respectively lead to another two systolic arrays for QR decomposition. The procedure to systolize the SFGs is very similar to the procedure for Version 2 systolic array for LU decomposition. The obtained systolic arrays are shown in Figures 4.19 and Figure 4.20 while the sytolization procedure is omitted.


Figure 4.19 Systolic array for QR decomposition (Version 2)


Figure 4.20 Systolic array for QR decomposition (Version 3)

The systolic array in Figure 4.20 was first proposed by Gentleman and Kung in [Gentleman81]. The systolic array uses the triangular array of processors connected orthogonally. The Occam implementation of this systolic array is also appended in this chapter [Wan90a]. Observe that circle PEs in Figures 4.18-4.20 are more complex than square PEs. The circle PEs perform Givens generation operations which compute square roots as will as reciprocals, whereas the square PEs perform only additions and multiplications. Since all PEs in the systolic array must operate at the same throughput rate in order to acquire the maximum efficiency, the circular PEs could form a bottleneck for the overall performance. The square-root-free Givens transformation, originally introduced by

Gentleman in [Gentleman73], reduces considerably the work required and removes the need for the calculation of any square root in the QR decomposition algorithm described above. The triangular version of square-root-free systolic array for QR decomposition has a similar structure to the version 3 systolic array in Figure 4.20 except that the former has a main diagonal connection. The details on designing the square-root-free systolic array can be referred to [McWhirter86], [McWhirter89] and [Wan90a]. The Occam program for this systolic array is also appended in this chapter.

Now we briefly compare the three versions of systolic arrays for QR decomposition. Version 1 systolic array in Figure 4.18 has a rectangular geometry. One advantage of this array is that the resultant triangular matrix can be conveniently obtained at the boundary PEs of the systolic array. This array, however, may be very inefficient when $M$ is much greater than $N$. The versions 2 and 3 systolic array for QR decomposition enjoy high efficiency as less PEs are used (approximately $50 \%$ of Version 1 when $M=N$ and more saving when $M \because N$ ). Versions 2 and 3 arrays have a pipelining period of 1 , compared with a value of 2 for the version 1. Versions 2 and 3 arrays have similar performance except that the input data to the array is in different orders. Version 3 systolic array may be preferable for many practical applications which make the input data available from the first row of the matrix to the last row of the matrix.

### 4.6 Discussions on the SFG Design Methodology

### 4.6.1 Optimization of the Systolic Array Design

There are many factors in determining the optimization criteria for the design of systolic arrays. The final choice of optimality criteria will be application dependent. In Sec. 3.5, we have discussed the performance measures of a systolic array. Among those performance measures are the computation time, pipelining rate, block pipelining period, array size, and I/O channels. Depending upon different applications, different optimization criteria may be chosen. For example, the computation time is perhaps a more important criteria than others when a single problem instance with finite input data is required to solve. On the other hand, the pipelining rate may play a more critical role when a single problem with indefinite input data is concerned (such problems often occur in many DSP applications). In the case that many problem instances are to be processed by the same systolic array, then we should focus on the block pipelining rate.

In order to obtain an optimal systolic array design under certain criteria, cares should be taken in the three design stages: (a) DG design stage; (b) SFG design stage; and (c) systolization stage.

### 4.6.1.1 Optimization in DG Design Stage

The DG design plays a very important role in the systolic array design. When designing a DG for a given problem, the following techniques might be potentially useful.

Localizing the Broadcast Data. As a non-localized DG is likely to result in an array with global interconnection, it is desired and preferable to obtain a localised DG although such global arcs can be avoided by using a proper projection direction in the mapping schemes. The broadcast data which requires global communication can be replaced by the transmittent data which is propagate via local arcs. Figure 4.2 gives an example of localization of the broadcast data.


Figure 4.21 Two DGs for band-matrix and vector multiplication with reversed vertical arcs and their corresponding SFGs

Reversible Arcs for Associative Operations. Note that if the operation used in the recursion is associative, then the direction of the arcs may be reversible. This can be illustrated by a simple example for band-matrix and vector multiplication. A DG for the band-matrix and vector
multiplication is shown in Figure 4.21a. A diagonal projection direction can be adopted to derive an array of size $w$ (where $w$ is the bandwidth of the matrix). This projection will result in an SFG array of size $w$ which is, in most applications, smaller than the array size $N$. The schedule selected is also given in the same figure. Note that the DG in Figure 4.2la can be modified since the operation for the recursion along the vertical direction is addition (which is associative). The vertical arcs in Figure 4.2 la can be reversed to form a new DG as shown in Figure 4.2 lb . The benefit due to this modification is that the new DG can be mapped into a systolic array with a pipelining rate of 1 by choosing an appropriate schedule vector.

### 4.6.1.2 Optimization in SFG Design Stage

Project Direction. The array size and I/O structures of the resulting SFG of a given DG depend on the projector vector. Therefore, we can select a suitable project direction in order to minimize the hardware requirement. For example, the systolic arrays for QR decomposition in Figures 4.19 and 4.20 are obtained by selecting a project direction to minimize the number of PEs required. Another example is given in Figure 4.21.

Effect on $\alpha$ and $\beta$ by the Projection Direction. In many instances, the projection vector chosen have significant effects on the optimal pipelining rate $\alpha$ and the optimal block pipelining rate $\beta$ obtainable.

### 4.6.1.3 Optimization in Systolization Stage

Minimal Pipelining Period Design. The pipelining period ( $\alpha$ ) is often a good optimality criterion for the case that the input data is either very long or indefinite, as frequently incurred in signal processing applications. Given a systolic schedule $\boldsymbol{s}$ and a projection direction $\boldsymbol{d}$ of a DG, with both $\boldsymbol{s}$ and $\boldsymbol{d}$ in coprime form, the pipelining period $\alpha$ can be calculated as

$$
\begin{equation*}
\alpha=s^{T} d \tag{4.28}
\end{equation*}
$$

Therefore, an integer programming for minimizing $\alpha$ can be formulated as follows

$$
\begin{gather*}
\min s^{T} d  \tag{4.29}\\
\text { subject to } s^{T} d>0 \text { and } s^{T} e>0 \text { for any edge }
\end{gather*}
$$

In Figure 4.2 lb , for instance, we choose an optimal schedule vector to obtain a systolic array with pipelining rate of 1 .

Multirate Systolic Design. In a systolic array, the processing or communication times of different operations may vary a great deal. As a result, the global clock period has to be the maximum of these operation times, plus some safety margin. This is clearly undesirable for achieving maximum array throughput. One solution is to allow different operations in the array take different time periods by use of a finer clocking period. This type of systolic array is called a multirate systolic array. A multirate systolic array is a generalized systolic array, allowing different operations to consume different time units.

### 4.6.2 Non-Linear Assignment

By linear projection, we assign multiple nodes along a straight line in a DG into a single PE in its corresponding SFG. In contrast, non-linear assignment means that multiple nodes not necessarily along a straight line are assigned to a PE. The SFG representation can make the non-linear assignment easier (than the RIA representation discussed in Chap. 3, for example).

By using a non-linear assignment, a more flexible design can be devised and a broad range of algorithms can be covered. One such example is an algorithm represented by cascaded DG, in which the algorithm is comprised of a group of DGs connected in cascade. A simple example is shown in Figure 4.22, where three DGs for matrix-vector multiplication are cascaded to compute $\boldsymbol{e}=\boldsymbol{A B C I}$. For this cascaded DG, all the DGs involved are same. In this case, the nonlinear mapping


Figure 4.22 Non-linear assignment of cascaded DG for matrix-vector multiplication
allows handling of three DGs together in one piece rather than separately. This will ease the data reformatting and increase the pipelining rate.

### 4.7 Summary

The systolic array synthesis methodology using SFG representations can be divided into three stages. Firstly, a dependence graph (DG) is designed for a given problem in order to obtain a concise representation of the data structure of the algorithms. The DG provides a useful expression of the algorithm and permits certain structured modifications on the graph, whereas the mathematical formulation might be very abstract to be modified. In the second stage, the obtained DG (with a certain degree of regularity) is mapped into an SFG array by a simple projection method. In the last stage, a cut-set retiming procedure is applied to the SFG to derive a systolic array.

The systematic transformation from DG to systolic arrays greatly simplifies the understanding of systolic design and it also provides a means to verify the design. The graph representations used in all the 3 stages of the design make the design methodology very instructive to show the systolic principle, and they also provide clues to improve the systolic design in the three different stages.

As to optimal systolic design, it depends on the performance criteria. Obviously, the selection of a particular DG for the algorithm and the types of projection and scheduling can greatly affect the performance of the resulting systolic array.

By applying the design methodology, we also design different systolic arrays for the matrix triangularization problem, namely LU decomposition and QR decomposition. The matrix triangularization plays a very important role in many applications and we will introduce some of their applications in the later chapters.

The SFG design methodology has been proved to be a powerful tool for the systolic array synthesis. Many systolic array have been derived by using this method. Some examples are lattice filter, matrix multiplication, banded-matrix multiplication, LU decomposition, QR decomposition, convolution, transitive closure, dynamic programming, artificial neural network, and so on.

### 4.8 Occam Programming Lists

### 4.8.1 Systolic Algorithm for LU Decomposition

```
- Systolic algorithm for LU decomposition
- by C.R. Wan
--
- File Name: "A:LU"
```

- protocols

PROTOCOL horizontal IS REAL32; BOOL:
PROTOCOL vertical IS REAL32; BOOL:
-- user call
\#USE snglmath
\#USE mathhdr
\#USE userio
\#USE mathvals
-- declarations

VAL INT N IS 4:
VAL INT NN IS N*N:
VAL INT whole.elements IS $\left((\mathrm{N}+1)^{*} \mathrm{~N}\right) / 2$ :
VAL INT square.elements IS whole.elements-N:
VAL INT circle.elements IS $N$ :
VAL INT iteration.times IS N :
[iteration.times] [N] REAL32 X:
[iteration.times] [ N ] BOOL flag:
[whole.elements] CHAN OF vertical pipe.vertical:
[whole.elements] CHAN OF horizontal pipe.horizontal:
[whole.elements] CHAN OF REAL32 test:
[whole.elements] REAL 32 temp:
[N] REAL 32 input, output:
[N] CHAN OF vertical interface.vertical:
[N] CHAN OF REAL32 interface.horizontal:
[ N$][\mathrm{N}]$ INT cell.row.col:
[whole.elements] INT row.cell:
[whole.elements] INT col.cell:
PROC square.element ( CHAN OF horizontal west, east, CHAN OF vertical north, south, CHAN OF REAL 32 test.out)

REAL32 x, x.in, x.out, m.in, m.out :
BOOL v.in, v.out, flag:
SEQ
PAR
west ? m.in; v.in
north ? x.in; flag
IF
flag
$\mathrm{x}:=0.0$ (REAL32)
TRUE
SKIP
IF
v.in

SEQ
x.out $:=x+\left(m\right.$. in $^{*} x$.in $)$
$x:=x$.in
TRUE
x.out : $=\mathrm{x} . \mathrm{in}+\left(\mathrm{m} . \mathrm{in}^{*} \mathrm{x}\right)$

PAR
east ! m.in; v.in south ! x.out; flag

```
    test.out ! x
```

PROC circle.element ( CHAN OF horizontal east,
CHAN OF vertical north,
CHAN OF REAL 32 test.out )

REAL32 x, x.in, m.out:
BOOL v.out, flag:

## SEQ

north ? x.in ; flag
IF
flag
$\mathrm{x}:=0.0$ (REAL32)
TRUE
SKIP
IF
ABS (x.in) $>=\mathrm{ABS}(\mathrm{x})$
SEQ
v.out := TRUE

IF
$x$.in $>0.0$ (REAL32)
m.out $:=-(x / x$. in $)$

TRUE
m.out $:=0.0$ (REAL32)
$x:=x . i n$
TRUE
SEQ
v.out:= FALSE
m.out := -(x.in/x)
east ! m.out; v.out
test.out ! x
PROC initial ()
SEQ $j=0$ FOR N
SEQ $\mathrm{i}=0$ FOR iteration.times
$\mathrm{X}[\mathrm{i}][\mathrm{j}]:=0.0$ (REAL32)
PROC input.interface ( []REAL32 input,
[]BOOL flag.input,
[]CHAN OF vertical pipe,vertical)
PAR $\mathrm{i}=0$ FOR N
pipe.vertical[i] ! input[i] ; flag.input[i]
PROC output.interface ( CHAN OF horizontal right.most, REAL32 variable )
BOOL m:
SEQ
right.most ? variable; m
PROC display.data([][]REAL32 X, VAL INT N1,N2,
VAL []BYTE title)
-display the data matrix X[i.j] on screen
$-\mathrm{N} 1, \mathrm{~N} 2$ give the scope of displaying
SEQ
newline(screen)
write.full.string(screen,title)
newline(screen)
SEQ $\mathrm{i}=0$ FOR N1
SEQ
SEQ $\mathrm{j}=0$ FOR N2
write.real32(screen, X[ij[j], 6, 2 )
newline(screen)
PROC read.data()
SEQ

| $\mathrm{X}[0][0]=1.0$ | (RE |
| :---: | :---: |
| $\mathrm{X}[0][1]=2.0$ | (REAL32) |
| $\mathrm{X}[0][2]:=3.0$ | (REAL32) |
| $X[0][3]=-2.0$ | (REAL32) |
| $\mathrm{X}[1][0]=2.0$ | (REAL32) |
| $\mathrm{X}[1][1]=4.0$ | (REAL32) |
| $\mathrm{X}[1][2]=1.0$ | (REAL32) |
| $\mathrm{X}[1][3]:=0.0$ | (REAL32) |
| $\mathrm{X}[2][0]=3.0$ | (REAL32) |
| $\mathrm{X}[2][1]=3.0$ | (REAL32) |
| $\mathrm{X}[2][2]:=2.0$ | (REAL32) |
| $\mathrm{X}[2][3]:=5.0$ | (REAL32) |
| X[3][0]:=-1.0 | (REAL32) |
| $\mathrm{X}[3][1]:=0.0$ | (REAL32) |
| $\mathrm{X}[3][2]=2.0$ | (REAL32) |
| $\mathrm{X}[3][3]=1.0$ | (REAL32) |

SEQ $\mathrm{i}=0$ FOR iteration.times
SEQj=0 FOR N
flag[i][j]:=FALSE
SEQ $=0$ FOR N
flag[0][j]:=TRUE
PROC row.col.cell ( []INT row, col, VAL INT N)
-calculate the row and col number of each cell

VAL N21 IS ( $\mathrm{N}^{*} 2$ )+1:
SEQ $i=0$ FOR whole.elements
VAL INT delt IS INT ROUND( SQRT( REAL32 ROUND(( N21*N21)-(8*i)) )): SEQ
row[i]:=(N21-dett)/2
$\operatorname{col}[\mathrm{i}]:=(\mathrm{i}+\mathrm{row}[\mathrm{i}])-((\mathrm{N} 21-$ row $[\mathrm{i}]) *$ row $[\mathrm{i}]) / 2)$
PROC cell.row.and.col ([][]INT cell.row.col, VAL INT N)
-calculate the cell number of (row,col)th cell
SEQ row=0 FOR N
SEQ col=0 FOR N
cell.row.col[row][col]: $=\left(\left(((\mathrm{N}+1)+(\mathrm{N}-\text { row }))^{*}\right.\right.$ row $\left.) / 2\right)$

+ (col-row)
.
- main process starting hear

INT any:
SEQ
cell.row.and.col ( cell.row.col, N)
VAL string IS "cell No. of ( $\mathrm{i}, \mathrm{j}$ )":
SEQ
write.full.string( screen, string)
newline(screen)
SEQ $i=0$ FOR $N$
SEQ
SEQ $\mathrm{j}=0$ FOR i
write.full.string(screen," ")
$S E Q j=i$ FOR ( $\mathrm{N}-\mathrm{i}$ )
write.int(screen, cell.row.col[i][j],9)
newline(screen)
row.col.cell ( row.cell, col.cell, N)
SEQ
write.full.string (screen, "row and col number of ith cell") newline(screen)
SEQ $i=0$ FOR whole.elements
SEQ
write.int(screen, row.cell[i],9)
write.int(screen, col.cell[i],9) newline(screen)
initial ()
read.data ()
VAL string IS "Input data $\mathrm{X}(\mathrm{i}, \mathrm{j})$ ":
display.data( $\mathrm{X}, \mathrm{N}, \mathrm{N}$, string )

```
PAR
SEQ iteration=0 FOR iteration.times
    []REAL32 input IS X[iteration]:
    [BOOL flag.input IS flag[iteration]:
[]CHAN OF vertical interface.vertical IS [pipe.vertical
        FROM O FOR N]:
    SEQ
    newline(screen)
    write.full.string(screen,"Iteration time is " )
    write.int(screen,iteration, 8)
    newline(screen)
    write.full.string(screen, "Input from interface:")
    newline(screen)
    write.int(screen, iteration, 4)
    SEQ ii=0 FOR N
    write.real32(screen, input[ii], 6, 2)
    newline(screen)
    PAR
    input.interface ( input, flag.input, interface.vertical )
    PAR i=0 FOR whole.elements
        IF
        VAL INT row IS row.cell[i]:
        VAL INT col IS col.cell[i]:
        row=col
            CHAN OF vertical up IS pipe.vertical[ i ] :
        CHAN OF horizontal right IS pipe.horizontal[ i ]:
        test.chan IS test[i]:
        circle.element(right, up, test.chan )
        TRUE
        VAL INT row IS row.cell[i]:
        VAL INT col IS col.cell[i]:
        VAL INT up.number IS cell.row.col[row][col]:
        VAL INT right.number IS up.number:
        VAL INT down.number IS
                            cell.row.col[row+1][col] :
        VAL INT left.number IS up.number-1:
        CHAN OF vertical up IS pipe.vertical[up.number]:
        CHAN OF vertical down IS
                            pipe.vertical[down.number]:
        CHAN OF horizontal right IS
                            pipe.horizontal[right.number]:
        CHAN OF horizontal left IS
                            pipe.horizonta[[left.number]:
        test.chan IS test[i]:
        square.element( left, right, up, down, test.chan )
    PAR i=0 FOR N
        VAL INT most.number IS cell.row.col[i][N-1] :
        CHAN OF horizontal right.most IS
                            pipe.horizontal[most.number]:
        output.interface (right.most, output[i])
    PAR i=0 FOR whole.elements
        test.chan IS test[i]:
        test.number IS temp[i]
        test.chan ? test.number
    SEQ
        write.full.string(screen, "The middle result is:")
        newline(screen)
        SEQ i=0 FOR whole.elements
        write.real32(screen, temp[i], 6, 2)
        newline(screen)
    SEQ
        VAL string IS "The result is:":
        write.full.string(screen,string)
        newline(screen)
        SEQ i=0 FOR N
        write.real32(screen, output[i], 6,2)
        newline(screen)
keyboard ? any
```


### 4.8.2 Systolic Algorithm for QR decomposition

```
- Systolic algorithm for QR decomposition
- by C.R. Wan
-- File Name: "A:QR"
- Standard Givens Rotation Method
-- protocols
PROTOCOL horizontal IS REAL32; REAL32:
PROTOCOL vertical IS REAL32; BOOL:
-- user.call
\#USE snglmath
\#USE mathhdr
\#USE userio
\#USE mathvals
-- declarations
VAL INT N IS 4:
VAL INT NN IS N*N:
VAL INT whole.elements IS \(\left((\mathrm{N}+1)^{*} \mathrm{~N}\right) / 2\) :
VAL INT square.elements IS whole.elements-N:
VAL INT circle.elements IS N :
VAL INT iteration.times IS N:
[iteration.times] [N] REAL32 X:
[iteration.times] [N] BOOL flag:
```

[whole.elements] CHAN OF vertical pipe.vertical:
[whole.elements] CHAN OF horizontal pipe.horizontal:
[whole.elements] CHAN OF REAL32 test:
[whole.elements] REAL 32 temp:
[N] REAL32 input, output.c, output.s:
[N] CHAN OF vertical interface.vertical:
[N] CHAN OF REAL 32 interface.horizontal:
[ N$][\mathrm{N}]$ INT cell.row.col:
[whole.elements] INT row.cell:
[whole.elements] INT col.cell:
PROC square.element ( CHAN OF horizontal west, east, CHAN OF vertical north, south, CHAN OF REAL 32 test.out)

```
REAL32 x, x.in, x.out, c.in, s.in :
BOOL v.in, v.out, flag:
SEQ
    PAR
    west ? c.in; s.in
    north ? xin; flag
    IF
        flag
        x:=0.0 (REAL32)
        TRUE
            SKIP
    x.out:=(c.in*x.in)-(s.in*x )
    x := (c.in * x ) +(s.in* x.in)
    PAR
        east!c.in; s.in
        south ! x.out; flag
        test.out ! x
```

PROC circle.element ( CHAN OF horizontal east,
CHAN OF vertical north,
CHAN OF REAL 32 test.out )
REAL32 x , x in, c.out, s.out, r:
BOOL flag:

SEQ
north ? x.in; flag

IF

## flag

$\mathrm{x}=0.0$ (REAL32)
TRUE
SKIP
IF
$x . \operatorname{in}=0.0$ (REAL32)
SEQ
c.out: $=1.0$ (REAL32)
s.out: $=0.0$ (REAL32)

TRUE SEQ
$r:=\operatorname{SQRT}\left(\left(x^{*} x\right)+\left(x . \operatorname{in}^{*} x . i n\right)\right)$
c.out : $=\mathrm{x} / \mathrm{r}$
s.out $:=\mathrm{x} . \mathrm{in} / \mathrm{r}$
x : $=\mathrm{r}$
east ! c.out; s.out
test.out ! x
:
PROC initial ()
SEQ $\mathrm{j}=0$ FOR N
SEQ $\mathrm{i}=0$ FOR iteration.times
$\mathrm{X}[\mathrm{i}][\mathrm{j}]=0.0$ (REAL32)
PROC input.interface ( []REAL32 input, []BOOL flag.input, []CHAN OF vertical pipe.vertical)
PAR $\mathrm{i}=0$ FOR N
pipe.vertical[i] ! input[i] ; flag.input[i]
PROC output.interface ( CHAN OF horizontal right.most, REAL32 variable 1, variable2 )
BOOL m:
SEQ
right.most ? variablel; variable2
PROC display.data([J]]REAL32 X, VAL INT N1,N2, VAL []BYTE title)
-display the data matrix X[i.j] on screen

- N1, N2 give the scope of displaying

SEQ
newline(screen)
write.full.string(screen,title)
newline(screen)
SEQ $i=0$ FOR N1
SEQ
SEQ $\mathrm{j}=0$ FOR N 2
write.real32(screen, $\mathrm{X}[\mathrm{i}] \mathrm{Ej}], 6,2$ )
newline(screen)
PROC read.data()
SEQ

| $X[0][0]:=1.0$ | (REAL32) |
| :--- | :--- |
| $X[0][1]:=2.0$ | (REAL32) |
| $X[0][2]:=3.0$ | (REAL32) |
| $X[0][3]:=-2.0$ | (REAL32) |
| $X[1][0]:=2.0$ | (REAL32) |
| $X[1][1]:=4.0$ | (REAL32) |
| $X[1][2]:=1.0$ | (REAL32) |
| $X[1][3]:=0.0$ | (REAL32) |
| $X[2][0]:=3.0$ | (REAL32) |
| $X[2][1]:=3.0$ | (REAL32) |
| $X[2][2]:=2.0$ | (REAL32) |
| $X[2][3]:=5.0$ | (REAL32) |
| $X[3][0]:=-1.0$ | (REAL32) |
| $X[3][1]:=0.0$ | (REAL32) |
| $X[3][2]:=2.0$ | (REAL32) |
| $X[3][3]:=1.0$ | (REAL32) |

```
    SEQ i=0 FOR iteration.times
    SEQ j=0 FOR N
        flag[i][j]:=FALSE
    SEQj=0 FOR N
    flag[0][j]:=TRUE
PROC row.col.cell ( [JINT row, col, VAL INT N)
-calculate the row and col number of each cell
VAL N21 IS (N*2)+1:
SEQ i=0 FOR whole.elements
    VAL INT delt IS INT ROUND(SQRT(
        REAL32 ROUND(( N21*N21)(8*i)) )):
    SEQ
    row[i]:= (N21-delt)/2
    col[i]:=(i+row[i])-(((N21-row[i])*row[i])/2)
:
PROC cell.row.and.col ( [][]INT cell.row.col, VAL INT N)
--calculate the cell number of (row,col)th cell
SEQ row=0 FOR N
    SEQ col=0 FOR N
    cell.row.col[row][col]:=((((N+1)+(N-row))*row//2)
                                    +(col-row)
-- main process starting hear
INT any:
SEQ
cell.row.and.col ( cell.row.col, N)
VAL string IS "cell No. of (i,j)":
SEQ
    write.full.string( screen, string)
    newline(screen)
    SEQ i=0 FOR N
    SEQ
        SEQ j=0 FOR i
            write.full.string(screen," ")
        SEQ j=i FOR (N-i)
            write.int(screen, cell.row.col[i][j],9)
        newline(screen)
row.col.cell ( row.cell, col.cell, N)
SEQ
    write.full.string (screen, "row and col number of ith cell")
    newline(screen)
    SEQ i=0 FOR whole.elements
        SEQ
        write.int(screen, row.cell[i],9)
        write.int(screen, col.cell[i],9)
        newline(screen)
initial ()
read.data ()
VAL string IS "Input data X(i,j)":
display.data( X, N,N, string )
PAR
```

SEQ iteration=0 FOR iteration.times
[]REAL32 input IS X[iteration]:
[]BOOL flag.input IS flag[iteration]:
[]CHAN OF vertical interface.vertical IS
[pipe.vertical FROM 0 FOR N]:
SEQ
newline(screen)
write.full.string(screen,"Iteration time is ")
write.int(screen,iteration, 8)
newline(screen)
write.full.string(screen, "Input from interface:")
newline(screen)
write.int(screen, iteration, 4)
SEQ ii=0 FOR N
write.real32(screen, input[ii], 6, 2)
newline(screen)
PAR
input.interface ( input, flag.input, interface.vertical)
PAR $\mathrm{i}=0$ FOR whole.elements
IF
VAL INT row IS row.cell[i]:
VAL INT col IS col.cell[i]: row=col

CHAN OF vertical up IS pipe.vertical $[i]$ : CHAN OF horizontal right IS pipe.horizontal[ $i$ ]: test.chan IS test[i]:
circle.element( right, up, test.chan ) TRUE

VAL INT row IS row.cell[i]:
VAL INT col IS col.cell[i]:
VAL INT up.number IS cell.row.col[row][col] :
VAL INT right.number IS up.number:
VAL INT down.number IS
cell.row.col[row+1][col] :
VAL INT left.number IS up.number-1:
CHAN OF vertical up IS pipe.vertical[up.number]:
CHAN OF vertical down IS
pipe.vertical[down.number]:
CHAN OF horizontal right IS
pipe.horizontal[right.number]:
CHAN OF horizontal left IS
pipe.horizontal[left.number]:
test.chan IS test[i]:
square.element( left, right, up, down, test.chan )
PAR $\mathrm{i}=0$ FOR N
VAL INT most.number IS cell.row.col[i][N-1] :
CHAN OF horizontal right.most IS
pipe.horizontal[most.number]:
output.interface ( right.most, output.c [i],
output.s[i])
PAR $\mathrm{i}=0$ FOR whole.elements
test.chan IS test[i]:
test.number IS temp[i]:
test.chan ? test.number
SEQ
write.full.string(screen,"The middle result is:")
newline(screen)
SEQ $i=0$ FOR whole.elements
write.real32(screen, temp [i], 6, 2)
newline(screen)

## SEQ

VAL string IS "The result is:":
write.full.string(screen,string)
newline(screen)
SEQ $i=0$ FOR N
write.real32(screen, output.c[i], 6,2)
newline(screen)
SEQ $\mathrm{i}=0$ FOR N
write.real32(screen, output.s[i], 6,2)
newline(screen)
keyboard ? any

### 4.8.3 Systolic Algorithm for Square-Root-Free QR decomposition

[^0]
## by C.R. Wan

[^1]-- protocols
PROTOCOL horizontal IS REAL32; REAL32; REAL32; REAL32:
PROTOCOL vertical IS REAL 32; BOOL:
PROTOCOL diagonal IS REAL32:

- call user
\#USE snglmath
\#USE mathhdr
\#USE userio
\#USE mathvals
- declarations

VAL INT N IS 4:
VAL INT NN IS $\mathrm{N}^{*} \mathrm{~N}$ :
VAL INT whole.elements IS $\left((\mathrm{N}+1)^{*} \mathrm{~N}\right) / 2$ :
VAL INT square.elements IS whole.elements-N:
VAL INT circle.elements IS $N$ :
VAL INT iteration.times IS N:
[iteration.times] [ N ] REAL32 X:
[iteration.times] REAL 32 diagonal.input:
[iteration.times] [N] BOOL flag:
[whole.elements] CHAN OF vertical pipe.vertical: [whole.elements] CHAN OF horizontal pipe.horizontal: [circle.elements +1 ] CHAN OF diagonal pipe.diagonal:
[whole.elements] CHAN OF REAL32 test:
[whole.elements] REAL 32 temp:
[N] REAL32 input, output.c, output.s:
[ N ] CHAN OF vertical interface.vertical:
[N] CHAN OF REAL32 interface.horizontal:
[ N ] N$]$ INT cell.row.col:
[whole.elements] INT row.cell:
[whole.elements] INT col.cell:
REAL32 nothing1, nothing2:
PROC square.element ( CHAN OF horizontal west, east, CHAN OF vertical north, south, CHAN OF REAL32 test.out)

REAL32 x .in, x .out, $\mathrm{z}, \mathrm{c}, \mathrm{s}, \mathrm{r}, \mathrm{dl}$ :
BOOL flag:
SEQ
PAR
west ? c; s; di; z
north ? x.in; flag
IF
flag
$\mathrm{r}:=0.0($ REAL 32$)$
TRUE
SKIP
x .out: $=\mathrm{x} . \operatorname{in}-\left(\mathrm{z}^{*} \mathrm{r}\right)$
r $:=\left(s^{*} x . \operatorname{in}\right)+\left(c^{*} r\right)$
PAR
east ! c; s; dl; z
south ! x.out; flag
test.out ! $\mathbf{r}$.
PROC circle.element ( CHAN OF horizontal east, CHAN OF vertical north, CHAN OF diagonal northwest, southeast, CHAN OF REAL32 test.out )
REAL32 d1, d, delt.in, x.in, c, s, z, delt.out:
BOOL flag:
SEQ
PAR
north ? x.in; flag
northwest ? delt.in

## flag

$\mathrm{d}:=0.0($ REAL32 $)$
TRUE
SKIP
$\mathrm{dl}:=\mathrm{d}+($ dett.in * (x.in * $x . i n))$
IF
$(x . \operatorname{in}=0.0($ REAL32 $)) O R($ delt.in $=0.0($ REAL32 $))$
SEQ
$\mathrm{c}:=1.0$ (REAL32)
$\mathrm{s}:=0.0$ (REAL32)
TRUE
SEQ
$\mathrm{c}:=\mathrm{d} / \mathrm{dl}$
$\mathrm{s}:=$ delt.in ${ }^{*}(\mathrm{x} . \mathrm{in} / \mathrm{dl})$
d:= dl
delt.out $:=c^{*}$ delt.in
z :=x. in
PAR
east ! $\mathrm{c} ; \mathrm{s} ; \mathrm{dl} ; \mathrm{z}$
southeast ! delt.out
test.out ! d
:

PROC initial ()
SEQ $j=0$ FOR $N$
SEQ $\mathrm{i}=0$ FOR iteration.times
$\mathrm{X}[\mathrm{i}][\mathrm{j}]:=0.0$ (REAL32)
:

PROC input.interface ( []REAI32 input, []BOOL flag.input,
[]CHAN OF vertical pipe.vertical)
PAR $i=0$ FOR $N$
pipe.vertical[i] ! input[i] ; flag.input[i]
:
PROC output.interface ( CHAN OF horizontal right.most,
REAL32 variable 1, variable2, variable3, variable4 )
right.most ? variablel; variable2; variable3; variable4
:
PROC display.data([][]REAL32 X, VAL INT N1,N2,
VAL []BYTE title)
-display the data matrix X[i.j] on screen
$--N 1, N 2$ give the scope of displaying
SEQ
newline(screen)
write.full.string(screen,title)
newline(screen)
SEQ $\mathrm{i}=0$ FOR N1
SEQ
SEQ $\mathbf{j = 0}$ FOR N2
write.real32(screen, X[i][j], 6, 2 )
newline(screen)
:
PROC read.data()
SEQ

| $X[0][0]:=1.0$ | (REAL32) |
| :--- | :--- |
| $X[0][1]:=2.0$ | (REAL32) |
| $X[0][2]:=3.0$ | (REAL32) |
| $X[0][3]:=-2.0$ | (REAL32) |
|  |  |
| $X[1][0]:=2.0$ | (REAL32) |
| $X[1][1]:=4.0$ | (REAL32) |
| $X[1][2]:=1.0$ | (REAL32) |
| $X[1][3]:=0.0$ | (REAL32) |
| $X[2][0]:=3.0$ | (REAL32) |
| $X[2][1]:=3.0$ | (REAL32) |
| $X[2][2]:=2.0$ | (REAL32) |
| $X[2][3]:=5.0$ | (REAL32) |
| $X[3][0]:=-1.0$ | (REAL32) |
| $X[3]\{1]:=0.0$ | (REAL32) |

$\begin{array}{ll}\mathrm{X}[3][2]:=2.0 & \text { (REAL32) } \\ \mathrm{X}[3][3]:=1.0 & \text { (REAL32) }\end{array}$
SEQ $i=0$ FOR iteration.times
SEQ $\mathrm{j}=0$ FOR N
flag[i][i]:=FALSE
SEQ $\mathrm{j}=0 \mathrm{FOR} N$
flag[0][j]:=TRUE
diagonal.input[0]:=1.0 (REAL32)
SEQ $i=0$ FOR iteration.times
diagonal.input[i]:= 1.0 (REAL32)

PROC row.col.cell ( []INT row, col, VAL INT N)
--calculate the row and col number of each cell
VAL N21 IS ( $\mathrm{N}^{*}$ 2)+1:
SEQ $i=0$ FOR whole.elements
VAL INT delt IS INT ROUND( SQRT( REAL32 ROUND((N21*N21)-(8*i)) )):
SEQ
row $[\mathrm{i}]:=(\mathrm{N} 21-\mathrm{delt}) / 2$
$\operatorname{col}[\mathrm{i}]:=(\mathrm{i}+$ row $[\mathrm{i}])-\left((\mathrm{N} 21-\text { row }[\mathrm{i}])^{*}\right.$ row $\left.\left.[\mathrm{i}]\right) / 2\right)$

PROC cell. row.and.col ( [][]INT cell.row.col, VAL INT N)
--calculate the cell number of (row,col)th cell
SEQ row=0 FOR N
SEQ col=0 FOR N
cell.row.col[row][col]: $=\left(\left(((\mathrm{N}+1)+(\mathrm{N} \text {-row }))^{*}\right.\right.$ row $\left.) / 2\right)$ +(col-row)

- main process starting hear

INT any:
SEQ
cell.row.and.col ( cell.row.col, N)
VAL string IS "cell No. of $(i, j)$ ":
SEQ
write.full.string( screen, string)
newline(screen)
SEQ $\mathrm{i}=0$ FOR N
SEQ
SEQ $\mathrm{j}=0$ FOR i
write.full.string(screen," ")
SEQ $=\mathrm{i}$ FOR ( $\mathrm{N}-\mathrm{i}$ )
write.int(screen, cell.row.col[i][j],9)
newline(screen)
row.col.cell ( row.cell, col.cell, N)
SEQ
write.full.string (screen, "row and col number of ith cell") newline(screen)
SEQ $\mathrm{i}=0$ FOR whole.elements SEQ
write.int(screen, row.cell[i],9)
write.int(screen, col.cell[i],9)
newline(screen)
initial ()
read.data ()
VAL string IS "Input data X(i,j)":
display.data( $X, N, N$, string )
PAR
SEQ iteration=0 FOR iteration.times
[]REAL32 input IS X[iteration]:
[BOOL flag input IS flag[iteration]:
[]CHAN OF vertical interface.vertical IS [pipe.vertical
FROM 0 FOR N]:

## SEQ

newline(screen)
write.full.string(screen,"Iteration time is ")
write.int(screen,iteration, 8) newline(screen)
write.full.string(screen, "Input from interface:")
newline(screen)
write.int(screen, iteration, 4)
SEQ $\mathrm{ij}=0$ FOR N
write.real32(screen, input[ii], 6, 2)
newline(screen)
PAR
input.interface (input, flag.input, interface.vertical )
corner.nw IS pipe.diagonal[0] :
input.nw IS diagonal.input[iteration]:
corner.nw ! input.nw
PAR $\mathrm{i}=0$ FOR whole.elements
VAL INT row IS row.cell[i]:
VAL INT col IS col.cell[i]:
IF
row $=\mathrm{col}$
CHAN OF vertical up IS pipe.vertical[ i] :
CHAN OF horizontal right IS pipe.horizontal [i]: test.chan IS test[i]:
northwest IS pipe.diagonal[row]:
southeast IS pipe.diagonal[row+1]:
circle.element( right, up, northwest, southeast, test.chan )
TRUE
VAL INT row IS row.cell[i]:
VAL INT col IS col.cell[i]:
VAL INT up.number IS cell.row.col[row][col] :
VAL INT right.number IS up.number:
VAL INT down number IS cell.row.col[row+1][col] :
VAL INT left.number IS up.number-1:
CHAN OF vertical up IS pipe.vertical[up.number]:
CHAN OF vertical down IS
pipe.vertical[down.number]:
CHAN OF horizontal right IS pipe.horizontal[right.number]:
CHAN OF horizontal left IS pipe.horizontal[left.number]: test.chan IS test[i]:
square.element( left, right, up, down, test.chan )
PAR $\mathrm{i}=0$ FOR N
VAL INT most.number IS cell.row.col[i][N-1] :
CHAN OF horizontal right.most IS
pipe.horizontal[most.number]:
output.interface ( right.most, output.c[i], output.s[i],
nothing1, nothing2 )
PAR $i=0$ FOR whole.elements
test.chan IS test[i]:
test.number IS temp[i]:
test.chan $?$ test.number
CHAN OF diagonal se.out IS pipe.diagonal[circle.elements] :
se.out ? nothingl
SEQ
write.full.string(screen,"The middie result is:") newline(screen)
SEQ $i=0$ FOR whole.elements
write.real32(screen, temp[i], 6, 2)
newline(screen)
SEQ
VAL string IS "The result is:":
write.full.string(screen,string)
newline(screen)
SEQ $i=0$ FOR N
write.real32(screen, output.c[i], 6,2)
newline(screen)
SEQ $i=0$ FOR N
write.real32(screen, output.s[i], 6,2)
newline(screen)
keyboard ? any

### 4.8.4 A Brief Introduction to Transputers and Occam

### 4.8.4.1 Transputers

The transputer is a single-chip microprocessor developed by Inmos Ltd. It is the first microcomputer designed specially for use in parallel processing systems. Among its distinguishing features are: (a) special hardware for context switching between parallel processes on a single transputer processor; (b) point-topoint communication links for connecting two processors together, (c)special direct memory access hardware to move data quickly into and out of the links, and (d) an on-chip memory array [INMOS88a]. All of these features contribute to the efficient implementation of parallel processing tasks.

Figure 4.23 is a diagram of a generic transputer microprocessor. Several different types of transputer microprocessors actually exist range from T2xx, T4xx, T8xx to T9xxx, all of which


Figure 4.23 Block diagram of a transputer share the special features listed above. The actual size of the memory array, the number of links and the structure of the processing units and external memory interface differ among the various transputers available.

### 4.8.4.2 Occam

The transputer was developed with the specific intention of providing an efficient platform for the execution of the occam programming language. A brief review of the occam programming language is provided below. More details can be found from the occam 2 Reference Manual [INMOS88b] [Cok91].

The basic program structure in occam is the process. A process is an instruction or group of instructions in a program. These instructions initiate a variety of operations, the most primitive of which are assignment and communication. As a simple example of a process, we consider the assignment of two integer variables, a and b :

```
INT a,b:
a:=b
```

The first statement instantiates, or defines, the variable. Within this first statement, the word INT defines data type, and the list of variables to be defined follows. An array of variables is defined similarly with an array size prefix of the form [size], where size indicates the number of elements in the array. The scope of a variable is limited to the process following its definition. The assignment of $b$ to $a$, done in the second line, is a simple process. Larger processes can be constructed from groups of smaller processes. Theses similar processes must then indented as a group by two spaces.

In describing the temporal relationship between multiple processes, occam is among the most elegant of languages. Any process may execute before, after, at the same time as, or in place of another process. There are three occam statements which define the relationship between multiple processes: SEQ (sequential), PAR (parallel), and ALT (alternate). These three statements, together with the CHAN statement, replicated structures, and control structures are presented below.

SEQ. The SEQ (sequential) construct causes all of the following processes indented by two spaces to executed in the order listed. For example:

```
INT a,b:
SEQ
    a:=3
    b:=a
```

assigns 3 to $a$, and then a to $b$. Together, the four lines of the program can be considered a single process. The SEQ construct is the implicit programming structure found in traditional, singleprocessor computers.

PAR. The PAR (parallel) statement defines a set of processes which execute in parellel, at the same time. Each individual process included must be indented by two spaces in just the same way as the SEQ process. For example, consider three integer variables, $a, b$, and $c$. Using a PAR construct, we can multiply all of these variables by 2 at the same time:

```
INT a,b,c:
PAR
    a:=a*2
    b:=b*2
    c:=c*2
```

The occam programming model does not support shared memory. The following example is not legalm and will not compile, since $b$ is assigned simultaneously in both processes.

```
INT a,b:
PAR
    b:=a
    b:=2
```

The final value of $b$ cannot be determined, since in a parallel construct it is impossible to predict which process will execute first.

If several processes must use different portions of an array at the same time, the array must be broken down into disjoint subsets of elements using abbreviations. Each of the processes can then uniquely access an abbreviated portion of the original array. For example:

```
[2]INT array:
INT val1 IS array[pointer0]:
INT val2 IS array[pointer1]:
PAR
    val1:=3
    va12:=4
```

The valuses pointer 0 and pointer1 must be defined and assigned earlier in the program. If they are equal, the program will return a run-time error.

Although all of the processes in a PAR construct should, by definition, execute at the same time, on a single-processor computer the parallel processes will in fact have to time-share the CPU.

An additional structure, the PRI PAR (priority parallel) construct, provides a means of executing one process in preference to another. Only when the priority process is unable to proceed further (while waiting on an input or output, for example) can the other processes execute. A PRI PAR structure is written in the same way as a PAR, but with the first process listed in the PRI PAR structure having the higher priority.

The PRI PAR structure is especially useful for programs which need to execute a communication shell at the same time as a normal program task. Typically, the communication should be expedited at the expense of the task, since delaying the communication may mean starving another processor of work. PRI PAR structures are also very useful for real-time systems which must react to external interrupts.

Processes executing at the same time may communicate with each other an can run on physically separate processors. The PLACED PAR structure con configure parallel processes to run on physically distinct processors.

CHAN. Although two parallel processes cannot both assign values to the same variables, they can cummunicate variables through a CHAN (channel) structure. An input from a channel is performed with a statement of the channel name followed by a question mark and the variable to be assigned. An output on a channel is performed with a channel name followed by an exclamation point and the variable to be communicated. For example, one process can pass an integer value to another through the integer channel talk:

```
CHAN OF INT talk:
INT a,b:
PAR
    talk!a
    talk?b
```

A channel type can be any of the other variable types (for example, BYTE or REAL32), array of such types, or a combination of these. A channel with no defined type is defined as CHAN OF ANY. Channels can also be defined in arrays just as variables are.

The channel communication itself must take place simutaneously in both the input and output processes. This means that the two processes communicating must, at some level, be executing in parallel with each other. If one process wishes to output on a channel and there is no corresponding process doing an input on the same channel, the process attempting to output cannot proceed.

The TIMER channel is a special channel defined in occam. The TIMER definition allows an input from an associated channel to return the current system time. This channel is useful for real-time systems and performance measurements.

```
TIMER time:
INT a:
SEQ
    time?a
```

ALT. The ALT (alternative) construct provides a mechanism for selecting among a group of input processes. In an ALT construct, the first process albe to input will proceed and none of the other processes will execute. For example, given two channels, talk1 and talk2, we can write

```
CHAN OF INT talk1, talk2:
INT a,b:
ALT
        talk1!a
        b:=a*34
        talk2?a
            b:=a+1
```

In this case, if talk1 inputs $a$, then $a * 34$ will be assigned to $b$. If talk2 inputs $a$, then $a+1$ is assigned to b .

A processor implements an ALT structure by sequentially testing each of the channel inputs. If several inputs can proceed simultaneously it is not possible to predict which process will be chosen. Just as the PRI PAR provides a means to preferentially execute one of a group of parallel processes, so a PRI ALT will preferentially select one of several simultaneous inputs.

Replicated Structures. The SEQ, PAR, and ALT structures can all be replicated, that is, a single statement can define multiple processes. A replicated SEQ structure is written as follows:

```
SEQ i=start FOR count
```

The statement creates a sequential loop indexed by the integer $i$ which is initialized to start and repeats count times. The parameters start and count are also integer values. Because the structure is a SEQ, each iteration will proceed sequentially in numerical order. The integer variable $i$ is within scope only inside the process and does not have to be defined outside the process. A replicated PAR structure has the following form:

```
PAR i=start FOR start
```

This statement creats count number of processed which proceed in parallel. Each process is indexed by the integer $i$, whose value ranges from start to start+count -1 . A replicated ALT structures requires an array of channels of size start+count, and is written as:

```
[start+count]CHAN OF INT in:
INT a:
ALT i=start FOR start
    in[i]?a
        a:=a+i
```

This code creats a set of processes, each of which attempts to do an input with its respective element of the array of channels in. The first process to do an input on its channel will proceed and add the index value to the input value.

Control Structure. The occam language also includes control structures which permit branching in a program. These structures include WHILE, IF and CASE statements, as well as subroutine and function calls. To support the IF and WHILE structures, logical variables are used which can be either TRUE or FALSE. A WHILE statement will repeat a process as long as its associated logical variable is TRUE. For example:

```
INT a:
SEQ
    a:=0
    WHILE (a<4)
        a:=a+1
```

will iterate in the WHILE loop until $a=4$. An IF structure will select the first process in its list whose guard is TRUE. For example:

```
INT a:
SEQ
    a:=0
    IF
        a=4
            a:=8
        a=0
            a:=2
```

will select the second alternative and set a equal to two. Note that at least one of the logical processes must be TRUE or the IF statement will never complete. IF structures can alse be replicated:

```
[start+cound] INT a:
IF i=start FOR count
    a[i]=0
    a[i]:=3
```

The CASE statement, another control structure, is similar to an IF structure in that one process from a group is seleted and the others areignored. The CASE statement does note use logical variables, but executes the process whose guard is equal to the argument of the CASE itself. For example:

```
INT a:
SEQ
        a:=3
        CASE a
        7
            \(a:=a+4\)
        3
            \(a:=a / 4\)
        ELSE
            \(\mathrm{a}:=0\)
```

The ELSE process at the end will execute only if no previous processes were executed.

Procedure subroutines are created with the PROC (process) definition. A simple process with two arguments, double, is illustrated here. Notice that the type of each argument must be stated.

```
PROC double(INT arg1, arg2)
        SEQ
            arg1:=arg2*2
:
```

A colon indicates the end of a process definition. The procedure is called with a statement of the procedure name:

```
INT a,b:
SEQ
    a:=4
    double (a,b)
```

Functions are defined in a slightly different way. A function is defined with a data type, and must explicitly return a value:

```
INT FUNCTION double(INT arg)
    VALOF
        RESULT (arg*2)
:
```

As with a process, a function is called by using the name of the function with the appropriate arguments:

```
INT \(a, b:\)
SEQ
    \(a:=4\)
    b:=double (a)
```

In order to assist programmers in the organization and construction of large programs, the occam language supports the use of libraries. These libraries can be defined as separate routines accessible to any other program which refereneces the library. A library can be accessed by including the statement:
\#USE "library_file_name"
within the scope of any references to routines or variables defined within the library.

## Chapter 5

## SYSTOLIC ARRAY FOR LINEAR AND INVERSE MATRIX SYSTEM

### 5.1 Introduction

THE problems of solving linear systems and matrix inverses are central in scientific computations. In addition to the scientific computations, they have found wide applications in areas such as digital signal processing, control systems, image processing and finite element analyses.

Among the various methods for the linear system solution and matrix inversion, the method of LU decomposition with backward and forward substitutions, motivated by Gaussian elimination, is proved to be an efficient method for solving linear systems as well as matrix inverse.

When the number of equations is larger than the number of unknowns in a linear system, we need find the least squares solution of the system; and the corresponding matrix inverse becomes pseudoinverse. For the least squares and matrix pseudo-inverse problem, orthogonization procedures such as QR decomposition are more appropriate.

The solution of linear systems (or least squares problems) and matrix inverse (or pseudo-inverse) requires a computational load of $O\left(n^{3}\right)$ operations (or $O\left(m n^{2}\right)$ for least squares problems and pseudoinverse problems). As the demands for fast computing increase, e.g. in real time signal processing and real time control, the parallel processing for the linear system and matrix inverse becomes highly necessary.

Systolic array architecture is very efficient for parallel matrix computing since the architecture is easy to implement by advanced VLSI techniques owing to its characteristics of local connections, data flow structures, regular modules of processing elements. Many systolic architectures have been developed for matrix computations such as matrix multiplication, matrix triangularization, SVD, etc.

In this chapter, we use the LU decomposition and QR decomposition methods, introduced by W.M. Gentleman and H.T. Kung [Gentleman81], to solve the linear systems (and least squares problems) and matrix inversion (and pseudo-inverse) [Evans93b]. In the next section, a brief treatment for solving linear systems and matrix inverse is given. Sec. 5.3 describes the systolic array for linear systems using LU decomposition in detail. Sec 5.4 extends the result in Sec. 5.3 by applying QR decomposition in the elimination to compute the pseudo-inverse of a rectangular matrix and solving least squares problems. Sec. 5.5 discusses the possible applications of the proposed systolic arrays. Sec. 5.6 analyses the performance of the systolic architecture while Sec. 5.7 presents a more efficient solution for linear system and matrix inverse. Sec. 5.8 summarizes the chapter. The simulation of the systolic architecture in a single transputer by the parallel language Occam 2 is appended in Sec. 5.9.

### 5.2 Solution of Linear Systems and Matrix Inverse

Consider the problem of finding $X$ such that

$$
\begin{equation*}
A X=B \tag{5.1}
\end{equation*}
$$

where $\boldsymbol{A}$ is an $n \times n$ matrix, $X$ an $n \times p$ matrix and $B$ an $n \times p$ matrix. The matrix $A$ is assumed to be nonsingular throughout the chapter. Eq. (5.1) represents a multiple right-hand-side (rhs) linear system problem. To solve the system, the matrix $\boldsymbol{A}$ is factorised into the product of $L$ and $\boldsymbol{U}$, that is

$$
\begin{equation*}
A=L U \tag{5.2}
\end{equation*}
$$

where $L$ is lower triangular and $U$ is upper triangular if the first ( $1 \mathrm{l}-1$ ) principal submatrices of $A$ are nonsingular. Then the problem becomes two triangular systems

$$
\begin{equation*}
L Y=B \tag{5.3}
\end{equation*}
$$

$$
\begin{equation*}
U X=Y \tag{5.4}
\end{equation*}
$$

The first triangular system in Eq. (5.3) can be solved by the forward substitution method while the second one in Eq. (5.4) can be solved by the back substitution.

The above discussion is based on the presumption of the existence of the LU decomposition of the given matrix $\boldsymbol{A}$. However, the LU decomposition described above does not always exist or maybe causes numerical instability in some cases. The pivoting approach is then introduced in order to guarantee the existence of the LU decomposition and to improve the numerical stability. The modification of the above decomposition is given as

$$
\begin{equation*}
P A=L U \tag{5.5}
\end{equation*}
$$

where $P$ is the permutation matrix with the dimension of $n \times n$ and $P A$ is a row permuted version of the original matrix $\boldsymbol{A}$. The triangular systems then become

$$
\begin{equation*}
L Y=P B \tag{5.6}
\end{equation*}
$$

$$
\begin{equation*}
U X=Y \tag{5.7}
\end{equation*}
$$

There are several methods for pivot selecting. For example, the column pivoting and the complete pivoting are commonly used to ensure a stable solution. The following algorithm describes the LU decomposition with column pivoting [Golub89]:

```
for }k=1:|-
```

    pivot finding: determine \(\mu\) so that \(\mid A(\mu, k)=\|A(k: n, k)\|_{\infty}, \quad k \leq \mu \leq n\)
    exchange rows: \(A(k, k: n)-A(\mu, k: n) ; \quad p i v(k)=\mu\)
    if \(A(k, k) \neq 0\)
            for \(j=k+1: n\)
                            construct multiplier: \(m=A(j, k) / A(k, k)\)
                        row operation: \(A(j, k: n)=A(j, k: n)-A(k, k: n) * m\)
            end
    end
    end

However, the global communications are involved in the above algorithm in selecting the pivots by comparing the entries of certain columns of the matrix $\boldsymbol{A}$. The algorithm is hard to be localized in order to form a systolic algorithm. A modification of the method of selecting pivots is necessary so that the global communications are avoided. Gentleman and Kung first proposed the neighbour pivoting strategy to make it possible to implement the triangular factorisation in a systolic array [Gentleman81].

The solution of the upper triangular system in Eq. (5.7) is easily obtained by the back substitution method. The algorithm for the back substitution can be expressed as follows:

```
\(Y(n, 1: p)=Y(n, 1: p) / U(n, n)\)
for \(i=n:(-1): 1\)
    \(Y(i, 1: p)=\left(Y(i, 1 ; p)-\sum_{j=i+1}^{n} U(i, j) Y(j, 1: p)\right) / U(i, i)\)
end
```

The forward substitution is a similar algorithm as the back substitution and we do not list it here.

Assuming that $B$ is an $n \times \pi$ unit matrix $I$ in Eq. (5.1), the problem of finding $X$ then becomes the matrix inverse problem, i.e.,

$$
\begin{equation*}
X=A^{-1} \tag{5.8}
\end{equation*}
$$

Therefore, we can use the same systolic array to solve the problems of both the linear system and the matrix inverse.

### 5.3 Systolic Array for Linear and Inverse Matrix Systems

In this section, we first introduce the systolic array for LU decomposition. Then the systolic arrays for back substitution and matrix multiplication are also discussed. Finally, we combine these systolic arrays together to form a new Systolic Array for the Linear and Inverse Matrix Systems (SALIMS).

### 5.3.1 LU Decomposition

In Chap. 4, we have designed systolic arrays for LU decomposition. We redraw the triangular systolic array for LU decomposition originally introduced by Gentleman and Kung [Gentleman81] in Figure 5.1a. The systolic array enjoys a pipelining period of 1 and thus it is preferred for many applications. There are two types of processing elements (PEs) or cells which we name as circle element/cell and square element/cell respectively. The details of the PEs are given in Figure 5.1 lb where the neighbour pivoting scheme is adopted to ensure a stable solution while local communication property of systolic
array is maintained. During each time step, each circle element executes one division and a comparison while each square element performs one inner product step (ips).


Figure 5.1 Systolic array for LU decomposition: (a) The array; (b) PE definitions

The upper triangular matrix $\boldsymbol{U}$ is stored in the triangular array after the matrix $\boldsymbol{A}$ passes through the array. However, the lower triangular matrix $L$ as well as the permutation matrix $P$ cannot be derived from the array directly.

In our case of solving linear systems and matrix inverse, we attach a rectangular array at the right hand side of the triangular array as shown in Figure 5.2. The square elements have the same function as those in the triangular array. Instead of storing information about $L$ and $P$, the matrix $L^{-1} P B$, i.e. the
matrix $\boldsymbol{Y}$ in Eq. (5.6), is stored in the PEs of the rectangular array. This is the same operation as the forward substitution of Eq. (5.6). It is easy to understand when we note that the systolic array performs the operation of premultiplying the input matrices of both $A$ and $B$ by $\boldsymbol{L}^{-1} \boldsymbol{P}$.

### 5.3.2 Back Substitution

After the triangularization of the matrix $\boldsymbol{A}$ and the forward substitution given respectively in Eqs. (5.5) and (5.6), there remains the


Figure 5.2 Systolic array for LU decomposition and forward substitution problem of solving the triangular system in Eq. (5.7). In Sec. 5.2 we describe an algorithm in sequential form for the back substitution. Based on the algorithm, we can design systolic array to solve the upper triangular system.

There exist different kinds of systolic array for solving a upper triangular matrix system. For example, we could use the linear array of Kung and Leiserson [KungHT80c]. Although the linear array is efficient to solve a given triangular system, there are difficulties to connect the output of the systolic array in Figure 5.2 to the input of the linear array for solving a triangular system.

We propose here an on-the-fly solution of the triangular system by using the same systolic array shown in Figure 5.2. Assume that the entries of the upper triangular matrix $U$ is stored in the triangular systolic array shown in Figure 5.3a with the PE definitions shown in Figure 5.3b. The matrix $C$ of size $q \times n$ enters into the systolic array from the top of the array. The output matrix at the right hand
side of the array is $\left(C U^{-1}\right)^{T}$. If $C$ is set to an $n \times n$ unit matrix $I$, the output is the inverse of the $U$ transpose.

If we multiply the $\boldsymbol{Y}$ by $\left(C U^{-1}\right)^{T}$, we can obtain the solution of Eq. (5.7) when $C$ is the $n \times n$ unit matrix $I$. The systolic array for the matrix multiplication, which we have intensively studied in Chap. 3, is illustrated in Figure 5.4. The matrix $Y$ is stored in the rectangular systolic array while the input matrix is from the left and upper sides and the result matrix, $D+C A^{-1} B$,


Figure 5.3 Systolic array for computing $\left(C U^{-1}\right)^{T}$.

### 5.3.3 Systolic array for linear

 systemHaving discussed the systolic arrays for the LU decomposition, the forward substitution, the backward substitution and the matrix multiplication, we can combine these arrays to compose a more complex systolic array to solve the linear system as well as matrix inverse.

The computation of the systolic array for linear and inverse matrix system


Figure 5.4 Systolic array for matrix multiplication
can be divided into two phases. In the first phase, the LU decomposition and the forward substitution are executed in a triangular array and a rectangular array respectively. Then the back substitution, including the triangular matrix inverse and the matrix-matrix multiplication, is also executed in a triangular systolic array and a rectangular systolic array respectively during the second phase. Thus we can implement the two phase processing in the same systolic array. Figure 5.5 presents the systolic array with the input data and output data. Figure 5.6 defines the PE functions for the SALIMS. The definitions is actually a combination of the definitions in Figures $5.1 \mathrm{~b}, 5.3 \mathrm{~b}$ and 5.4 b .

With the four matrices $A$


Figure 5.5 Systolic array for linear and inverse matrix system $(n \times n), B(n \times p), C(q \times n), D$
$(q \times p)$ as the input arranged in a form as given in Figure 5.5 , the systolic array can compute the following output

$$
\begin{equation*}
X=C A^{-1} B+D \tag{5.9}
\end{equation*}
$$



Figure 5.6 PE definitions for SALIMS

If the matrix $C$ is a unit matrix and $D$ is a zero matrix, the array gives the solution of linear system as in Eq. (5.1). This systolic array can solve many other problems which we will discuss in a later section.

It is noticed that there is a slight difference between the definitions of the PEs in triangular array and the PEs in rectangular array during the back substitution phase. The PEs in rectangular array can be modified so that the square PEs in the whole array have same function by changing addition operation into subtraction. This modification cause the output matrix of the systolic array to change from $\boldsymbol{D}+\boldsymbol{C} \boldsymbol{A}^{-1} \boldsymbol{B}$ to $\boldsymbol{D}-\boldsymbol{C} \boldsymbol{A}^{-1} \boldsymbol{B}$. Therefore the systolic array has only two type of PEs after this modification. In the PE definitions, we also add one control signal $p$ into the vertical data channels to signify the computation phases.

The array stores the results of the upper triangular matrix $U$ in the triangular sub-array and the matrix $\boldsymbol{Y}$, i.e. $\boldsymbol{L}^{-1} P \boldsymbol{P}$, in the rectangular array. The array outputs the matrix $\left(C U^{-1}\right)^{T}$ on the right hand side and matrix $X$ at the bottom of the array.

### 5.4 Least Squares Problems

The systolic array discussed in the last section can be immediately extended to least squares problems by applying the QR decomposition instead of the LU decomposition.

Let $\boldsymbol{A}$ be a rectangular matrix of size $m \times n$, with $m \geq n$ (in practice, we often have $m » n$ ), $\boldsymbol{b}$ a column vector with $m$ components, and $\boldsymbol{x}$ an unknown column vector with $n$ components. The linear system $\boldsymbol{A} \boldsymbol{x}=\boldsymbol{b}$ has more equations than unknowns. This is an overdetermined system and thus there is, in general, no exact solution for such a system. The LU decomposition method used in the systolic array discussed above can not handle this kind of linear system.

Now we seek to minimise the Euclidean norm of the vector $\|\boldsymbol{A x} \boldsymbol{x} \boldsymbol{b}\|$. Assume that the QR decomposition of the matrix $A$ can be given by $A=Q R$ (more details can be found in Chap. 4). We apply the triangularization process to both the matrix $\boldsymbol{A}$ and the vector $\boldsymbol{b}$ to obtain

$$
\varrho^{T}(A \quad b)=\binom{Q_{1}}{Q_{2}}\left(\begin{array}{ll}
A & b
\end{array}\right)=\left(\begin{array}{cc}
R_{1} & Q_{1} b  \tag{5.10}\\
0 & \varrho_{2} b
\end{array}\right)
$$

where $R_{1}$ is an upper triangular matrix of size $n \times n$ (the first $n$ rows of $R$ ). Since $Q$ is orthogonal, we have

$$
\begin{equation*}
\|A x-b\|=\left\|Q^{T}(A x-b)\right\|=\left\|R_{1} x-Q_{1} b\right\|+\left\|Q_{2} b\right\| \tag{5.11}
\end{equation*}
$$

Thus to minimize $\|A x-b\|$, we need to minimize the first term in the right hand side of the above equation as the second term is a constant value. Consequently, the solution of the least squares problem is the solution to the system of $n$ equations

$$
\begin{equation*}
R_{1} x=Q_{1} b \tag{5.12}
\end{equation*}
$$

and the residual error is $\left\|Q_{2} b\right\|$ [Golub89]. The above discussion can be also expanded to a multiple right hand side least squares problem $A X=B$ with the solution as given by

$$
\begin{equation*}
X=\left(A^{T} A\right)^{-1} A^{T} B=R_{1}^{-1} Q_{1} B \tag{5.13}
\end{equation*}
$$

If $\boldsymbol{B}$ is a unitary matrix, $\boldsymbol{X}$ becomes the pseudo-inverse of the matrix $\boldsymbol{A}$

$$
\begin{equation*}
A^{*}=\left(A^{T} A\right)^{-1} A^{T} \tag{5.14}
\end{equation*}
$$

When $\boldsymbol{A}$ is a full-rank square array, the above solution to least squares problem and matrix pseudoinverse become respectively the usual solution linear system and matrix inverse as discussed earlier.

In this sense, the least squares system can be considered as a generalized linear system while the pseudo-inverse can be considered as a generalized matrix inverse.

Now we explain how the systolic array in Figure 5.5 can be modified to find the least squares solution and the matrix pseudo-inverse. We modify the PE definitions in Figure 5.6 by changing the LU decomposition operation into QR decomposition while keeping the rest operations unchanged. The input matrix $A, B, C$ and $D$ are arranged in the exact same format as shown in Figure 5.5 except that $\boldsymbol{A}$ can be a rectangular matrix now.

During the first phase, i.e., QR decomposition phase, the systolic array performs the Givens rotations on the matrices $\boldsymbol{A}$ and $\boldsymbol{B}$. The triangular array performs the operation of $\boldsymbol{Q}^{\boldsymbol{T}} \boldsymbol{A}$ and results in an upper triangular matrix $\boldsymbol{R}_{1}$ which is stored in the individual PEs in the triangular array. In the mean time, the rectangular array performs the operation of $Q^{T} B$ and stores the result of $Q_{1} B$ in the individual PEs in the rectangular array while $Q_{2} B$ is propagated out of the array at bottom.

The second phase is the back substitution. The triangular matrix receives the input $C$ and sends the output ( $\left.C R^{-1}\right)^{T}$ to the rectangular array. The rectangular array performs matrix multiplication and addition operation and generates the following result:

$$
\begin{equation*}
X=D+C R^{-1} Q_{1} B=D+C A^{+} B \tag{5.15}
\end{equation*}
$$

Thus the systolic array can be used to solve the linear squares problem and find the matrix pseudoinverse by setting the input matrix appropriately.

### 5.5 Possible Applications

Although the systolic array is designed for linear system solution (or least squares problem), it can solve a variety of linear algebra problems. From Eqs. (5.9) and (5.15), we can easily find more applications by setting the four matrices $A, B, C$, and $D$ in different ways.

We will concern with only the systolic array using $L U$ decomposition here, but the results can be applied to the systolic array using QR decomposition. The systolic array can solve different kinds of linear systems including the right side system and the left side system. It can compute the matrix
inverse with $B=C=I$ and $D=0$. The matrix-matrix-multiplication and the matrix-multiplication-andaddition can be solved by the same array. The array also gives the result of the LU decomposition by storing $\boldsymbol{U}$ in the triangular array. Table 5.1 lists out the possible applications of the array and the settings are also given for the different problems.

Table 5.1 The possible applications of the proposed systolic array

| Applications | Formulae | Settings | K. Solution |
| :---: | :---: | :---: | :---: |
| right side linear system (1) | $A X=B$ | $C=I, D=0$ | $X=A^{-1} B$ |
| left side linear system (2) | $X A=C$ | $B=I, D=0$ | $X=C A^{-1}$ |
| linear system (3) | $A(X-D)=B$ | $C=I$ | $X=A^{-1} B+D$ |
| linear system (4) | $(X-D) A=C$ | $B=I$ | $X=C A^{-1}+$ D |
| matrix-matrix multiplication | $C B$ | $A=I, D=0$ | $X=C B$ |
| matrix multiplication and addition | $C B+$ ) | $A=I$ | $X=C B+D$ |
| LU decomposition | $A=L U$ |  | $U$ stored in the triangular array |
| matrix inverse | $A X=I$ or $\boldsymbol{X} A=\boldsymbol{I}$ | $B=I, C=I, D=0$ | $X=A^{-1}$ |
| Faddeev's algorithm | $A Z=B, X=C Z+D)$ |  | $X=I)+C A^{-1} B$ |

The above list includes only the multiple right hand side systems and matrix-matrix operations. Their special case contains single right hand side systems and matrix-vector operations which we do not provide with Table 5.1.

The use of QR decomposition can augment the above list by including a variety of least square problems and pseudo matrix inverse. The above list may be further extended by applying the basic operations in the list to practical problems or by combining with another systolic array to solve other problems.

### 5.6 Performance Analysis

In this section, a brief treatment is given on the performance analysis of the proposed systolic array architecture in Figure 5.5.

The number of processing elements in the systolic array is $n(n+1) / 2+n p$. The circle processing element executes a division with a comparator in the LU stage during each time step while the square processing element performs one inner product step operation. Roughly both types of PEs have the same computational complexity. We use the term "flop" to refer to a inner product step or a division operation with comparison.

The systolic array has a pipelining rate of 1 , which makes it very attractive. The block pipelining period of the systolic array is $(n+q)$.

The systolic array starts its computation when the first element of $A$ enters into the $P E$ at the top left corner. It takes $(n+p-1)$ time steps before the last element of the first row of $B$ reaches the systolic array at the top right corner due to the skew format of the input data. Afterwards, it takes another $(n+q-1)$ times step for the last element of the last rows of $D$ to reach the top right PE. The computation result at the top right PE will propagate vertically (and be modified) until it passes through the systolic array after $n$ time steps. Therefore, the systolic array requires a $3 n+p+q-2$ computation time to complete the operation in Eq. (5.9).

Below, we will discuss the speedup and efficiency of the systolic array for different problems.

### 5.6.1 The Right Side Linear System Solution

The LU decomposition requires $n^{3 / 3}$ flops while both the backward substitution and the forward substitution stages require $n^{2} p / 2$ flops. Therefore it requires $n^{3} / 3+n^{2} p$ flops to solve the linear system in Eq. (5.1) using the LU decomposition method.

To solve the linear system, we need to set $C$ to be a unit matrix of the same size as $A$, i.e., $q=n$. Thus the computation time for the right side hand linear system is $411+p-2$. The speedup of the array is therefore given by

$$
\begin{equation*}
s=\frac{\frac{n^{3}}{3}+n^{2} p}{4 n+p-2} \tag{5.16}
\end{equation*}
$$

The efficiency is

$$
\begin{equation*}
e=\frac{s}{\frac{n(n+1)}{2}+n p} \tag{5.17}
\end{equation*}
$$

If $p$ is much smaller than $n$ and $n$ is large enough, we have

$$
\begin{equation*}
s=\frac{1}{12} n^{2}, \quad e=\frac{1}{6}=16.7 \% \tag{5.18}
\end{equation*}
$$

If $p$ is equal to $n$, the speedup and efficiency become

$$
\begin{equation*}
s=\frac{4}{15} n^{2}, \quad e=\frac{8}{45}=17.8 \% \tag{5.19}
\end{equation*}
$$

For $k$ linear systems solved in a pipeline way, it requires $k\left(n^{3} / 3+n^{2} p\right)$ flops finishing the multiple tasks in $2 k n+(2 n+p-2)$ steps. Thus, the speedup becomes

$$
\begin{equation*}
s=\frac{k\left(\frac{n^{3}}{3}+n^{2} p\right)}{2 k n+(2 n+p-2)}, \quad \lim _{k \rightarrow \infty} s=\frac{n}{2}\left(\frac{n}{3}+p\right) \tag{5.20}
\end{equation*}
$$

Similarly, we can obtain the efficiencies of the array as

$$
\begin{array}{ll}
e=1 / 3=33.3 \%, & \text { for } p<n \\
e=4 / 9=44.4 \%, & \text { for } p=n \tag{5.21}
\end{array}
$$

### 5.6.2 Faddeev's Algorithm

Faddeev's algorithm computes the matrix $X$ in Eq. (5.9). It requires $n^{3} / 3+n^{2} p$ flops to compute $A^{-1} B$ by the method of LU decomposition and the backward and forward substitutions. The remaining matrix multiplication and addition requires $n^{2} p$ flops. In total, it therefore needs $n^{3} / 3+2 n^{2} p$ flops to calculate the matrix $X$. Now the speedup becomes

$$
\begin{equation*}
s=\frac{\frac{n^{3}}{3}+2 n^{2} p}{4 n+p-2} \tag{5.22}
\end{equation*}
$$

For a single computing task, the efficiency is

$$
\begin{array}{cc}
e=1 / 6=16.7 \% & \text { for } p « n \\
e=14 / 45=30.1 \% & \text { for } p=n \tag{5.23}
\end{array}
$$

Similarly, we can obtain the efficiency for multiple tasks as

$$
\begin{array}{ll}
e=1 / 3=33.3 \% & \text { for } p « n \\
e=7 / 9=77.8 \% & \text { for } p=n \tag{5.24}
\end{array}
$$

### 5.6.3 Matrix Inverse

For matrix inverse, the array has the same speedup and the same efficiency as those for the linear system with $p=\boldsymbol{n}$ as discussed above if we do not consider the special property of the setting $B=I$. Below, we will consider the speedup and efficiency of the systolic array when specially applied for matrix inversion problem.

The LU decomposition still requires $n^{3} / 3$ flops. We can find that the forward substitution stages require $n^{3 / 6}$ flops by taking into account the special property of the matrix $B=I$. The back substitution takes $n^{3} / 3$ flops because the right hand side after the forward substitution is a triangular matrix. Totally it requires $5 n^{3} / 6$ flops to find the matrix inverse. In this case, the speedup and efficiency of the systolic array for finding the matrix inverse will be reduced. The speedup of the array is given by

$$
\begin{equation*}
s=\frac{5 n^{3} / 6}{5 n-2} \approx n^{2} / 6 \tag{5.25}
\end{equation*}
$$

The efficiency is

$$
\begin{equation*}
e=\frac{s}{\frac{n(n+1)}{2}+n^{2}} \approx \frac{1}{9} \tag{5.26}
\end{equation*}
$$

Similarly, we can obtain the efficiency for multiple tasks as

$$
\begin{equation*}
s=5 n^{2} / 12, \quad e=5 / 18=27.8 \% \tag{5.27}
\end{equation*}
$$

### 5.7 A More Efficient Systolic Array for Linear System

We consider the systolic array in Figure 5.5 to solve the following problem

$$
\begin{equation*}
X=C A^{-1} \tag{5.28}
\end{equation*}
$$

According to Table 5.1, it is required to set $D=0$ and $B=I$. We assume that no pivoting is needed to find the solution in Eq. (5.28), i.e., $\boldsymbol{P}=\boldsymbol{I}$. During the first phase of processing, the systolic array receives the matrices $A$ and $B(=I)$, and then performs the LU decomposition and the forward substitution to result in an upper triangular matrix $\boldsymbol{U}\left(=\boldsymbol{L}^{-1} P A\right)$ and a lower triangular matrix $\boldsymbol{L}^{-1}\left(=\boldsymbol{L}^{-1} \boldsymbol{P B}\right)$. In the second phase, the matrix $\boldsymbol{X}$ in Eq. (5.28) can be computed.

Note that in this special case, we have a resultant lower triangular matrix and an upper triangular matrix after the first phase of computation. This suggests that a triangular array can be used in order to reduce the number of $P E$ required instead of the rectangular array in Figure 5.5. As the matrices $B(=I)$ and $D(=0)$ are constant matrix, we can eliminate the input channels for them and modify the PE functions accordingly. The modified systolic array with less PEs is shown in Figure 5.7 where the PE definitions are also included.

This systolic array for computing $C^{-1}$ was first proposed by Comon and Robert [Comon87]. This design was also obtained by the multimesh


Figure 5.7 A more efficient systolic array for linear system graph method (MMG) in [Moreno92].

This array can also be used to solve a right side linear system which can be obtained by transpose of a left side linear system.

This systolic array is more efficient than the systolic array in Figure 5.5 when it is used to solve linear systems and find matrix inverses. It has $n(n+1)$ PEs comparing $n(3 n+1) / 2$ for the previous systolic array with $p=n$. This means a $50 \%$ saving of hardware while keeping the same throughput. The previous systolic array has an advantage that it can solve more problems using the same array.

This systolic array has a constraint that it can not be used to solve a linear system or a matrix inverse when pivoting is necessary due to the array structure. There are, however, some important matrix systems which do not need pivoting. Examples are strictly column-sum dominant matrix (in which the element on the main diagonal is larger than the sum of the magnitudes of the remaining elements of that column), diagonal dominant matrix, and positive definite matrix [Nielen91].

### 5.8 Summary

In this chapter, a Systolic Array architecture for Linear and Inverse Matrix Systems (SALIMS) is designed by combining LU decomposition, backward and forward substitutions and matrix multiplication in a $n p+n(n+1) / 2$ PE array. The array can be used to solve a variety of linear systems and other problems such as matrix multiplication etc. The architecture is simulated in a single transputer by Occam 2 and the program and some numerical examples are presented in the appendices.

We also expand the systolic array by applying the QR decomposition instead of LU decomposition. The QR decomposition has a more stable numerical property. The introducing of the QR decomposition also allows the systolic array to perform least squares solution and pseudo-inverse computation.

We also present a more compact and efficient systolic array for linear systems and matrix inverse when pivoting is not necessary.

The proposed systolic arrays can solve a lot of problems including the linear system and the matrix inversion. We have analysed the performance of the systolic array in Figure 5.5. The array has different efficiencies for different problems. In some cases such as matrix inverse, it is not very efficient. However in cases such as Faddeev's algorithm it is very efficient. Like most of other systolic arrays, this systolic array is much more efficient when processing multiple systems in a pipeline way. The feature that the array can solve the many problems is very interesting. The complex computing problems composed of different kinds of matrix operations may be easily solved by this array.

### 5.9 Occam Program List

### 5.9.1 Occam Program for the Systolic Architecture

```
#- }\begin{array}{c}{\mathrm{ Systolic Array for Linear and Inverse matrix }}\\{\mathrm{ -- Systms (SALIMS) via LU Decomposition Technique}}\\{\mathrm{ (y C.R. WAN }} by C.R. WAN
```


## --

```
- user call
\#USE snglmath
\#USE userio
-- declarations
PROTOCOL MV IS REAL32; BOOL:
\begin{tabular}{ll} 
VAL INT \(n\) IS 3: & \(-n \times n\) is the dimension of the input matrix \\
VAL INT p IS 3: & -nxp is the dimension of the matrix B
\end{tabular}
```

VAL INT numchan ver IS total.element +p : - number of the ver channels
VAL INT num.chan.hor IS total.element: - number of the hor channels
[num.chan.ver] CHAN OF REAL32 pipe.vertical:

- Channels above and on the
[mumchan hor] CHAN OF MV pipe.horizontal: -- right of each element
[num chan.ver] INT row:
-- row number of ith element
[num.chan.ver] INT col:
[ $\mathrm{n}+1][\mathrm{np}]$ INT num:
corder number for ith row
-- order number for ith row
- and jth column element

INT any:

- data preparation

PROC data([][]REAL32 A, [][|REAL32 X)

```
[n][p] REAL32 B: - |A B|
[n][n] REAL32 C: -- X=| | extended input matrix
[n][p] REAL32 D: -- |C D|
SEQ
- assign values for matrix A
-
A[0][0]:= 1.0 (REAL32)
A[0][1]:= 0.0 (REAL32)
A[0][2]:= 3.0 (REAL32)
A[1][0]:=2.0 (REAL32)
A[1][1]:=4.0 (REAL32)
A[1][2]:= 1.0 (REAL32)
A[2][0]:= 3.0 (REAL32)
A[2][1]:=2.0 (REAL32)
A[2][2]:=6.0 (REAL32)
- assign values for matrix B (nxp unit matrix)
-
SEQ i=0 FOR n
    SEQ j=0 FOR p
        IF
            i=j
            B[i][j]:=1.0 (REAL32)
            TRUE
            B[i][j]:=0.0 (REAL32)
--
-- assign values for matrix C (n\timesn unit matrix)
-
SEQ i=0 FOR n
    SEQj=0 FOR n
        IF
            i=j
            C[i][j]:=1.0 (REAL32)
            TRUE
                C[i][j]:=0.0 (REAL32)
--
- assign values for matrix D (nxp zero matrix)
-
SEQ i=0 FOR n
    SEQ j=0 FOR p
        D[i][j]:=0.0 (REAL32)
-- assign values for matrix X ( (n+n) x(n+p) matrix)
```

```
SEQ i=0 FOR n
    SEQj=0 FOR n
        X[i][j]:=A[i][j]
    SEQ i=0 FOR n
    SEQ j=0 FOR n
        X[n+i][j]:=C[i][j]
    SEQ i=0 FOR n
    SEQ j=0 FOR p
        X[i][n+j]:=B[i][j]
    SEQ i=0 FOR n
    SEQ j=0 FOR p
        X[n+i][n+j]:=D[i][j]
    - display X on screen
    -
    write.full.string(screen, "The Extended Input Matrix: ")
    newline(screen)
    newline(screen)
    SEQ i=0 FOR n2
        SEQ
        SEQ j=0 FOR np
            write.real32(screen, X[i][j], 6, 2)
        newline(screen)
    newline(screen)
```

- 

-- define the function of circle elements of triangular array
-
PROC circle.element(CHAN OF MV east, CHAN OF REAL32 north)
REAL32 x, x.in, m.out, $z$ :
BOOL v.out:
SEQ
- 1. initialisation
$\bar{x}:=0.0$ (REAL32)
- 2. LU decomposition
--
SEQ $i=0$ FOR $n$
SEQ
north ? x.in
IF
$\mathrm{ABS}(\mathrm{x} . \mathrm{in})>=\mathrm{ABS}(\mathrm{x})$
SEQ
v.out := TRUE
IF
$x$.in $<>0.0$ (REAL32)
m.out := -(x/x.in)
TRUE
m.out: $=0.0$ (REAL32)
$\mathrm{x}:=\mathrm{x}$.in
TRUE
SEQ
v.out := FALSE
m.out :=-(x.in/x)
east ! m.out; v.out
-
- 3. Back substitution (inverse of U)
--
SEQ $\mathrm{i}=0$ FOR n
SEQ
north ? x in
$z:=x$ in $/ x$
east $!z$; TRUE
:

```
-1. initialisation
-
x:=0.0(REAL32)
-2. LU decomposition
SEQ i=0 FOR n
    SEQ
        PAR
        north? x.in
        west?m.in; v.in
        IF
        v.in
            SEQ
            x.out := x+(m.in*x.in)
            x := x.in
        TRUE
            x.out := x.in + (m.in*x)
        PAR
        south ! x.out
        east ! m.in; v.in
-3. Back substitution (matrix multiplication)
SEQ i=0 FOR n
    SEQ
        PAR
        north ? xin
        west ? z; v.in
        x.out:=x.in-(z*x)
        PAR
        south ! x.out
        east ! z; TRUE
- define the function of square elements of rectangular array
PROC square.element.R(CHAN OF MV west, east, CHAN OF REAL32
north, south)
REAL32 x, x.in, x.out, z,m.in:
BOOL v.in:
SEQ
-1. initialisation
-> := 0.0 (REAL 32)
--
-- 2. LU decomposition (Forward substitution)
SEQ i=0 FOR n
        SEQ
        PAR
        north ? x.in
        west ? m.in; v.in
        IF
            v.in
                SEQ
                x.out:= x+(m.in*x.in)
                x := xin
                TRUE
                x.out := x.in+(m.in*x)
        PAR
        south ! x.out
        east ! m.in; v.in
-
-3. Matrix Multiplication
-
SEQ i=0 FOR n
    SEQ
        PAR
            north ? x.in
            west ? z; v.in
        x.out:=x.in+(z**)
        PAR
            south ! xout
```

-- assign the number to i-th row and j-th column PE
PROC row.col.to.num([][] INT num)
SEQ i=0 FOR ( }\textrm{n}+1
SEQ j=i FOR (np-i)
num[i][j]:=((((2*np)-(i-1))*i)/2)+(j-i)
:
-- assign the number to i-th row and j-th column PE
--
PROC num.to.row.col([]INT row, []INT col)
INT k:
SEQ i=0 FOR (n+1)
SEQj=i FOR (np-i)
SEQ
k:=num[i][j]
row[k]:=i
col[k]:=j
:
-- wait until typing any key
-
PROC type.any.key()
INT any:
SEQ
write.full.string(screen, "Type any key to continue . . ")
keyboard ? any
newline(screen)
:
-
-- main programme for the matrix inversion
--
SEQ
- 1. preparation
-1.1 display the input data matrix
--
data(A,X)
type.any.key()
-- 1.2 assign order numbers to elements
-
row.col.to.num(num)
num.to.row.col(row, col)
SEQ
write.full.string(screen, "The order number of (i,j)th element")
newline(screen)
SEQ i=0 FOR (n+1)
SEQ
SEQ j}=0\mathrm{ FOR i
write.full.string(screen," ")
SEQ j=i FOR (np-i)
write.int(screen, num[i][j], 6)
newline(screen)
newline(screen)
type.any.key()
write.full.string(screen, "The row and column numbers for ith PE")
newline(screen)
SEQ i=0 FOR num.chan.ver
SEQ
write.int(screen, i, 6)
write.int(screen, row[i], 6)
write.imt(screen, col[i], 6)
newline(screen)
type.any.key()

* --
    - 2. systolic array processing
--

```
```

```
    east ! z; TRUE
```

```
```

    east ! z; TRUE
    ```
```

- 

--

```

PAR
-2.1 data input
\(-\)
PAR \(\mathrm{j}=0\) FOR np SEQ \(\mathrm{i}=0\) FOR n 2 pipe.vertical[i]! \(\mathrm{X}[\mathrm{i}][\mathrm{j}]\)
\(-\)
-- 2.2 systolic array
-
PAR \(\mathrm{i}=0\) FOR total.element VAL INT irow IS row[i]: VAL INT icol IS col[i]: IF
irow=icol
- 2.2.1 Circle Elements
circle.element(pipe.horizontal[i], pipe.vertical[i]) (icol<n) AND (irow \(>\) icol)
-2.2.2 Square Elements of Triangular Array
VAL INT ilef IS (i-1):
VAL INT idown IS num[irow+1][icol]: square.element.T(pipe.horizonta][ileft], pipe.horizontal[i], pipe.vertical[i], pipe.vertical[idown])
TRUE
- 2.2.3 Square Elements of Rectangular Array \(-\) VAL INT ileft IS (i-1):
VAL INT idown IS num [irow+1][icol]:
square.element.R(pipe.horizontal[ileft], pipe.horizontal[i], pipe.vertical[i], pipe.vertical[idown])
--
- 2.3 data sink for right

PAR \(\mathrm{i}=0\) FOR \(n\)
SEQ \(\mathrm{j}=0\) FOR n 2
VAL INT inum IS num[i][n2-1]:
REAL32 nothingl:
BOOL nothing2:
CHAN OF MV m.v IS pipe.horizontal[inum]:
m.v? nothingl; nothing2
-
-- 2.4 Receive the Result
-
PAR \(i=0\) FOR \(p\)
VAL INT inum IS num[n][n+i]:
REAL32 nothing 1:
BOOL nothing2:
CHAN OF REAL32 output IS pipe.vertical[inum]:
SEQ
SEQ \(=0\) FOR \(n\)
output? nothing 1
SEQ \(j=0\) FOR \(n\)
output ? A.output[j][i]
--
-3. Display the Result
--
newline(screen)
write.full.string(screen, "The inverse of the matrix \(A\) is:")
newline(screen)
SEQ \(i=0\) FOR \(n\)
SEQ
SEQ \(j=0\) FOR \(p\)
write.real32(screen, A.output[i][j], 8, 3)
newline(screen)
type.any.key()

\subsection*{5.9.2 Numerical Examples}
(a). Matrix Inverse
\(A\left[\begin{array}{lll}1 & 0 & 3 \\ 2 & 4 & 1 \\ 3 & 2 & 6\end{array}\right] \quad B\left[\begin{array}{lll}1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1\end{array}\right] \quad C\left[\begin{array}{lll}1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1\end{array}\right] \quad D\left[\begin{array}{lll}0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0\end{array}\right]\)
\[
X\left[\begin{array}{lll}
0.846 & 0.231 & 0.462 \\
0.577 & 0.115 & 0.269 \\
0.615 & 0.077 & 0.154
\end{array}\right]
\]
(b). Linear system
\(A\left[\begin{array}{llll}1 & 0 & 3 & 3 \\ 2 & 4 & 1 & 1 \\ 3 & 2 & 6 & 2 \\ 1 & 2 & 3 & 5\end{array}\right] \quad B\left[\begin{array}{lll}0 & 1 & 2 \\ 1 & 2 & 3 \\ 2 & 3 & 4 \\ 3 & 4 & 5\end{array}\right] \quad C\left[\begin{array}{llll}1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1\end{array}\right] \quad D\left[\begin{array}{lll}0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0\end{array}\right]\)
\(X\left[\begin{array}{lll}2.667 & 4.833 & 7.000 \\ 2.111 & 3.972 & 5.833 \\ 1.500 & 2.750 & 4.000 \\ 0.611 & 1.472 & 2.333\end{array}\right]\)

\section*{Chapter 6}

\section*{SYSTOLIC ARRAY FOR SCHUR COMPLEMENT COMPUTATION}

\subsection*{6.1 Introduction}

IN this chapter, we propose a systolic array to compute the Schur complement which often occurs in the domain decomposition method and other problems in Linear Algebra such as the numerical solution of elliptic partial differential equations [Golub89] and sparse linear systems [Ortega88].

The domain decomposition method can generally result in a block structured matrix system of a large size. To solve the system using the Gaussian elimination method, the computation of Schur component is required.

As the Schur complement computation involves matrix inversions and matrix-matrix multiplications, it always forms the bottleneck for the solution of the problems. Parallel computation of the Schur complement is therefore necessary in applications where problem sizes are so large that the sequential computing time is prohibitive or where real time environments are involved. A systolic array architecture can provide a good solution for parallel processing especially in matrix computing.

We will use the systolic array for linear and inverse matrix system (SALIMS) discussed in the last chapter as the core of new systolic array to compute the Schur complement iteratively [Evans93b].

In the next section, we begin from the discussion of the domain decomposition method in linear algebra and then introduce the Schur complement. Sec. 6.3 describes the systolic array algorithm for the Schur complement computation while Sec. 6.4 presents two possible implementation methods. Sec. 6.5 provides a systolic array architecture for the solution of the arrowhead matrix system. Sec. 6.6 analyses the performance of the systolic array including the efficiency and speedup. Finally a summary is given in Sec. 6.7.

\subsection*{6.2 Schur Component and Domain Decomposition}

We consider algebraic methods that rely only on the structure of the matrix system without exploiting too much knowledge about the problem. Assume that the domain \(\Omega\) can be divided into \(p\) subdomains \(\left\{\Omega_{i} \mid i=1,2, \ldots p\right\}\) which are not overlapping as shown in Figure 6.1. The remaining area outside the \(p\) subdomains is denoted as


Figure 6.1 Domain decomposition
\(\Gamma\). Because \(\Gamma\) separates the \(p\)
subdomains, it is sometimes called the separator. It is obvious that
\[
\begin{align*}
& \Gamma \cap\left(\bigcup_{i=1}^{p} \Omega_{i}\right)=\varnothing  \tag{6.1}\\
& \Omega=\Gamma \cup\left(\bigcup_{i=1}^{p} \Omega_{i}\right) \tag{6.2}
\end{align*}
\]

Firstly, we represent the unknowns in the subdomains \(\left\{\Omega_{i} \mid i=1,2, \ldots, p\right\}\) as \(\left\{x_{i} \mid i=1,2, \ldots, p\right\}\). Then we represent the unknowns in the separator as \(x_{s}\). The matrix representing the coupling of the unknowns \(x_{i}\) within the subdomain \(\Omega_{i}\) is named as \(A_{i}\) for \(i=1,2, \ldots, p\) respectively. There is no coupling between the unknowns of two different subdomains since they do not overlap. We denote the coupling between the unknowns of the subdomains \(\Omega_{i}\) and \(\Omega_{j}\) as
\[
C_{p}\left(\Omega_{i}, \Omega_{j}\right)=\left\{\begin{array}{ll}
A_{i} & \text { if } i=j  \tag{6.3}\\
0 & \text { if } i \neq j
\end{array} \quad(i, j=1,2, \ldots p)\right.
\]

Similarly, the coupling between the unknowns of the subdomain \(\Omega_{i}\) and the separator \(\Gamma\) can be given as follows
\[
\begin{equation*}
C_{p}\left(\Gamma, \Omega_{i}\right)=B_{i} \quad(i=1,2, \ldots, p) \tag{6.4}
\end{equation*}
\]
while the coupling between the unknowns of the separator \(\Gamma\) and the subdomain \(\Omega_{i}\) is given as
\[
\begin{equation*}
C_{p}\left(\Omega_{i}, \Gamma\right)=C_{i} \quad(i=1,2, \ldots p) \tag{6.5}
\end{equation*}
\]

The coupling of the unknowns within the separator is
\[
\begin{equation*}
C_{p}(\Gamma, \Gamma)=A \tag{6.6}
\end{equation*}
\]

Thus, we obtain a linear system of the following form
\[
\left[\begin{array}{ccccc}
A_{1} & & & & B_{1}  \tag{6.7}\\
& A_{2} & & & B_{2} \\
& & \ddots & & \vdots \\
& & & A_{p} & B_{p} \\
C_{1} & C_{2} & \cdots & C_{p} & A_{s}
\end{array}\right]\left[\begin{array}{c}
x_{1} \\
x_{2} \\
\vdots \\
x_{p} \\
x_{s}
\end{array}\right]=\left[\begin{array}{c}
b_{1} \\
b_{2} \\
\vdots \\
b_{p} \\
b_{s}
\end{array}\right]
\]

The coefficient matrix in the above equation is sometimes called the arrowhead or bordered block diagonal matrix according to its geometrical pattern. The system in Eq. (6.7) is a commonly used system in linear algebra. It can occur frequently in applications involving differential equations and banded systems.

To solve the system in a direct way, we need to eliminate \(\left\{x_{i} \mid i=1,2, \ldots p\right\}\) to get an equation solely on the separator. Thus, the Gaussian elimination process is executed on the system, so that the system finally becomes
\[
\left[\begin{array}{ccccc}
A_{1} & & & & B_{1}  \tag{6.8}\\
& A_{2} & & & B_{2} \\
& & \ddots & & \vdots \\
& & & A_{p} & B_{p} \\
& & & & \hat{A}_{s}
\end{array}\right]\left[\begin{array}{c}
x_{1} \\
x_{2} \\
\vdots \\
x_{p} \\
x_{s}
\end{array}\right]=\left[\begin{array}{c}
b_{1} \\
b_{2} \\
\vdots \\
b_{p} \\
\hat{b}_{s}
\end{array}\right]
\]
where
\[
\begin{align*}
& \hat{A}_{s}=A_{s}-\sum_{i=1}^{p} C_{i} A_{1}^{-1} B_{i}  \tag{6.9}\\
& \hat{b}_{s}=b_{s}-\sum_{i=1}^{p} C_{i} A_{1}^{-1} b_{i} \tag{6.10}
\end{align*}
\]

The matrix \(\hat{\boldsymbol{A}}_{s}\) in Eq. (6.9) is the so called Schur complement. The computation of the Schur complement consists of matrix inversions, matrix-matrix multiplications and additions.

Once we solve the following system for \(\boldsymbol{x}_{\boldsymbol{s}}\)
\[
\begin{equation*}
\hat{A}_{s} x_{s}=\hat{b}_{s} \tag{6.11}
\end{equation*}
\]
the remaining unknowns in the subdomains can be obtained by solving a series of smaller systems
\[
\begin{equation*}
A_{i} x_{i}=b_{i}-B_{i} x_{s} \quad(i=1,2, \ldots p) \tag{6.12}
\end{equation*}
\]

From the above discussion, we observe that there are two steps to solve the system in Eq. (6.7). The first step is to obtain the Schur complement \(\hat{\boldsymbol{A}}_{\boldsymbol{s}}\) as well as \(\hat{\boldsymbol{b}}_{s}\) in Eqs. (6.9) and (6.10) respectively. The second step is to solve a series of small linear systems in Eqs. \((6,11)\) and \((6.12)\).

In addition to the direct Gaussian elimination method, the problem can be also solved by an iterative method, for examples, the successive over relaxation (SOR) method and the conjugate gradient (CG) method. In order to make the CG method converge faster, an appropriate preconditioner is beneficial and this results in preconditioned conjugate gradient (PCG) method. The construction of the preconditioner still encounters the Schur complement computation, or an approximate Schur complement computation. More details on the construction of a preconditioner are referred to [Golub89]. This shows that the computation of Schur complement is very important in solving the system in Eq. (6.7) either by the direct Gaussian elimination method or by a preconditioned iterative method.

In most cases, the matrices \(\boldsymbol{A}_{1}, \boldsymbol{A}_{2}, \ldots, \boldsymbol{A}_{p}\) have the same dimension. In the following discussion, we assume that \(\boldsymbol{A}_{1}, \boldsymbol{A}_{2}, \ldots, \boldsymbol{A}_{p}\) are \(n\) by \(n\) nonsingular matrices. The matrix \(\boldsymbol{A}_{\boldsymbol{s}}\) is assumed as an \(s\) by \(s\)
matrix. Thus the matrices \(B_{1}, B_{2}, \ldots, B_{p}\) are \(n\) by \(s\) matrices while the matrices \(C_{1}, C_{2}, \ldots, C_{p}\) are \(s\) by \(n\) matrices.

In the following section, we propose a systolic algorithm to compute the Schur complement by recursively using the systolic array for linear and inverse matrix system developed in Chap. 5.

\subsection*{6.3 Systolic Array for Schur Complement Computation}

In this section, the Systolic Array for Schur Complement computation (SASCO) is designed. First we briefly review the systolic array for linear and inverse matrix system (SALIMS) developed in the last chapter. Then we use the SALIMS to accomplish the computation in Eq. (6.9) recursively. Finally we explain how \(\hat{b}_{s}\) in Eq. (6.10) is calculated.

\subsection*{6.3.1 Systolic Array for Linear and} Inverse Matrix System (SALIMS)

In the last chapter, we have developed the systolic array for linear and inverse matrix system [Wan93b] which is capable of performing the following matrix computation:
\[
\begin{equation*}
X=D-C A^{-1} B \tag{6.13}
\end{equation*}
\]

The SALIMS in Figure 5.5 is redrawn in Figure 6.2 with simplified input data format.

The SALIMS receives four input matrices, namely \(A, B, C\) and \(D\), from top side of the


Figure 6.2 Systolic array for linear and inverse matrix system
array. It performs the matrix computation in Eq. (6.13). The output matrix can be obtained at the bottom of the systolic array.

\subsection*{6.3.2 Systolic Array for Schur Complement Computation}

We rewrite Eq. (6.9) in a recursive form as
\[
\begin{align*}
& X_{0}=A_{i} \\
& X_{i}=X_{i-1}-C_{i} A_{i}^{-1} B_{i} \quad \text { for } i=1,2, \ldots p \tag{6.14}
\end{align*}
\]

After \(p\) iterations, the matrix \(\boldsymbol{X}_{\boldsymbol{p}}\) becomes the Schur complement in Eq. (6.9), i.e.,
\[
\begin{equation*}
\dot{A}_{s}=X_{p} \tag{6.15}
\end{equation*}
\]

By comparing Eq. (6.14) with Eq. (6.13), we find that Schur complement computation can be decomposed into \(p\) basic matrix operations in Eq. (6.13). Therefore it is possible to use the systolic array in Figure 6.2 to compute the Schur complement in a recursive way as shown in Eq. (6.14).

If the input data of the systolic array for the Schur complement computation (SASCO) is arranged in the way as in Figure 6.3 (where the skewed data format is ignored for simplicity) and the PEs recursively perform the operations as required by the SALIMS (see Figure 5.6), we can


Figure 6.3 Input data for SASCO obtain the Schur complement after \(p\) iterations at the bottom of the array.

Now the problem is how can we arrange the input data in the required way? For the triangular array, the input data come from entries of matrices \(\boldsymbol{A}_{1}, \boldsymbol{A}_{2}, \ldots, \boldsymbol{A}_{p}\) and \(C_{1}, C_{2}, \ldots, \boldsymbol{C}_{p}\) and it is convenient to arrange them in the required form. However, the input data for the rectangular array need the feedback of the output data from the bottom of the array and a special data interface is therefore required to arrange the data in the desired form.

Figure 6.4 shows the proposed interface. For the triangular array, the data interface receives the arranged data which is then sent to the array without any extra complexity. For the rectangular array, the data interface received the data both from the input data of the given matrices and from the output of the rectangular array. The interface then selects the desired input according to the processing phase or stage so that the data entering into the rectangular arrays takes the form as shown in Figure 6.3. The detailed definition of PEs will be given in a later section.

The adoption of data interface in Figure 6.4 allows the output data from the bottom of the rectangular array to


Figure 6.4 Input data for SASCO feedback to the input at the top of the rectangular array. Thus, a correct input data sequence to the processing array can be formed.

\subsection*{6.3.3 Computation of the Right Hand Side}

When we examine Eqs. (6.9) and (6.10), we find that \(\hat{\boldsymbol{A}}_{s}\) and \(\hat{b}_{s}\) have very similar expressions. We can simply attach one column of PEs at the right side of the systolic array to compute the right hand
side \(\hat{\boldsymbol{b}}_{\boldsymbol{s}}\). The PEs attached perform the exact same function as those in the rectangular array. The input data to the attached PEs have a same format to the input data to the rectangular array in Figure 6.4. The systolic architecture which is able to compute both \(\hat{A}_{s}\) and \(\hat{b}_{s}\), with attached PEs is shown in Figure 6.5.


Figure 6.5 Attached PEs for computation of the right hand side

The systolic array in Figure 6.5 can compute the Schur complement \(\hat{\boldsymbol{A}}_{s}\) in Eq. (6.9) as well as the right hand side \(\hat{b}_{s}\) in Eq. (6.10) in the same time. If the right hand side is of multiple columns, we can attach multiple columns of PEs at the right side of the systolic array.

\subsection*{6.4 Solution of the Arrowhead Matrix System}

In Sec. 6.3, we have designed the systolic array to perform the Schur complement computation. There is still the task left for solving the linear systems in Eqs. (6.11) and (6.12). Recalling that the systolic array in Figure 6.2 can be used to solve linear system, we can implement the computations in

Eqs. (6.11) and (6.12). We first assume that the matrices \(\boldsymbol{A}_{1}, \boldsymbol{A}_{2}, \ldots, \boldsymbol{A}_{p}\) and \(\boldsymbol{A}_{s}\) have same dimension, i.e., \(n=s\). We add more PEs at the bottom of the array to complete the tasks given in Eqs. (6.11) and (6.12). The resulting processing array is shown in Figure 6.6.

After the Schur complement computation, the systolic array solves the linear system in Eq. (6.11). The Schur complement \(\hat{\boldsymbol{A}}_{s}\) and a unit matrix \(\boldsymbol{I}\) are input to the triangular array and \(\hat{b}_{s}\) is sent to the attached PEs at right. The solution of Eq. (6.11), - \(\boldsymbol{x}_{s}\), is sent to the attached PE at bottom (represented by shaded squares) and it is stored in the individual PEs for later processing.

Subsequently, the linear systems in Eq. (6.12) can be solved. This can be conceptually showed as follows. The systolic array receives \(A_{1}, B_{1}\) and \(b_{1}\) through the data interface at top. Then the array computes \(-A_{1}^{-1} B_{1}\) and \(-A_{1}^{-1} b_{1}\) which are sent to the


Figure 6.6 Systolic array for the solution of the arrowhead matrix system attached PE array at bottom. The attached PE array at bottom perform the computation of matrix multiplication-and-addition. Thus it generates an output matrix of \(-x_{1}=-A_{1}^{-1} b_{1}+A_{1}^{-1} B_{1} x_{s}\). This procedures is repeated until all the systems in Eq. (6.12) are solved.

In the case of a single right hand side system to be solved, the shaded square PEs can be combined with the interface PEs. The interface PEs for this alternation are required to be communicated
horizontally and \(-x_{s}\) should be sent to the most left interface PE for the rectangular array in the middle part.

If \(s>n\), the proposed array can still solve the system in Eq. (6.11). However, partitioning of a larger system (of order \(s\) ) is needed in order to be solved in a smaller array (of order \(n\) ). The introduction of partitioning mechanism also necessitates more sophisticated data interface.

\subsection*{6.5 Two Possible Implementation Methods}

In this section, we present two possible methods to implement the systolic algorithm discussed above. In the first method, each PE is programmed such that it knows the every function to be performed during each clock tick before the input data arrives. We call this method an internal control method. In the second method, we programme each PE with only the basic operations, e.g., inner product step (ips), division, while the outside world (host processor) provides control signals to tell the PE which function is active during the given clock tick. This method is termed an external control method.

\subsection*{6.5.1 Internal Control Method}

We define the PEs for SASCO in the middle of Figure 6.7. The PE definition can be further extended to include the function for solving the linear systems in Eqs (6.11) and (6.12). The definition specifies what the PEs will do in every clock tick (or time step). The PEs know exactly how many steps will be run and what kind of operation will be performed in every time step. All the iteration loops are inside the PEs and each PE then has a more complex program structure. This requires each PE has a degree of programmability. In addition, some global parameters, for example, \(n, s\) and \(p\), should be sent to each PE before the PE begins to perform its operation.

The square PEs and the circle PEs perform useful operations required by the Schur complement computation while the rectangular PEs (both unshaded and shaded) are performing data selecting and transferring tasks. The shaded square PEs employed in solving the arrowhead matrix system in Figure 6.6 can be defined in a similar way although it is not given in Figure 6.7.
\begin{tabular}{|c|c|c|c|}
\hline  &  & \(x\).in
\[
1
\]
\[
x . o u t
\] &  \\
\hline \multicolumn{4}{|c|}{PE definition for the internal control method} \\
\hline ```
for j=1:p
    initialisation:
    x=0
    LU decomposition:
    for i=1:n
        if |x.in|> |x 
        v=TRUE
        m=.x/x.in; x=x. in
        else
        v=FALSE
        m=-x.in/x
        end
    end
    Backsubstitution
    for i=1:s
        m=x.in/x
    end
end
``` & ```
for j=1:p
    initialisation:
    x=0
    LU decomposition:
    for }i=1:
        if v
        x.ouf=x+m*x.in
        x=x.in
        else
        x.ouf=x.in+m*x
        end
    end
    Backsubsilution
    for }i=1:
        x.ont=x.in-m*x
    end
end
``` & \[
\begin{aligned}
& \text { for } j=1: p \\
& \text { for } i=1:(n+s) \\
& m=x \cdot \ln / x \\
& \text { end } \\
& \text { end }
\end{aligned}
\] & \[
\begin{aligned}
& \text { Input Bl and As: } \\
& \text { for } i=1:(n+s) \\
& x . o u t=x . i n \\
& x=0 \\
& \text { for } j=2: p \\
& \text { for } i=1: n \\
& x . o w t=x . i n \\
& \text { sink }=\text { in } \\
& \text { end } \\
& \text { Feedhack: } \\
& \text { for } i-1: s \\
& x . o u t=z . i n \\
& \text { end } \\
& \text { end }
\end{aligned}
\] \\
\hline \multicolumn{4}{|c|}{\(P E\) definition for the external control method} \\
\hline ```
if mode \(=0\)
    initialisation:
    \(x=0\)
elseif mode \(=1\)
    \(L U\) decomposition:
    if \(\mid x\), in \(|>|x|\)
        \(v=T R U E\)
        \(m=-x / x . i n ; x=x . i n\)
    else
        \(\nu=F A L S E\)
        \(m=-x . i n / x\)
    end
elseif mode \(=2\)
    Backsubstitution
    \(m=x \cdot i n / x\)
end
``` & ```
if mode=0
    initialisation:
    x=0
elseif mode=1
    LU decomposition:
    ifv
        x.out=x+m*x.in
        x==x. in
    else
        x.out=x.in+m*x
    end
elseif mode=2
    Backsubstitution
    r.out=x.in-m*x
end
``` & .r.out=x.in & ```
if mode \(=0\)
    Input BI and As
    \(x\).out \(=x\).in
elseif mode \(=1\)
    Switch to Bi:
        x.out \(=x\). in
        \(\sin k\) zin
    elseif mode \(=2\)
    Switch to feedback
    \(x .0 u t=s . i n\)
end
``` \\
\hline
\end{tabular}

Figure 6.7 PE definition for the internal and external control methods

\subsection*{6.5.2 External Control Method}

Another method for the implementation of the proposed systolic array is to define the PEs in the bottom of Figure 6.7. We only define the basic function of the PEs. Which functions PEs select
during the given time step is dependent upon the external control signals. How many loops the PEs will perform is determined by the external control signals as well. The PEs obtain such information only from the external control signals. This may provide flexibility to change the size of problems to be solved. However, it transfers the complexity to the host processor which provide the control signals for all PEs in the systolic array.

The external control signals can enter the systolic array accompanying with the input data. If we have a serial communication channel, we can add two extra bits to the front of each input data to indicate the three processing modes given in Figure 6.7. If we have, on the other hand, a parallel communication channel, we can add two separate control lines to signal the three processing modes.

When the systolic array in Figure 6.6 is considered, more processing modes are required. This can be implemented with 3-bit control signals which are able to specify up to 8 processing modes.

\subsection*{6.5.3 The Comparison of the Two Methods}

The internal control method needs no external control signals and this will simplify the data format while the external control method needs external control signal which are provided by the host. The first method needs global parameters to be sent to the PEs in the array before the computation tasks are executed. The second method eliminates the global parameters in the PEs and this allows the extension of the array to a bigger problem to be easier.

The first method is suitable for the soft-systolic and semi-hard systolic [Evans91] implementation such as programmable processor array (e.g., Transputer network, WARP machine) while the second one is more suitable for hard-systolic implementation such as chip level implementation.

\subsection*{6.6 Performance Analysis}

In this section, we analyse the performance of the proposed systolic array for the Schur complement computation as shown in Figure 6.4. The proposed array employs \(n(n+1) / 2+s n\) processing elements to compute the Schur complement. In addition to these PEs, there are \(n+s\) interface PEs. As the interface PEs are very simple, we do not count them into the total number of PEs. The circle PEs and
square PEs approximately have the same computational complexity during each clock tick. The computational complexity is more or less that of an inner product step (ips). We use the term flop to refer to an inner product step or the operation with similar complexity such as division plus comparison in the circle PEs.

Now we consider the computational load of the Schur complement computation in Eq. (6.9). In each iteration, it requires to perform a task as given in Eq. (6.13) which is composed of an LU decomposition for an \(n \times n\) matrix, a forward substitution \(L_{i}^{-1} \boldsymbol{P}_{i} \boldsymbol{B}_{i}\), a backward substitution including \(C_{i} \boldsymbol{U}_{i}^{-1}\) and the matrix-matrix multiplication \(\left(C_{i} \boldsymbol{U}_{i}^{-1}\right)\left(\boldsymbol{L}_{i}^{-1} P_{i} B_{i}\right)\). These operations requires \(n^{3} / 3\) flops, \(n^{2} s / 2\) flops, \(n^{2} s / 2\) flops and \(n s^{2}\) flops respectively. Then it needs \(p\) iterations to obtain the Schur complement. Therefore it requires a total computational load as follows
\[
\begin{equation*}
L=\left(\frac{1}{3} n^{3}+n^{2} s+n s^{2}\right) p \tag{6.16}
\end{equation*}
\]

It requires \((n+s)\) clock ticks for each iteration. By considering the skewed input data format, the number of total computation time becomes
\[
\begin{equation*}
T=p(n+s)+(n+s-1)+(n-1)=(p+1)(n+s)+n-2 \tag{6.17}
\end{equation*}
\]

The speedup of the systolic array is the ratio between the time required for execution of the parallel algorithm and the time required by one processor. Eq. (6.16) gives the time required by one processor and Eq. (6.17) gives the time required by the systolic array. Thus, the speed up is as follows
\[
\begin{equation*}
S=\frac{L}{T}=\frac{\left(\frac{1}{3} n^{3}+n^{2} s+n s^{2}\right) p}{(p+1)(n+s)+n-2} \tag{6.18}
\end{equation*}
\]

The efficiency of the systolic array then can be given as
\[
\begin{equation*}
E=\frac{S}{\frac{1}{2} n(n+1)+n s} \tag{6.19}
\end{equation*}
\]

We consider the case where \(p \cdots 1\) and \(n=s\). The speedup and the efficiency then become
\[
\begin{equation*}
S=\frac{L}{T}=\frac{\left(\frac{1}{3} n^{3}+n^{2} s+n s^{2}\right) p}{(p+1)(n+s)+n-2} \approx \frac{\frac{7}{3} n^{3} p}{2} p n=\frac{7}{6} n^{2} \tag{6.20}
\end{equation*}
\]
\[
\begin{equation*}
E=\frac{S}{\frac{1}{2} n(n+1)+n s} \approx \frac{\frac{7}{6} n^{2}}{\frac{3}{2} n^{2}}=\frac{7}{9}=77.8 \% \tag{6.21}
\end{equation*}
\]

This shows that the proposed systolic array is very efficient to compute the Schur complement in Eq. (6.9). The speedup of the systolic array is significant when \(n\) is large. Similar performance result can be obtained for the systolic array with attach PEs to compute the right hand side.

Similarly, we can analyse the performance of the systolic array for the solution of the arrowhead matrix system as shown in Figure 6.6.

\subsection*{6.7 Summary}

The computation of the Schur complement in the domain decomposition method often forms a bottleneck to the problem of solving the large sparse linear system which occur in the Finite Element Method (FEM). We have in this chapter designed systolic arrays for Schur complement computation and the arrowhead matrix system.

The proposed systolic for the Schur complement computation employs \(n(n+1) / 2+n s\) PEs and accomplishes the Schur computation in \((p+1)(n+x)+n-2\) clock ticks. The systolic array enjoys a pipelining rate of 1 and achieves a high efficiency to compute the Schur complement. In the case where \(n=s\), the speedup is about \(7 n^{2} / 6\) and the efficiency is about \(77.8 \%\). With an appropriate value of \(n\), we can obtain a significant speedup. This can greatly reduce the time required for solving the linear system.

We have also designed systolic array to solve the arrowhead matrix system which arises in the domain decomposition method.

The designed systolic arrays for the Schur complement computation and the arrowhead matrix system solution may be helpful in solving large linear system arising from the finite element discretization of elliptic partial differential equations in the domain decomposition method.

\section*{Chapter 7}

\section*{SYSTOLIC ARRAY FOR QR DECOMPOSITION OF BLOCK STRUCTURED SPARSE SYSTEMS}

\subsection*{7.1 Introduction}

IN this chapter, we present a systolic array architecture for the QR decomposition of block structured sparse systems ( \(\mathrm{BS}^{3}\) ) [Wan94a] which find applications in areas such as photogrammetry [Golub79], Doppler radar positioning [Manneback85] and geodetic survey problems [Golub80].

Some of the block structured sparse systems arising in the above-mentioned areas may be extremely large in size. For example, the problem of least squares adjustment of coordinates of the geodetic stations comprising the North American Datum consists of about six million equations in 400,000 unknowns. In such problems, the acceleration of the solution of the system, particularly the QR decomposition, is very important. Systolic array is one of the suitable candidatesto perform the acceleration of problem solving by using multiprocessing.

In this chapter, we present a systolic array architecture to compute the QR decomposition of the block structured sparse systems. The QR decomposition of the block structured sparse system is broken into a certain number of QR decomposition of smaller size. Then the proposed systolic architecture, based on the Gentleman and Kung's systolic array for matrix triangularization [Gentleman8 1], computes the smaller size QR decompositions recursively to derive the QR decomposition of the block structured sparse system.

The rest of this chapter is organised as follows. In the next section, we will have a brief introduction to the block structured sparse systems (which has a dual block angular form) and some of their applications. Sec. 7.3 develops the QR decomposition algorithm of the block structured sparse matrix.

In Sec. 7.4, we design a systolic array to compute the QR decomposition. Sec. 7.5 deals with the simulation of the systolic algorithm by the parallel programming language OCCAM 2 in a single transputer system. Finally, Sec. 7.6 contains a summary of the chapter.

\subsection*{7.2 Block Structured Sparse System}

We discuss a general procedure, called substructuring or dissection to obtain a block structured sparse system ( \(\mathrm{BS}^{3}\) ). As an example, we consider a geodetic position network consisting of geodetic stations connected through observations. Each station corresponds to a set of unknown coordinates. In geodetic problems, the idea of breaking down


Figure 7.1 Dissection of a geodetic problem a problem into geographically defined subproblems connected in a well-defined way has been applied for a long time [Björck90]. The idea is to choose a set of stations \(\boldsymbol{B}\), which separates the other stations into two regional blocks \(\boldsymbol{A}_{1}\) and \(\boldsymbol{A}_{2}\) so that stations in \(\boldsymbol{A}_{1}\) are not connected by observations to stations in \(\boldsymbol{A}_{2}\). We then order the station variables so that those in \(\boldsymbol{A}_{1}\) appear first, those in \(\boldsymbol{A}_{2}\) second, and those in \(\boldsymbol{B}\) last. Finally, we order the equations so that those including \(\boldsymbol{A}_{1}\) come first and those including \(\boldsymbol{A}_{2}\) come last. The blocking of the region is depicted in Figure 7.1 and the corresponding structure of the observation matrix can be expressed as
\[
A=\left[\begin{array}{lll}
A_{1} & & B_{1}  \tag{7.1}\\
& A_{2} & B_{2}
\end{array}\right]
\]

More generally, the structure of the observation matrix has the form as follows:
\[
A=\left[\begin{array}{ccccc}
A_{1} & & & & B_{1}  \tag{7.2}\\
& A_{2} & & & B_{2} \\
& & \ddots & \ddots & \vdots \\
& & & A_{M} & B_{M}
\end{array}\right]
\]
where \(A \in \Re^{m_{\Sigma} \cdot n_{\Sigma}}, A_{i} \in \mathscr{M}^{m_{i} \cdot n_{t}}\), and \(B_{i} \in \Re^{m_{1} \cdot n_{M}}\) for \(i=1,2, \ldots, M\). We call this matrix a dual block angular form matrix.

We also partition the unknown vector \(\boldsymbol{x}\) and right-hand side \(\boldsymbol{b}\) conformally with Eq. (7.2), then we have
\[
\begin{gather*}
x^{T}=\left(x_{1}^{T}, \cdots, x_{M}^{T}, x_{M+1}^{T}\right)  \tag{7.3}\\
b^{T}=\left(b_{1}^{T}, b_{2}^{T}, \cdots, b_{M}^{T}\right) \tag{7.4}
\end{gather*}
\]

Then the corresponding linear system can be expressed as
\[
\begin{equation*}
A x=b \tag{7.5}
\end{equation*}
\]

Some examples where the block structure in Eq. (7.2) arises naturally are in photogrammetry [Golub79], Doppler radar positioning [Manneback85] and geodetic survey problem [Golub80]. On the other hand, a general sparse matrix can be permuted into this form [Weil71]. More application areas of such systems include the molecular structure problem, gravity field of the earth, cluster analysis and pattern matching.

In many cases, we often need to find the linear least squares solution of the following problem:
\[
\begin{equation*}
\underset{x}{\min }\|A x-b\|_{2} \tag{7.6}
\end{equation*}
\]

In order to find the solution of the system in Eq. (7.6), we can use an orthogonalization method. The QR decomposition method is one of the most widely used methods with good numerical performance.

We assume that \(m_{\Sigma} 2 n_{\Sigma}\) and \(\operatorname{rank}(A)=n_{\Sigma}\). Then there is an orthogonal matrix \(Q \in \Re^{m_{\Sigma} \cdot m_{\Sigma}}\) such that
\[
\begin{equation*}
Q^{T} A=\binom{R}{0} \tag{7.7}
\end{equation*}
\]
where \(R\) is upper triangular with nonnegative diagonal elements. The decomposition in the above equation is called the QR decomposition of \(A\).

We let
\[
\begin{equation*}
Q^{T} b=\binom{c_{1}}{c_{2}} \tag{7.8}
\end{equation*}
\]
and notice that \(Q\) is orthogonal, the least squares problem in Eq. (7.6) can then be rewritten as
\[
\begin{equation*}
\|A x-b\|_{2}^{2}=\left\|Q^{T} A x-Q^{T} b\right\|_{2}^{2}=\left\|R x-c_{1}\right\|_{2}^{2}+\left\|c_{2}\right\|_{2}^{2} \tag{7.9}
\end{equation*}
\]

Obviously the residual norm is minimised by solving the upper triangular system
\[
\begin{equation*}
x=R^{-1} c_{1} \tag{7.10}
\end{equation*}
\]
and its minimum equals the norm of \(c_{2}\).

Therefore, the QR decomposition is the key stage of the orthogonalization method for least squares problems. If the size of the problem is large, the QR decomposition requires a large amount of computations and takes a long time. An example given in [Björck90] shows that the problem of least squares adjustment of coordinates of the geodetic stations comprising the North American Datum consists of about six million equations in 400,000 unknowns. In such problems, the acceleration of the solution of the system, particularly the QR decomposition, is very important.

\subsection*{7.3 QR Decomposition of the \(\mathrm{BS}^{3}\)}

We assume that the submatrices \(\boldsymbol{A}_{1}, \boldsymbol{A}_{2}, \cdots, \boldsymbol{A}_{M}\) all have the same size and the submatrices \(\boldsymbol{B}_{1}, \boldsymbol{B}_{2}, \cdots, \boldsymbol{B}_{M}\) all have the same size (but different from the size of submatrices \(\boldsymbol{A}_{1}, \boldsymbol{A}_{2}, \cdots, \boldsymbol{A}_{M}\) ), that is, \(\boldsymbol{A}_{i} \in \Re^{m \times n}, \boldsymbol{B}_{i} \in \Re^{m \times p}\) for \(i=1,2, \ldots, M\). We also presume that \(m \geq n\) and \(\operatorname{rank}\left(\boldsymbol{A}_{i}\right)=n\) for \(i=1,2, \ldots, M\). Now we describe the QR decomposition algorithm in the following steps:

Step 1. Reduce the diagonal block \(\boldsymbol{A}_{i}\) to upper triangular form by a sequence of orthogonal transformations, and apply these also to the blocks \(B_{i}\) and the right-hand side blocks \(b_{i}\), for \(i=1,2\), ..., \(M\), yielding
\[
Q_{i}^{T}\left(A_{i}, B_{i}\right)=\left(\begin{array}{cc}
R_{i} & S_{i}  \tag{7.11}\\
0 & T_{i}
\end{array}\right)
\]

Step 2. Reorder the equations in the reduced system by ordering first the rows corresponding to \(\boldsymbol{R}_{i}\), for \(i=1,2, \ldots, M\), and lastly the rows corresponding to \(\boldsymbol{T}_{i}\), for \(i=1.2, \ldots . M\). Form the reduced matrix \(T\) and the right-hand side \(\boldsymbol{l}\)
\[
T=\left(\begin{array}{c}
T_{1}  \tag{7.12}\\
T_{2} \\
\vdots \\
T_{M}
\end{array}\right), \quad d=\left(\begin{array}{c}
d_{1} \\
d_{2} \\
\vdots \\
d_{M}
\end{array}\right)
\]
where \(\quad \boldsymbol{T}_{i} \in \mathscr{H}^{(m-n) \vee p}\) for \(i=1,2, \ldots, M\) and \(\quad T \in \mathscr{H}^{M(m-n) \vee p}\).

Step 3. Compute the QR decomposition of \(T\) and transform the vector \(\boldsymbol{I}\)
\[
\begin{equation*}
\varrho_{M+1}^{T} T=\binom{R_{M+1}}{0}, \quad Q_{M+1}^{T} d=\binom{c_{M+1}}{d_{M+1}} \tag{7.13}
\end{equation*}
\]
(Step 4). For the least squares problem, the residual norm of the system is given by \(\left\|d_{M+1}\right\|_{2}\). The solution can be obtained by solving the following triangular system:
\[
\begin{align*}
& R_{M+1} x_{M+1}=c_{M+1} \\
& R_{i} x_{i}=c_{i}-S_{i} x_{M+1} \quad(i=1,2, \cdots, M) \tag{7.14}
\end{align*}
\]

In the above algorithm, it is required to compute the QR decompositions of the \(M\) submatrices \(\boldsymbol{A}_{1}, \boldsymbol{A}_{2}, \cdots, \boldsymbol{A}_{M}\), and the orthogonal transformations on the \(M\) submatrices \(\boldsymbol{B}_{1}, \boldsymbol{B}_{2}, \cdots, \boldsymbol{B}_{M}\), and the QR decomposition of the \(M(m-11)\)-by-p matrix \(T\). Of course, the same transformations are also required for the right-hand side.

In the next section, a systolic array architecture is designed to perform all the computations mentioned above

\subsection*{7.4 Systolic Array Design}

From the discussion in the last section, we know that in order to obtain the QR decomposition of the sparse matrix, we require to perform the computations in Eq. (7.11) iteratively and then perform the computation in Eq. (7.13).

In this section, we give a brief introduction to the widely used triangular systolic array introduced by Gentleman and Kung [Gentleman81]. Then, we design a new systolic array based on the Gentleman and Kung's array.

\subsection*{7.4.1 Gentleman and Kung's Systolic Array for QR decomposition}

The systolic array introduced by Gentleman and Kung has a triangular geometry. The input data enter the array from the top of the array. After all the data has passed through the array, the entries of the upper triangular array are stored in the individual processing elements (PEs) of the array as shown in Figure 7.2. The PE definitions are shown in Figure 7.3. A small difference to the Gentleman and Kung's PE definition has been introduced by the circle PEs in order to ensure the numerical stability of the QR decomposition. We add a statement to examine if the input \(x\).in and the stored variable \(r\) are zeros, or more exactly if their absolute values are less than a small number, eps. If so, we assign 1 and 0 to the variables \(c\) and \(s\) respectively. The circle PEs then perform 2 multiplications and 2 divisions and one square root operation required by the


Figure 7.2 Systolic array for QR decomposition

Givens Generation. The square PEs receive data from north and west, and then perform 4 multiplications required by the Givens Rotation.


Figure 7.3 PE definition for QR decomposition

This systolic array has been used in many applications in signal processing such as adaptive filtering and adaptive beamforming. More details on this systolic array design are given in Chap. 4.

\subsection*{7.4.2 Systolic Array for the Computations in Eq. (7.11)}

In Eq. (7.11), we compute not only the QR decomposition on the matrix \(\boldsymbol{A}_{i}\) but also the same orthogonal transformation on the matrix \(\boldsymbol{B}_{\boldsymbol{i}}\). This can be done by attaching a rectangular array at the right side of the systolic array in Figure 7.2. If the right-hand side is required to process, we can simply add another column of PEs. Figure 7.4 shows the extended systolic array for performing the computations in Eq. (7.11). The PE definition is same as that in Figure 7.3. The systolic array in Figure 7.4 can perform the following computation:
\[
Q_{i}^{T}\left(A_{i} B_{i} b_{i}\right)=\left(\begin{array}{ccc}
R_{i} & S_{i} & c_{i}  \tag{7.15}\\
0 & T_{i} & d_{i}
\end{array}\right) \quad \underset{(m-n)}{(n)}
\]

The input data is now expanded to the entries of \(\boldsymbol{A}_{i}, B_{i}\) and \(b_{i}\). They enter the systolic array from the top. The result matrices \(\boldsymbol{R}_{i}, \boldsymbol{S}_{\boldsymbol{i}}\) and \(\boldsymbol{c}_{\boldsymbol{i}}\) are stored in the PE array. The matrices \(\boldsymbol{T}_{\boldsymbol{i}}\) and \(\boldsymbol{d}_{i}\) are piped out of the bottom of the systolic array. Notice that some zeros accompany with the matrices \(\boldsymbol{T}_{i}\) and \(\boldsymbol{d}_{i}\). The number of the rows of zeros is equal to 11 .


Figure 7.4 Extended systolic array for orthogonalization operation

We can use this systolic array in an iterative way so that we can obtain all the matrices of \(\boldsymbol{R}_{1}, \boldsymbol{R}_{2}, \cdots, \boldsymbol{R}_{M}\), \(\boldsymbol{S}_{1}, \boldsymbol{S}_{2} \cdots, \boldsymbol{S}_{M}, \boldsymbol{c}_{1}, \boldsymbol{c}_{2}, \cdots, \boldsymbol{c}_{M}, \boldsymbol{T}_{1}, \boldsymbol{T}_{2} \cdots, \boldsymbol{T}_{M}\) and \(\boldsymbol{d}_{1}, \boldsymbol{d}_{2}, \cdots, \boldsymbol{d}_{M}\). The submatrices \(\boldsymbol{A}_{1}, \boldsymbol{A}_{2}, \cdots, \boldsymbol{A}_{M}, B_{1}, \boldsymbol{B}_{2}, \cdots, B_{M}\) and \(b_{1}, b_{2}, \cdots, b_{M}\) as input enter the systolic array from the top sequentially and the output data flow out at the bottom.

\subsection*{7.4.3 Systolic Array for the Computation in Eq. (7.13)}

Notice that the computation in Eq. (7.13) is a QR decomposition of the matrix \(T\) as well as the transformation on the vector \(d\). Thus we can use the systolic array with the same structure as shown in Figure 7.2 with an attached column of PEs for right-hand side processing to perform the computational tasks.

From the observation on the output of the systolic array in Figure 7.4, we know that the submatrices \(\boldsymbol{T}_{1}, \boldsymbol{T}_{2}, \cdots, \boldsymbol{T}_{M}\) do not come out in the same way as in Eq. (7.12). Instead some zeros are inserted between the adjacent submatrices. In matrix form, we obtain the output as
\[
\dot{\boldsymbol{T}}=\left(\begin{array}{c}
0  \tag{7.16}\\
T_{1} \\
0 \\
T_{2} \\
\vdots \\
0 \\
T_{M}
\end{array}\right), \quad \dot{d}=\left(\begin{array}{c}
0 \\
d_{1} \\
0 \\
d_{2} \\
\vdots \\
0 \\
d_{M}
\end{array}\right)
\]

The rearrangement of the matrix \(\dot{\boldsymbol{T}}\) can form the matrix \(\boldsymbol{T}\) in order to complete the computation in Eq. (7.13). However, the rearrangement will break up the data pipeline structure in the systolic array. From the fact that a row of zeros entering into the systolic arrays in both Figure 7.2 and Figure 7.4 will not change any desired results, we can allow those zeros in Eq. (7.16) to enter the array while we can still obtain the same result. In the next subsection, we will discuss the overall architecture of the systolic array for the QR decomposition of \(\mathrm{BS}^{3}\) by employing the systolic arrays in Figure 7.2 and Figure 7.4.

\subsection*{7.4.4 The Overall Structure of the Systolic Array Architecture for \(B S^{\circ}\)}

\subsection*{7.4.4.1 The systolic array architecture}

The overall systolic array architecture is shown in Figure 7.5 where \(m=5, n=3, p=4\), and \(M=3\).

The submatrices \(\boldsymbol{A}_{1}, \boldsymbol{A}_{2}, \cdots, \boldsymbol{A}_{M}\), \(\boldsymbol{B}_{1}, \boldsymbol{B}_{2}, \cdots, \boldsymbol{B}_{M}\) and \(\boldsymbol{b}_{1}, b_{2}, \cdots, \boldsymbol{b}_{M}\) enter into the systolic array from the top side. The parts 1-3 PEs in the systolic array perform the computations in Eq. (7.15) recursively for all the submatrices. This has been detailed in Figure 7.4. The resulting submatrices \(\boldsymbol{R}_{1}, \boldsymbol{R}_{2}, \cdots, \boldsymbol{R}_{M}, \quad \boldsymbol{S}_{1}, \boldsymbol{S}_{2}, \cdots, \boldsymbol{S}_{M}\) and \(c_{1}, c_{2}, \cdots, c_{M}\) are stored in the PEs


Figure 7.5 The overall architecture of the systolic array for \(\mathrm{BS}^{3}\) in part 1, part 2 and part 3 respectively.

The two matrices \(\tilde{\boldsymbol{T}}\) and \(\tilde{\boldsymbol{d}}\) as the results of the upper part of the systolic array are passed to the lower part of the systolic array. The parts 4-5 PEs perform the computation in Eq. (7.13) and the resulting matrices \(R_{M+1}\) and \(c_{M+1}\) are stored in the PEs in part 4 and part 5 respectively.

From the above discussion, we know that each PE in parts \(1-3\) is required to store \(M\) results. This indicates that each PE in parts 1-3 should accommodate sufficient memories to store the M results. The PEs in part 4-5 only store one result each and they are therefore simpler. Extra time is required for emptying the final results from the systolic array.

\subsection*{7.4.4.2 The PE definition}

The definition of the PEs is very straightforward because the upper part of the systolic array performs a number of \(Q R\) decompositions recursively while the lower part perform a single \(Q R\) decomposition.

\section*{The PEs in parts 1 to 3 can be defined in the following way:}

Circle PEs in parts 1-3:
\[
\begin{aligned}
& \text { for } i=1 \text { to } M \\
& \qquad \begin{array}{l}
r(i)=0 \\
\text { for } j=1 t 1 o m \\
\text { if }|x . i m|<e p s \text { and }|r(i)|<e p s \\
c=1, s=0 \\
e l s e
\end{array} \quad \begin{array}{l}
t=s q r t\left(x . i m^{2}+r(i)^{2}\right) \\
c=r(i) / t, \quad s=x . i n / t, r(i)=t \\
e n d
\end{array} \\
& \text { end } \\
& \text { end }
\end{aligned}
\]

Square PEs in parts 1-3:
for \(i=11 \circ \mathrm{M}\)
\(r(i)=0\)
for \(j=1\) to m
\(x . o m t=c \cdot x . i m-s \cdot r(i)\)
\(r(i)=c \cdot r(i)+s \cdot x \cdot i n\)
end
cind

The PEs in parts 4 and 5 can be defined in the following way:

Circle PEs in parts 4-5:
for \(i=1\) to Mm
\[
\begin{aligned}
& r=0 \\
& \text { if }|x . i m|<e p s \text { and }|r|<e p s \\
& \quad c=l, s=0 \\
& \text { else } \\
& \quad t=\operatorname{sqr} t\left(x . i m^{2}+r^{2}\right) \\
& c=r / t, s=x . i m / t, r=t
\end{aligned}
\]
end
end

\subsection*{7.5 Simulation of the Systolic Algorithm}

We implement the systolic algorithm for QR decomposition of the block structured sparse system on a single transputer system using the parallel programming language OCCAM 2.

\subsection*{7.5.1 Simulation Method}

The OCCAM 2 programming language was designed for the transputer, a microprocessor which can perform as a building block of an interconnected network. Many different network topologies may be created with a system of transputers by connecting the four communication links of each transputer in different ways. The parallel algorithms developed in a single transputer by the OCCAM 2 language can be directly configured or mapped to transputer networks. Thus the programming for the parallel architectures in transputers usually begins with the parallel architecture simulation in a single transputer system. After the parallel algorithms have been tested in a single transputer system, the algorithms may be distributed to a transputer network for parallel operation.

We simulate the Systolic algorithm by OCCAM 2 in a single transputer system. We outline the main structure of the simulation algorithm as following:
```

PAR
... send data to systolic array
PAR i=0 FOR number.of.PEs
IF
The PE is in diagonal
IF
The PE is in upper part of the array
circle.PE.up -- circle PEs in part 1-3
TRUE
circle.PE.low -- circle PEs in part 4-5
TRUE
IF
The PE is in upper part of the array
square.PE.up -- circle PEs in part 1-3

```

TRUE
square. PE.low -- circle PEs in part 4-5
... data sink for right -- consume the unwanted data
... data sink for bottom
... get result

\subsection*{7.5.2 Simulation Result}

Below is a numerical example to test the systolic algorithm, where \(M=3, m=5, n=3, p=4\). The input data can be written in a compact form as
\[
\left[\begin{array}{lll}
A_{1} & B_{1} & b_{1}  \tag{7.17}\\
A_{2} & B_{2} & b_{2} \\
A_{3} & B_{3} & b_{3}
\end{array}\right]
\]

The above matrix has the values of
\begin{tabular}{rrrrrrrr}
1.0 & 2.0 & 1.0 & 1.0 & -1.0 & -1.0 & 1.0 & 4.0 \\
0.0 & 1.0 & 0.0 & -1.0 & 1.0 & 1.0 & -1.0 & 1.0 \\
0.0 & 0.0 & 1.0 & -1.0 & -1.0 & 1.0 & 1.0 & 1.0 \\
1.0 & 1.0 & 2.0 & 1.0 & 1.0 & -1.0 & -1.0 & 4.0 \\
-1.0 & 1.0 & 2.0 & 2.0 & 0.0 & 1.0 & 1.0 & 6.0 \\
& & & & & & & \\
-2.0 & 1.0 & 0.0 & 2.0 & 1.0 & 1.0 & 1.0 & 4.0 \\
1.0 & 0.0 & 0.0 & -1.0 & -2.0 & -1.0 & -1.0 & -4.0 \\
0.0 & 1.0 & 1.0 & -3.0 & -2.0 & -1.0 & 1.0 & -3.0 \\
1.0 & 2.0 & 0.0 & 2.0 & 1.0 & 2.0 & 1.0 & 9.0 \\
2.0 & 1.0 & -3.0 & 1.0 & 2.0 & 1.0 & 2.0 & 6.0 \\
& & & & & & & \\
-1.0 & 1.0 & 2.0 & 2.0 & 1.0 & -3.0 & 4.0 & 6.0 \\
-1.0 & 0.0 & 1.0 & 1.0 & 4.0 & -2.0 & 1.0 & 4.0 \\
1.0 & -1.0 & 0.0 & -1.0 & -1.0 & -1.0 & -1.0 & -4.0 \\
-2.0 & 1.0 & 0.0 & 1.0 & 1.0 & 2.0 & 1.0 & 4.0 \\
1.0 & 2.0 & 1.0 & 1.0 & 0.0 & 0.0 & 1.0 & 6.0
\end{tabular}

The result triangular matrix with the right hand-side can be written in a compact form as follows:
\[
\left[\begin{array}{lll}
R_{1} & S_{1} & c_{1}  \tag{7.18}\\
R_{2} & S_{2} & c_{2} \\
R_{3} & S_{3} & c_{3} \\
0 & S_{4} & c_{4}
\end{array}\right]
\]

The above matrix takes the following values
\begin{tabular}{lllllccc}
1.732 & 1.155 & 0.577 & 0.0 & 0.0 & -1.732 & -0.577 & 1.155 \\
0.0 & 2.38 & 2.24 & 1.68 & 0.0 & 0.42 & 0.7 & 7.421 \\
0.0 & 0.0 & 2.156 & 1.037 & 0.00 & 0.027 & 0.355 & 3.575
\end{tabular}
\begin{tabular}{llrrcrcr}
3.162 & 0.632 & -1.897 & -0.316 & 0.316 & 0.316 & 0.632 & 2.846 \\
0.0 & 2.569 & -0.311 & 1.635 & 1.09 & 1.868 & 2.18 & 9.031 \\
0.0 & 0.0 & 2.511 & -2.426 & -2.812 & -1.123 & -1.243 & -5.094
\end{tabular}
\begin{tabular}{lcrrrrrr}
2.828 & -0.707 & -0.707 & -1.768 & -2.828 & 0.00 & -2.475 & -5.657 \\
0.0 & 2.55 & 1.373 & 1.863 & 0.392 & 0.00 & 2.451 & 8.629 \\
0.0 & 0.0 & 1.901 & 1.153 & 1.821 & -4.207 & 2.569 & 3.236
\end{tabular}
\begin{tabular}{llllllll}
0.0 & 0.0 & 0.0 & 3.81 & 1.487 & 0.64 & -0.48 & 5.457 \\
0.0 & 0.0 & 0.0 & 0.0 & 3.758 & 0.642 & -1.596 & 2.805 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 2.109 & 0.08 & 2.189 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.959 & 1.959
\end{tabular}

It's easy to verify the correctness of the result because we choose a problem where \(x=[1,1, \ldots, 1]^{T}\) and the residual is 0 . Thus the sum of the first 7 entries of the each row is equal to the value of the last entries in the same row.

\subsection*{7.6 Summary}

In this chapter, we have proposed a systolic architecture to compute the QR decomposition of the block structured sparse system ( \(\mathrm{BS}^{3}\) ) which finds a wide application in a variety of areas such as photogrammetry, Doppler radar positioning and geodetic survey problems. Because the size of the
problem in practical cases is usually very large, its parallel computation is very important. The proposed systolic array architecture can accelerate the problem solving whilst maintaining a high parallel efficiency. The square-root-free fast Givens transformation [Gentleman73] can further improve the efficiency of the systolic architecture by avoiding the possible bottleneck due to the square root operation in the circle PEs. Transputer networks, in which each PE accommodates a number of bytes of memory, may be well suitable for implementing such systems. The systolic algorithm can be also mapped into a general purpose parallel computer efficiently. The systolic array is very efficient because it enjoys a pipelining rate of one.

\section*{Chapter 8}

\section*{SYSTOLIC ARRAY FOR MATRIX INVERSION BY NEWTON ITERATION}

\subsection*{8.1 Introduction}

MATRIX operations are very prevalent in many applications such as signal processing and scientific computing applications. Matrix inversion is one of the most commonly used matrix operations and it plays an important role in linear algebra. The computational load for finding the inversion of a matrix with a large dimension is enormous. To speed up the computation, parallel processing is a logical choice. Among many parallel architectures, systolic array architecture features the important properties of modularity, regularity, local interconnection, a high degree of pipelining and a high efficiency. These properties make systolic array architecture suitable for VLSI/VHSIC (Very Large Scale Integration/Very High Speed Integrated Circuit) implementation. Systolic arrays have been proved to be an efficient solution for many matrix operations.

Matrix inversion can be computed either by the direct method or by the iterative method. The direct method computes the inversion by elimination method (eg. Gauss elimination) or matrix decomposition (eg. LU decomposition). The iterative method finds the approximate matrix inversion iteratively. Various systolic arrays have been designed to implement the matrix inversion using both methods.

Based on the Faddeev algorithm, a trapezoidal systolic array, which consists of a triangular array and a square array, can be used to compute the matrix inversion [Megson92] [Wan93b]. Another systolic design based on a triangularization algorithm has an orthogonal architecture which is a cascade of two triangular array [Robert85] [Comon87] (ref. Chap. 5). Systolic architecture can be also used to compute the matrix inversion iteretively (e.g. Evans and Margaritis designed an iterative systolic array using matrix power method [Evans91c]).

In this chapter, a novel systolic array architecture for matrix inversion by the Newton's iterative method is designed. The systolic array is obtained by integrating two systolic matrix multiplication
arrays and a systolic transposition array into a new square array to perform the matrix inversion iteratively. First, a systolic array for the fixed number of iteration is derived. Then a systolic array able to test convergence is presented. The proposed systolic array architecture is suitable for the applications where the size of the matrix to be inverted is large and the iterative method is favoured. Considerable speedup can be achieved while high efficiency is maintained.

The rest of this chapter is organized as follows. Sec. 8.2 describes the Newton method and gives the sequential algorithm. Sec. 8.3 introduces several systolic arrays which are used as building blocks of the systolic array for matrix inversion by the Newton iteration. Sec. 8.4 designs the overall systolic architecture where the number of the iterations is assumed to be fixed. Sec. 8.5 contains a modified systolic array which is able to test the convergence of the algorithm and to estimate the initial approximation. Finally, Sec. 8.6 concludes the chapter.

\subsection*{8.2 Newton Method for Matrix Inversion}

Pan and Reif [Pan85] have proposed a quadratically convergent method by Newton iteration for the calculation of the inverse of a non-singular square matrix. The algorithm is numerically stable within a relative precision. Further it provides an efficient method for computing the initial approximation of the matrix inverse for the iterative process to commence.

\subsection*{8.2.1 The Algorithm for Iterative Malrix Inversion}

For a well conditioned \(N \times N\) real matrix \(A\), the \(N\) ewton iterative matrix inversion algorithm can be outlined as follows.

\section*{Step 1. Initial Approximation}

Let \(X_{0}\) be the initial approximation to \(\boldsymbol{A}^{-1}\) defined by
\[
\begin{equation*}
X_{0}=\frac{1}{m} A^{T} \tag{8.1}
\end{equation*}
\]
where \(m\) is given by
\[
\begin{equation*}
m=\left\|A^{T} A\right\|_{1} \tag{8.2}
\end{equation*}
\]

The matrix \(p\)-norm are defined by
\[
\begin{equation*}
\|A\|_{p}=\sup _{x \rightarrow 0} \frac{\|A x\|_{p}}{\|x\|_{p}}, \quad\|x\|_{p}=\left(\sum_{i=1}^{N}\left|x_{i}\right|^{p}\right)^{1 / p} \quad \text { for any vector } x \tag{8.3}
\end{equation*}
\]

The matrix 1-norm and \(\infty\)-norm are specially given by
\[
\begin{align*}
& \|\boldsymbol{A}\|_{1}=\max _{1 \leq j \leq N} \sum_{i=1}^{N}\left|a_{i j}\right|  \tag{8.4}\\
& \|\boldsymbol{A}\|_{\infty}=\max _{1 \leq i \leq N} \sum_{j=1}^{N}\left|a_{i j}\right| \tag{8.5}
\end{align*}
\]

For this choice of initial approximation, it is proven that
\[
\begin{equation*}
\left\|R_{0}\right\|_{2}=1-\frac{1}{N K^{2}}<1 \tag{8.6}
\end{equation*}
\]
where \(\boldsymbol{R}_{0}\) is the initial residual matrix
\[
\begin{equation*}
R_{0}=I-A X_{0} \tag{8.7}
\end{equation*}
\]
and \(K=\|A\|_{2}\left\|A^{-1}\right\|_{2}\) is the condition number of \(A\).

\section*{Step 2. Newton Iteration}

Let \(\boldsymbol{X}_{k}\) denote an \(N \times N\) matrix to be the \(k\) th approximation to \(\boldsymbol{A}^{-1}\). Then the computation of the matrix inversion has the form of Newton's iteration:
\[
\begin{equation*}
X_{k+1}=X_{k}\left(2 I-A X_{k}\right) \tag{8.8}
\end{equation*}
\]
where \(I\) is an \(N \times N\) unit matrix. Eq. (8.8) shows the principle of the Newton's method for matrix inversion [Pan85] [Leighton92].

In order to examine why this rule works, it is useful to examine the residual matrix
\[
\begin{equation*}
R_{k}=I-A X_{k} \tag{8.9}
\end{equation*}
\]
which measures how far \(\boldsymbol{X}_{\boldsymbol{k}}\) is from \(\boldsymbol{A}^{-1}\). A simple calculation reveals that
\[
\begin{equation*}
R_{k+1}=I-A X_{k+1}=I-A\left(2 X_{k}-X_{k} A X_{k}\right)=\left(I-A X_{k}\right)^{2}=R_{k}^{2} \tag{8.10}
\end{equation*}
\]

This means that
\[
\begin{equation*}
R_{k}=R_{0}^{2^{k^{k}}} \tag{8.11}
\end{equation*}
\]

Thus, \(\boldsymbol{R}_{\boldsymbol{k}}\) converges quadratically to zero provided that the initial approximation is chosen according to Eq. (8.1).

To sum up, the Newton iterative method for matrix inverse computation can be given in algorithmic form as follows:

Algorithm 1: Matrix inversion by Newton iteration
Compute an initial approximation by: \(\boldsymbol{X}_{0}=\frac{1}{m} A^{T}\)
\(k=0 ; \boldsymbol{R}_{0}=\boldsymbol{I}-\boldsymbol{A} \boldsymbol{X}_{0}\);
while \(\left\|\boldsymbol{R}_{k}\right\|>\epsilon\) (where \(\epsilon\) is a given small positive number for convergence test)
\(k=k+1\);
\(\boldsymbol{R}_{k}=\boldsymbol{I}-\boldsymbol{A} \boldsymbol{X}_{k} ;\)
\(X_{k+1}=\boldsymbol{X}_{k}\left(I+R_{k}\right)\)

\subsection*{8.2.2 Initial Approximations for Special Matrices}

Eq. (8.1) provides a choice for initial approximation for the inversion of any arbitrary non-singular matrix. For the special matrices, it can be modified in order to provide better initial approximation and therefore reduce the iteration steps. Below we consider several special cases where alternative initial approximation exists.

Case 1. For symmetric positive definite matrix \(A\), which occurs frequently in many practical computations [Hageman81] [Varga62] [Golub89], the initial approximation can be chosen as
\[
\begin{equation*}
X_{0}=\frac{1}{m} I, \quad m=\|A\|_{1} \tag{8.12}
\end{equation*}
\]

The initial residual matrix has a norm of
\[
\begin{equation*}
\left\|R_{0}\right\|_{2} \leq 1-\frac{1}{N^{1 / 2} \operatorname{cond}(A)} \tag{8.13}
\end{equation*}
\]

Case 2. As it is shown in [Atkinson78] and [Dahlquist74], the inversion of an arbitrary non-singular matrix \(\boldsymbol{A}\) can be replaced by the inversion of \(\boldsymbol{A}^{\tau} \boldsymbol{A}\), which is always symmetric positive definite. Therefore, a general choice alternative to Eq . (8.1) can be given by
\[
\begin{equation*}
X_{0}=\frac{1}{m} I, \quad m=\left\|A^{T} A\right\|_{1} \tag{8.14}
\end{equation*}
\]
with
\[
\begin{equation*}
\left\|R_{0}\right\|_{2} \leq 1-\frac{2}{1+\operatorname{cond}(A)} \tag{8.15}
\end{equation*}
\]

Case 3. For a diagonally dominant matrix, which is again a frequent situation [Varga62] [Young71] [Golub89], the choice can be
\[
\begin{equation*}
X_{0}=\operatorname{diag}\left\{a_{11}^{-1}, a_{22}^{-1}, \cdots, a_{N N}^{-1}\right\} \tag{8.16}
\end{equation*}
\]
with
\[
\begin{equation*}
\left\|R_{0}\right\|_{2} \leq 1-\frac{1}{N c} \tag{8.17}
\end{equation*}
\]
where \(c\) is a suitable positive constant.

It is noticed that all the initial approximations discussed above require the matrix 1 -norm computation or either matrix \(\boldsymbol{A}\) itself or \(\boldsymbol{A}^{\boldsymbol{r}} \boldsymbol{A}\) except for Case 3 which has a simpler initial approximation.

\subsection*{8.3 Systolic Array Building Blocks for the Newton Method}

In this section, we consider the building blocks for systolic implementation of the Newton iterative method for matrix inverse. We apply some modifications to Algorithm 1. For simplicity, we assume that the algorithm converges to the required precision after a given number, say \(s\), of iterations. This assumption eliminates the converging test in Algorithm 1. We also assume that the initial approximation is given. With these modifications, Algorithm I becomes the form as follows:
```

Algorithm 2: Matrix inversion by Newton iteration (Simplified version)
Given $\boldsymbol{X}_{0}$ as an initial approximation to $\boldsymbol{A}^{-1}$;
for $k=0: s-1$
$\boldsymbol{R}_{k}=\boldsymbol{I}-A X_{k}$
$X_{k+1}=X_{k}+X_{k} R_{k}$

```

We will design a systolic array architecture to implement this algorithm while Sec. 8.5 will deal with the systolic array which is able to perform the convergence test and the initial approximation. From observation on Algorithm 2, we find that each iteration in the algorithm includes two matrix multiplication-and-accumulation operations. In this section, two systolic arrays for matrix multiplication-and-accumulation are introduced. A matrix transposition array is also treated as it is necessary in arranging data into the required format for executing Algorithm 2 systolically.

\subsection*{8.3.1 Systolic Arrays for Matrix Multiplication}

In Chap. 3, we have a detailed treatment of the systolic array designs for matrix multiplication. Firstly, the matrix multiplication is represented by a regular iterative algorithm (RIA). Then we select the processor space and iteration space. A feasible schedule can be selected. The RIA is then mapped into a systolic architecture based on the iteration space and the schedule selected.

Below, we briefly introduce two systolic arrays for matrix multiplication which will use in the systolic array for the iterative matrix inversion algorithm.

The first systolic array for matrix multiplication is shown in Figure 8.1. In this design, the matrix \(A\) flows through the systolic array from the left to the right while the matrix \(B\) flows through the array from the top to the bottom. The initial matrix \(C\) is pre-stored in the registers of the individual processing elements (PEs) and then updated during the systolic computation. The PE function is also shown in the same figure. In every time cycle, each PE in the array receives one input from the left and another from the top, then performs multiplication on


Figure 8.1 Systolic array I for matrix multiplication the two input operands and accumulation on the product and the previous partial result. The array can compute \(C=C+A B\) while \(C\) is stationary, and \(\boldsymbol{A}\) and \(\boldsymbol{B}\) are non-stationary [Megson92].

Another systolic design is illustrated in Figure 8.2. In this design, \(B\) is stationary while \(A\) and \(C\) are non-stationary. The matrix \(B\) is pre-loaded in the PEs of the systolic array. The matrix \(A\) in transposition flows through the systolic array from the left to the right. The initial matrix \(C\) enters the array from the top, then flows through the array downwards, and finally the resultant matrix comes out of the systolic array at the bottom. Each PE in the array receives two inputs from the left and the top, then performs the operation of multiplication and accumulation, and finally the result is sent to its downward neighbour.

\subsection*{8.3.2 Systolic Array for Matrix Transposition}

Figure 8.3 depicts a triangular systolic array (the solid line part) for matrix transposition. The dot line part completes the array into a square array. There are two kinds of PEs in the transposition array. The PE functions are also shown in the same figure. The circle PEs change the data transmitting from vertical direction to horizontal direction in order to transpose the input matrix. The square PEs perform only simple data transmitting tasks without changing data flow directions. The input matrix enters the array from the top of the array while the transposed matrix flows out of the array at


Figure 8.3 Systolic array for matrix transposition the right-hand side. The dotted PEs are unused for matrix transpose.

Below we will employ these systolic arrays as building blocks to construct a new systolic array for the Newton method for matrix inversion. We will show how to integrate these systolic arrays into a new systolic array to perform the tasks required by the Newton method.

\subsection*{8.4 Systolic Array for Iterative Matrix Inverse}

\subsection*{8.4.1 Overall Architecture}

The simplified version of the matrix inversion by the Newton iteration is given in Algorithm 2. The first operation in Algorithm 2 take the form of
\[
\begin{equation*}
R_{k}=I-A X_{k} \tag{8.18}
\end{equation*}
\]

To perform this operation, we can adopt the systolic array I for matrix multiplication as shown in Figure 8.1. At the start of the computation, the unit matrix, \(I\), is preloaded in the systolic array, and the matrix \(-\boldsymbol{A}\) and the matrix \(\boldsymbol{X}_{\boldsymbol{k}}\) enter the systolic array from the left and the top respectively. After the completion of the computation, the result matrix is store in


Figure 8.4 Utilization of systolic array I the PEs of the systolic array. The input and output relationship is shown in Figure 8.4.

The second matrix multiplication in Algorithm 2 has the form of
\[
\begin{equation*}
X_{k+1}=X_{k}\left(I+R_{k}\right)=X_{k}+X_{k} R_{k} \tag{8.19}
\end{equation*}
\]

We can employ the systolic array II for matrix multiplication in Figure 8.2 to perform this task. The input and the output of the systolic array are shown in Figure 8.5. At the start of the processing, \(\boldsymbol{R}_{k}\) stays in the systolic array while \(\boldsymbol{X}_{k}^{T}\) enters the array from the left and \(\boldsymbol{X}_{k}\) from the top. After the completion of the computation, the resultant matrix \(\boldsymbol{X}_{k+1}\) flows out of the


Figure 8.5 Utilization of systolic array II array at the bottom.

Figures 8.4 and 8.5 show how the two systolic arrays for matrix multiplication execute the tasks required by the Newton's method for matrix inversion. It is noticed that, in Figure 8.5, the transposition of a matrix is required.

Having discussed all the building blocks required by the Newton's method, we now consider the overall architecture to implement Algorithm 2.

From Figures 8.4 and 8.5 , we observe that the resultant matrix \(R_{k}\) of the systolic array I resides in the array while this matrix is required to be preloaded in the systolic array II. This inspires us to use a single systolic array to perform the two different tasks which are shown in Figures 8.4 and 8.5.

Figure 8.6 shows the systolic design for the Newton's iterative method for matrix inversion. There are 3 types of processing elements (PEs) in the proposed systolic array.


Figure 8.6 Overall architecture for Newton's method The shadowed square processing elements perform data interface tasks by sending the input data to the systolic array and feeding the output data back to the array. They act just like programmable multiplexers which switch the output data channels to appropriate input data channel. The rest of the processing elements perform useful computation required in Algorithm 2.

At the beginning of the computation, the input matrices \(-A\) and \(X_{0}\) enter into the systolic array through the data interface (shadowed square processing elements) from the left and the top of the array respectively. The systolic array computation can be classified as two phases for each iteration.

During the first phase of the computation, the array computes \(\boldsymbol{R}_{0}=\boldsymbol{I}-\boldsymbol{A} \boldsymbol{X}_{0}\) where the unit matrix \(I\) is assumed to be initially stored in the PEs of the systolic array. In this phase, the array performs the tasks as shown in Figure 8.4 together with the matrix transposition. The resultant matrix \(\boldsymbol{R}_{0}\) is stored in the individual PEs. The matrix - \(\boldsymbol{A}\) flows horizontally and feeds back to the array after passing through the array. The matrix \(X_{0}^{T}\) can be obtained by transposing the input matrix \(X_{0}\). The matrix \(X_{0}^{T}\) moves in the horizontal direction while the matrix \(\boldsymbol{X}_{0}\) moves in the vertical direction. Both of \(\boldsymbol{X}_{0}^{T}\) and \(\boldsymbol{X}_{0}\) are then feed back to the systolic array after passing through the array. During the second phase, the array performs the computation as shown in Figure 8.5. The matrix \(\boldsymbol{R}_{0}\) stays in the array while the matrices \(\boldsymbol{X}_{0}^{T}\) and \(\boldsymbol{X}_{0}\) flows through the array in horizontal direction and in vertical direction respectively. The result matrix for the first iteration, \(\boldsymbol{X}_{1}\), flows out of the array in vertical direction and then it is fed back to the array for the next iteration.

The processing of the two phases described above will repeat until the required number of iterations is completed. The final result can be collected at the top shadowed square processing elements or at the bottom of the systolic array. The systolic array has a similar architecture with a torus network.

\subsection*{8.4.2 Definitions of the Processing Elements}

Based on the above discussion, we can define the functions of the different types of the processing elements. The shadowed processing elements behave like multiplexers which select the external input data during the first phase of the first iteration and the feedback data later on. We can also add another channel to each of the shadowed PEs in order to send the result to the host system. Because these shadowed PEs are very simple, we will not give further details.


Figure 8.7 Definitions of PEs

The other PEs (unshadowed) in Figure 8.6 perform the useful computation required by the Newton's method. There is a slight difference between a diagonal PE, which is illustrated by a circle inside a square, and an off-diagonal PE, which is illustrated by a square. Figure 8.7 shows the definitions of processing elements. In Figure 8.7, the terms "Ist_phase" and "2nd_phase" indicate the control signal which specifies what operations PEs are required to perform at the given time instances. The term "initial" is another control signal which indicate the commencement of a new iteration. The "initial' signal may not be necessarily generated because it becomes true only when control signal changes from the second phase to the first phase. Therefore, it is possible to use a one-bit control signal to implement the phase control of the systolic array.

\subsection*{8.4.3 An Example}

To validate the proposed systolic design, it may be straightforward by showing the snapshots of the systolic array for a small size problem. We consider the inversion of the following matrix
\[
A=\left[\begin{array}{rrr}
3 & 2 & 1  \tag{8.20}\\
-1 & 2 & 3 \\
-2 & 1 & 1
\end{array}\right]
\]
and its inversion is as follows
\[
A^{-1}=\left[\begin{array}{rrr}
0.1 & 0.1 & -0.4  \tag{8.21}\\
0.5 & -0.5 & 1.0 \\
-0.3 & 0.7 & -0.8
\end{array}\right] \text {. }
\]

The initial approximation of the inversion is given by
\[
X_{0}=\left[\begin{array}{rrr}
0.0977 & 0.1016 & -0.4037  \tag{8.22}\\
0.5065 & -0.4906 & 1.0184 \\
-0.3040 & 0.6830 & -0.8051
\end{array}\right]
\]

Figures 8.8-8. 18 show the systolic computing process of one Newton iteration on the matrices given above. The result after one iteration is:
\[
X_{1}=X_{0}\left(2 I-A X_{0}\right)=\left[\begin{array}{rrr}
0.1000 & 0.1000 & -0.3999  \tag{8.23}\\
0.4998 & -0.4995 & .9992 \\
-0.2999 & 0.6994 & -0.7994
\end{array}\right] \text {. }
\]

Figure 8.8 shows the initial snapshot of the systolic architecture with the input data from the left and the top. After one step of computation, as shown in Figure 8.9, all the PEs in the array update their values and transfer the data to the appropriate channels according to the PE definitions shown in Figure 8.7. Notice that we list only the useful computation of the top left PE while ignoring the update of those PEs which wait for the useful data coming. The next update of Figure 8.10 is based on the input data as given in Figure 8.9. This process repeats until the required computation completes. We also use an anti-diagonal line to distinguish the computations of the two phases. We describe only one iteration in Figures 8.8-8.18. However, it is straightforward to perform more iterations by the array and the data flow is continuous between adjacent iterations.


Figure 8.8 Snapshot for \(\mathrm{t}=0\)


Figure 8.10 Snapshot for \(\mathrm{t}=2\)


Figure 8.12 Snapshot for \(\mathrm{t}=4\)


Figure 8.9 Snapshot for \(\mathrm{t}=1\)


Figure 8.11 Snapshot for \(t=3\)


Figure 8.13 Snapshot for \(\mathrm{t}=5\)


Figure 8.14 Snapshot for \(t=6\)


Figure 8.16 Snapshot for \(t=8\)


Figure 8.15 Snapshot for \(\mathrm{t}=7\)


Figure 8.17 Snapshot for \(\mathrm{t}=9\)


Figure 8.18 Snapshot for \(t=10\)

\subsection*{8.5 Modifications to the Basic Systolic Architecture}

\subsection*{8.5.1 Residual Matrix Evaluation}

The proposed systolic array can compute the matrix inversion iteratively. The above discussion is based on Algorithm 2 which assumes that the number of iteration is given. In practice, we need to know when the iteration should stop as described in Algorithm 1. This can be done by modifying the systolic architecture in Figure 8.6. We can add another vertical data channel for each PE and a row of extra PEs (rhombic PEs) at the bottom as shown in Figure 8.19 in order to accumulate the residual matrix (another alternative is to modify the shadowed PEs at top side of array to compute the residual). The new PE


Figure 8.19 Systolic array with convergence test definitions are described in Figure 8.20.

Notice that the square PEs and the diagonal PEs in Figure 8.20 differ from their corresponding PEs in Figure 8.7 only in the second phase. The square PEs and the diagonal PEs together with the rhombic PEs for the systolic array with convergence test accumulate the norm of the residual matrix during the second


Figure 8.20 PE definitions for systolic array with convergence test
phase. The output of the right most rhombic PE in Figure 8.19 gives the square of the Frobenius norm of the residual matrix defined by,
\[
\begin{equation*}
e=\left\|\boldsymbol{R}_{k}\right\|_{F}^{2}=\sum_{i=1}^{N} \sum_{j=1}^{N} r_{i j}^{2} \tag{8.24}
\end{equation*}
\]

This quantity can be used to generate a signal to control the systolic array to complete the computation when the solution with the required precision is obtained.

Of course, the estimation of the norm of the residual matrix increases the complexity of the processing elements as shown in Figures 8.19 and 8.20. During the second phase, the square PEs and the diagonal PEs are required to perform another multiplication. However, the estimation of the norm of the residual matrix may reduce the iteration number comparing the fixed iteration number scheme.

A simpler and therefore better way is to estimate the 1 -norm of the residual matrix, as given in Eq. (8.4). Instead of calculating the sum of the squares of the column vectors of the residual matrix, which involves multiplications, we calculate the sum of the absolute values in a column wise way. The 1norm computation involves only the additions and comparisons. This provides another alternative to perform the convergence test in less complexity.

It is possible to further simplify the convergence test if we choose a convergence criterion which compares the successive iterations based on calculations of 1 -norm or \(\infty\)-norm. Since the result of the iteration result \(X_{k}\) is available at the shadowed PEs at the top of the array in Figure 8.19, the shadowed PEs can be used to calculate the l-norm of \(\boldsymbol{X}_{\boldsymbol{k}}\). This eliminates the need of the extra vertical channels and the unshadowed PEs have same complexity as in the basic systolic architecture. Similarly, the left side shadowed PEs can be used to compute the \(\infty\)-norm and to test the convergence of the algorithm.

\subsection*{8.5.2 Initial Approximation Computation}

We first consider the initial approximation given by Eq. (8.1). The approximation involves operations of a matrix multiplication and a matrix 1 -norm computation followed by a matrix and scalar multiplication. The computation of the initial approximation is illustrated in Figure 8.21.

The systolic array first computes the product of matrix \(-A\) and the unit matrix \(I\) and store the result \(-A\) in the systolic array (as shown in Figure 8.4). Then the matrix - \(\boldsymbol{A}\) circulates back to the left side of the array and the 0 matrix arrives at the top of the array. The systolic array is then ready to compute \(\boldsymbol{A}^{T} \boldsymbol{A}\) (as shown in Figure 8.5). The product \(\boldsymbol{A}^{T} \boldsymbol{A}\) is piped out of the array at the bottom. This product can be sent back to the top shadowed PEs in Figure 8.19. The top shadowed PE can be modified to calculate the matrix 1-norm with connecting these PEs horizontally. Then another 0 matrix is fed into the systolic array before the matrix 1 -norm is available. Then the matrix \(\boldsymbol{A}^{T}\) passes through the top shadowed PEs and forms the initial approximation given by Eq. (8.1). the systolic array can perform Newton iteration as discussed before.


Figure 8.21 Initial approximation

The initial approximation of the matrix inverse requires \(3 N\) cycles. This is generally a small overhead for the systolic array. The incorporation of the initial approximation will slightly complcate the control of the systolic array.

The initial approximation given by Eq. (8.14) is similar to the one given by Eq. (8.1). We need only substitute the matrix \(I\) instead of the matrix \(A^{T}\) in the vertical input. This may result a simpler systolic array because there is no input is required for the top side of array. The shadowed PEs can generate all the input data, \(I\) and 0 , by themselves. The initial approximation given by Eq. (8.12) for a symmetric positive definite matrix can also be easily implemented by the existing systolic array.

\subsection*{8.6 Summary}

A novel systolic array architecture has been developed for the matrix inversion by the Newton's iteration method. The Newton's method has a very attractive feature that some simpler initial approximations can always guarantee a convergence comparing with other iteration method. The systolic architecture is obtained by integrating two different matrix-matrix multiplication arrays and
a matrix transposition array into a new square array. A systolic array able to test the convergence of the algorithm is also proposed. The initial approximations can also be incorporated into the systolic array. This systolic implementation of the iterative matrix inversion is highly parallel and the data flow is regular. The partition of the systolic array for large size problems is very straightforward. The speedup of the systolic array is significant when the problem size is large. The proposed systolic architecture is suitable for the applications where the size of the matrix to be inverted is large and the iterative method is favoured.

\section*{Chapter 9}

\section*{A SYSTOLIC ARCHITECTURE FOR CAPON'S DOA ESTIMATION METHOD}

\subsection*{9.1 Introduction}

THE array signal processing field deals with the processing of information-bearing signals collected by an array of sensors operating in an environment of interest. The aim of array processing is to estimate the object parameters. In sonar or radar array processing, the directions-ofarrival (DOA) of targets are the most important parameters for detecting and locating the targets.

Many approaches for DOA estimation have been developed. In recent years, the so-called high resolution algorithms have received more and more attention. These algorithms offer better performance in resolving two closely spaced emitters (source of radiations) than the conventional beamforming (CBF), or Fourier method.

The performance improvement of the high resolution algorithms over the CBF is usually achieved at the cost of more intensive computational loads for the algorithms. The huge computational loads limit the applications of the high resolution algorithms in a real time environment.

The parallel processing techniques employing multiple processors can speed up the algorithms so that real time processing becomes possible. The systolic array parallel architectures with simple processing elements (PEs) communicating locally are suitable for VLSI/VHSIC implementation.

Capon's DOA estimation method [Capon69], also known as the minimum variance distortionless response (MVDR) method, can provide a better resolution of spatial spectrum than the conventional beamforming method. Although other methods, such as the multiple signal classification (MUSIC) method, can give a better performance than Capon's method, they require more operations (eg. eigen system or SVD evaluation is involved in the MUSIC method) than Capon's method.

In this chapter [Wan93a], we present a systolic array architecture for Capon's DOA estimation method. Capon's method for DOA estimation is described in Sec. 9.2. In Sec. 9.3, we develop the algorithm for Capon's method based on the QR decomposition technique, and then investigate the computation tasks and the basic operations required by this algorithm. In Sec. 9.4, we propose a systolic architecture to implement Capon's method. Sec. 9.5 deals with the simulation of the architecture via occam 2 with simulation results while Sec. 9.6 contains a summary.

\subsection*{9.2 Capon's Method for DOA Estimation}

\subsection*{9.2.1 The array and signals}

Consider a uniformly spaced linear array consisting of \(N\) identical sensors and receiving signals from \(M\) narrowband signals \(u_{1}(t), u_{2}(t), \cdots, u_{M}(t)\) that arrive at the array from directions \(\theta_{1}, \theta_{2}, \cdots, \theta_{M}\) as shown in Figure 9.1. The extension of the discussion here to a sensor array of arbitrary geometry is


Figure 9.1. The sensor array and signals
straightforward. The received signal at \(i\) th sensor can be written as
\[
2 \quad\left(\begin{array}{l}
x)=A u(t)+n(t) \tag{9.1}
\end{array}\right.
\]
where
\(\boldsymbol{A} \in \mathbb{C}^{\mathcal{N} \mathcal{M}}=\left[\begin{array}{llll}\boldsymbol{s}\left(\theta_{1}\right) & \boldsymbol{s}\left(\theta_{2}\right) & \cdots & \boldsymbol{s}\left(\theta_{M}\right)\end{array}\right]\) is the array manifold;
\(N\) is the total number of sensors;
\(M\) is the number of incident signals from distinct directions;
\(\boldsymbol{s}\left(\theta_{i}\right)\) is a steering vector defined by
\[
s\left(\theta_{i}\right)=\left[1, e^{j \frac{i \alpha d}{2} \sin \theta_{1}}, \cdots, e^{j \frac{2 \alpha d}{2}(N-1) \sin \theta_{1}}\right]^{T}
\]
\(d\) is the interelement spacing of the linear array;
\(\lambda\) is the wavelength;
\(\theta_{i}\) is the \(i\) th angle of incidence of a planewave measured with respect to the normal to the array for \(i=1,2, \ldots, M\);
\(u(t) \in \mathbb{C}^{M \times 1}\) is the vector of amplitudes of the \(M\) incident signals at time \(t\);
\(n(t) \in \mathbb{C}^{N+1}\) is a Gaussian noise vector with zero mean value;
\(\boldsymbol{x}(t) \in \mathbb{C}^{N \times 1}\) is the vector of observations for the array output at time \(t\).

The specific parameters of interest are the directions-of-arrival, \(\left\{\theta_{i} \mid i=1,2, \ldots, M\right\}\), and the value of \(M\) itself. The basic idea of the method is to estimate a wavenumber spectrum (spatial spectrum) based on the received data. The peaks of this spectrum correspond to the DOA of the incident signals.

\subsection*{9.2.2 Capon's Method}

The essence of Capon's method, or MVDR algorithm, is to construct a spatial filter that processes the vector of observations to minimise the variance of the resulting output subject to the condition that there is a distortionless response at some spatial angle \(\theta\).

Assume that the spatial filter processes the input data vector \(\boldsymbol{x}\) by a filter with the coefficient vector \(w\) in the following way
\[
\begin{equation*}
y=w^{H} x \tag{9.2}
\end{equation*}
\]
where \(y\) is the resulting output of the spatial filter. The variance of the output can be expressed as
\[
\begin{equation*}
e=E\left\{|v|^{2}\right\}=w^{H} C_{x x} \boldsymbol{w} \tag{9.3}
\end{equation*}
\]
where
\[
\begin{equation*}
C_{x x}=E\left\{x x^{H}\right\} \tag{9.4}
\end{equation*}
\]
is the correlation matrix.

The constraint can be expressed as
\[
\begin{equation*}
w^{H} s(\theta)=1 \tag{9.5}
\end{equation*}
\]
where \(s(\theta)\) is the steering vector for the given direction \(\theta\).

Therefore, the problem becomes
\[
\begin{gather*}
\min _{w} w^{H} C_{x x} w  \tag{9.6}\\
\text { subject to } w^{H} s(\theta)=1
\end{gather*}
\]

To find the optimum weighting vector, we construct the following function
\[
\begin{equation*}
F(w, \mu)=w^{H} C_{x x} w-\mu\left[w^{H} s(\theta)-1\right] \tag{9.7}
\end{equation*}
\]

Using Lagrange's method, we let
\[
\begin{align*}
& \frac{\partial F}{\partial w}=2 C_{x x} \boldsymbol{w}-\mu s(\theta)=0  \tag{9.8}\\
& \frac{\partial F}{\partial \mu}=w^{H} \boldsymbol{s}(\theta)-1=0 \tag{9.9}
\end{align*}
\]

By solving Eqs. (9.8) and (9.9), we obtain the optimum weighting vector as
\[
\begin{equation*}
w_{o p t}=\frac{C_{x x}^{-1} s(\theta)}{s^{H}(\theta) C_{x x}^{-1} s(\theta)} \tag{9.10}
\end{equation*}
\]

Then Capon's spatial spectrum of the sensor output is defined by
\[
\begin{equation*}
P(\theta)=\min \{e\}=\frac{1}{s^{H}(\theta) C_{x x}^{-1} s(\theta)} \tag{9.11}
\end{equation*}
\]

Adaptive beamforming shares the same principle as discussed above. However, the output of the adaptive beamformer is a waveform as given in Eq. (9.2), which is often subsequently subjected to some form of spectral analysis in a sonar/radar application [Baggeroer78]. Some systolic array architectures for adaptive beamformer have been proposed in [McWhirter83] [Ward84]. In this chapter, we concern the computation as shown in Eq. (9.11), which provides the intensity of the environment signal as a function of direction [Wan93a].

In practice, we usually cannot obtain an ensemble averaged correlation matrix \(C_{x x}\). Rather, we have to use a time averaged estimate \(\hat{C}_{x x}\) that is obtained from a finite number of snapshots of measurement vectors collected at different instances of time. The time averaged estimate of the correlation matrix can be expressed as
\[
\begin{equation*}
\dot{C}_{x x}=\frac{1}{L} \sum_{k=1}^{L} x(k) x^{H}(k) \tag{9.12}
\end{equation*}
\]
where \(L\) is the number of the observed snapshots and \(x(k)\) (for \(k=1,2, \ldots, L)\) is the \(k\) th snapshot of sensor output \(x(f)\) at discrete time \(k\).

Thus Capon's method can be summed up as following:
Collect the data \(\boldsymbol{x}(k)\) (for \(k=1,2, \ldots, L\) ) from the sensors;
Estimate the correlatiom matrix according to Eq. (9.12);
Compute the inverse of the correlation matrix \(\hat{C}_{x x}^{-1}\);
Compute the spectrum according to Eq. (9.11) for different scanning directions;
Find the peaks of the spectrum corresponding to the DOA of the incident signals.

\subsection*{9.3 Computational Tasks of Capon's Method}

In this section, we discuss Capon's method based on the \(Q R\) decomposition technique. The computational tasks for Capon's spectral estimation are also analysed.

\subsection*{9.3.1 Capon's Method Bused on QR Decomposition Technique}

In Sec. 9.2, Capon's method is developed. We notice that the algorithm requires to compute the correlation matrix and its inverse. The estimation of the correlation matrix and the computation of its inverse are generally time consuming if the number of the sensors is large (and this is typically true for practical cases). Accordingly, special precautions have to be taken in the computation of the spectrum. The spectral analysis method based on singular value decomposition (SVD) on the data matrix is examined in [Haykin92]. In the following discussion, we present a method based on QR decomposition of the data matrix to compute Capon's spatial spectrum.

We let \(\boldsymbol{X}\) denote the data matrix obtained from \(L\) snapshots and obtain
\[
X=\left[\begin{array}{c}
x^{T}(1)  \tag{9.13}\\
x^{T}(2) \\
\vdots \\
x^{T}(L)
\end{array}\right]=\left[\begin{array}{cccc}
x_{1}(1) & x_{2}(1) & \cdots & x_{N}(1) \\
x_{1}(2) & x_{2}(2) & \cdots & x_{N}(2) \\
\vdots & \vdots & \cdots & \vdots \\
x_{1}(L) & x_{2}(L) & \cdots & x_{N}(L)
\end{array}\right]
\]

Then the correlation matrix can be expressed as
\[
\begin{equation*}
\hat{C}_{x x}=\frac{1}{L} X^{T} X^{*} \tag{9.14}
\end{equation*}
\]

Assuming that \(L \geq N\), we can decompose the data matrix \(\boldsymbol{X}\) as the product of an \(L \times L\) orthonormal matrix \(Q\) and an \(L \times N\) matrix \(\bar{R}\) as follows
\[
X=Q \bar{R}=\varrho\left[\begin{array}{l}
R  \tag{9.15}\\
0
\end{array}\right]
\]
where
\[
\begin{equation*}
Q Q^{H}=I \quad \text { or } \quad Q^{T} Q^{\bullet}=I \tag{9.16}
\end{equation*}
\]
and \(R\) is an \(N \times N\) upper triangular matrix.

Then we can rewrite Eq. (9.14) as follows
\[
\begin{equation*}
\hat{C}_{x x}=\frac{1}{L}(Q \bar{R})^{T}(Q \bar{R})^{*}=\frac{1}{L} \bar{R}^{T} Q^{T} Q^{\cdot} \bar{R}^{*}=\frac{1}{L} \bar{R}^{T} \bar{R}^{*}=\frac{1}{L} R^{\tau} R^{\bullet} \tag{9.17}
\end{equation*}
\]

We substitute Eq. (9.17) into Eq. (9.11) and then obtain
\[
\begin{equation*}
\hat{P}(\theta)=\frac{L}{s^{H}(\theta)\left(R^{T} R^{*}\right)^{-1} s(\theta)}=\frac{L}{\left[R^{-T} s(\theta)\right]^{H}\left[R^{-T} s(\theta)\right]}=\frac{L}{\left\|R^{-T} s(\theta)\right\|^{2}} \tag{9.18}
\end{equation*}
\]

Then we can outline the steps for Capon's method based on QR decomposition:
Collect the data from the sensors and form the data matrix \(\boldsymbol{X}\);
Perform the QR decomposition on \(X\) and obtain the upper triangular matrix \(R\);

Compute \(R^{-\tau} s(\theta)\) which is actually a version of forward substitution;
Compute the Euclidean norm of \(\boldsymbol{R}^{-T} \boldsymbol{s}(\theta)\) and obtain the spectrum;
Find the spectral peaks corresponding to the DOAs of the incident signals.

Thus, the computation tasks for Capon's method are composed of the following basic operations:
QR decomposition;
Forward substitution;
Vector norm computation.

These basic operations involved in Capon's method are discussed in the following sub-sections with details of their computational complexities.

\subsection*{9.3.2 QR Decomposition}

Gentleman and Kung [Gentleman81] first designed the widely used triangular systolic array for matrix triangularization. In their design, the Givens QR decomposition method was used. The Givens QR method for the case of complex data can be described in the algorithmic form as
```

for $j=1: N$
$X(j, j: N)=X(j, j: N) \cdot X^{*}(j, j)| | X(j, j) \mid$
for $i=j+1: L$
$[c, s]=\operatorname{givens}(X(j, j), X(i, j))$
$X(j, j: N), X(i, j: N)=\operatorname{row} \cdot \operatorname{rot}(X(j, j: N), X(i, j: N), c, s)$
end
end

```
function: \([c, s]=\operatorname{givens}(a, b)\)
(where \(a\) is real)
if \(|a|=0\) and \(|b|=0\)
\(c=1, s=0\)
else
\(t=\operatorname{sqrt}\left(a^{2}+|b|^{2}\right)\)
\(c=a / t\)
\(\operatorname{REAL}(s)=\operatorname{REAL}(b) / t\) \(\operatorname{IMAG}(s)=\operatorname{IMAG}(h) / t\)
end
end givens
function: \(Y, Z=\operatorname{row} . \operatorname{rot}(Y, Z, c, s)\)
(where \(c\) is real)
\(q=\operatorname{cols}(Y)\)
for \(j=1: q\)
\(11=Y(1, j), \mathrm{t} 2=Z(1, j)\)
\(Y(1, j)=c \circ 11-5 \circ / 2\)
\(Y(2, j)=[\stackrel{\square}{\circ} / 1+c \circ 12\)
end
end row.rot

We notice that the function givens involves 6 multiplications/divisions and 1 square root operation while the function row.rot involves 12 multiplications/divisions. We use the term 1 flop to represent one inner product step (ips). Then we can estimate the flops required for the above algorithm
\[
\begin{equation*}
F_{Q R}=\sum_{j=1}^{N} \sum_{i=j+1}^{L}\left(7+\sum_{k=j}^{N} 12\right) \approx 6\left[L N^{2}-N^{3} / 3\right] \tag{9.19}
\end{equation*}
\]
where the square root operation is counted as I flop.

\subsection*{9.3.3 Forward Substitution to Compute \(\boldsymbol{R}^{-T} \boldsymbol{s}(\theta)\)}

The forward substitution to compute \(\boldsymbol{y}=\boldsymbol{R}^{-\boldsymbol{T}} \boldsymbol{s}(\theta)\) can be described in the algorithmic form as
```

$y=s$
for $j=1: N-1$
$y(j)=y(j) / R(j, j)$
$y(j+1: N)=y(j+1: N)-y(j) R(j, j+1: N)$
end
$y(N)=y(N) / R(N, N)$

```

Considering the complex operation in the algorithm, we obtain the required computational load for the forward substitution as
\[
\begin{equation*}
F_{F S} \approx 2 N^{2} \tag{9.20}
\end{equation*}
\]

\subsection*{9.3.4 Compute the Norm of a Vector.}

Another basic operation involved in Capon's method is the computation of the vector norm (actually we compute the square of the vector norm). This can be described as
```

$v(1: N)$ : the input vector
norm: the square of the Euclidean norm
norm=0
for $\mathrm{i}=1: \mathrm{N}$
norm=norm $+v^{*}(j) v(j)$
end

```

This requires a computational load as
\[
\begin{equation*}
F_{N M}=2 N \tag{9.21}
\end{equation*}
\]

\subsection*{9.3.5 The Total Compuataional Load for Capon's Method}

We assume that the sampling frequency for the array signal processing system is \(f \mathrm{~Hz}\), the number of the spectra to be computed in a second is \(n_{s}\) and the number of the directions scanned in the interest range for each spectrum is \(n_{d}\).

In each second, there are \(f\) data snapshots which are divided into \(n_{s}\) segments. This means that, in a second, the system (sonar or radar) requires to compute \(n_{d}\) spectra and that each spectrum is from a segment of data snapshots with length of \(L=f / n_{d}\). Thus the total computational tasks for Capon's method in a second can be expressed as
\[
\begin{align*}
F & =n_{s}\left[F_{Q R} I_{L-\frac{f}{n_{d}}}^{n_{d}} F_{F S}+n_{d} F_{N M}\right] \\
& =n_{s}\left[6\left(\frac{f}{n_{s}} N^{2}-\frac{N^{3}}{3}\right)+2 n_{d} N^{2}+2 n_{d} N\right]  \tag{9.22}\\
& =6\left(f N^{2}-\frac{n_{s} N^{3}}{3}\right)+2 n_{d} n_{s} N(N+1)
\end{align*}
\]

For the case of a sonar system, a sample frequency may be assumed to be \(10,800 \mathrm{~Hz}\). The number of the sensors may have the value of \(60 . n_{d}\) can be assumed as 180 for the case where the sonar scans from \(-90^{\circ}\) to \(90^{\circ}\) with \(I^{\prime \prime}\) step. \(n_{s}\) can be chosen as around 20. With these assumptions, we obtain the computational load in a second for Capon's method as: \(F=251.0 \mathrm{Mflops} / \mathrm{s}\). With more examples, Table 9.1 lists the computational loads for Capon's method.

Table 9.1 Computational load for Capon's method
\begin{tabular}{|c|r|r|r|r|r|}
\hline\(f(\mathrm{~Hz})\) & 1800 & 3600 & 5400 & 10800 & 18000 \\
\hline\(N\) & 50 & 50 & 80 & 60 & 100 \\
\hline\(n_{s}\) & 5 & 20 & 15 & 20 & 25 \\
\hline\(n_{d}\) & 360 & 180 & 180 & 180 & 360 \\
\hline\(F(\) flops \(/ \mathrm{sec})\) & 34.9 M & 67.4 M & 227.0 M & 251.0 M & 1211.8 M \\
\hline
\end{tabular}

Table 9.1 shows that the computational foad for Capon's method is intensive. The system requires a throughput up to over 1000 mega flops per second. For a radar system, where the sampling frequency
is much higher than the sampling frequency in a sonar system, i.e where data rate is much higher, the computational load for the Capon's method may increase tremendously. Therefore, the parallel processing of Capon's DOA estimation method is necessary in a real time system. In next section, we discuss the systolic architecture for the Capon's method.

\subsection*{9.4 Systolic Array for Capon's Method}

In this section, we design a systolic array for Capon's method. First, we give a brief introduction to the systolic arrays for the basic operations discussed in the last section. Then we combine these systolic arrays and construct a new systolic array to perform the Capon's DOA estimation.

\subsection*{9.4.1 Systolic Array for QR Decomposition}

The systolic array architecture for QR decomposition was first introduced by Gentleman and Kung [Gentleman81]. The triangular systolic array depicted in Figure 9.2 has been applied to a wide range of problems in signal processing such as adaptive filtering and adaptive beamforming. More details on the design of this systolic array (for real matrix) has been treated in Chap. 4 .


Figure 9.2 Systolic array for QR decomposition


Figure 9.3 PE definitions for QR decomposition

The PE definitions for the QR decomposition with complex input data are given in Figure 9.3.

\subsection*{9.4.2 Systolic Array for Forward Substitution}

The systolic array for the forward substitution has the same geometry as the systolic array for QR decomposition. Figure 9.4 describes the systolic array architecture. The input vector enters into the array from the top of the array and the output vector comes out of the array at the right hand side. The entries of the upper triangular matrix are assumed to be stored in the PEs.

If a sequence of vectors enter into the systolic array as a pipeline, the output vectors come out of the array as a pipeline.

The PE definitions for the forward substitution are also described in Figure 9.4.


Figure 9.4 Systolic array for forward substitution

\subsection*{9.4.3 Systolic Array for Vector Norm Computation}

The vector norm computation can be achieved by the linear systolic array as shown in Figure 9.5. It is obvious that the output of the systolic array in Figure 9.5 is the sum of the squares of individual input value: When many vectors enter the array from the left side, the linear systolic array can compute their norms in a pipeline way.


Figure 9.5 Systolic array for norm computation

\subsection*{9.4.4 Systolic Array for Capon's Method}

Having discussed the systolic arrays for the basic matrix computations involved in the Capon's method for DOA estimation, we can combine these systolic array to construct a new systolic array to implement Capon's method.

We allow the triangular systolic array to receive the sampling data as well as the steering vectors from the north. Then the triangular systolic array can perform both the QR decomposition and the forward substitution for spectral scanning. Figure 9.6 shows the systolic array architecture for the Capon's DOA estimation method.

The systolic array implies that both the sampling data and the output spectral data have the same data rate. For instance, if the sampling frequency is 5000 Hz , the obtained output spectra also have in a second 5000 points which may be divided into a number of segments, each of them covering the interest scanning directions.

The circle PEs and the square PEs in the array compute both the QR decomposition and the forward substitution while the rhombic PEs compute the vector norms. The forward substitution requires the result of the QR decomposition, i.e, the upper triangular matrix \(R\). The problem now is how the


Figure 9.6 Systolic array for Capon's DOA estimation forward substitution uses the resulting data of the QR decomposition? We can define the PEs in this way:
```

for i=1:L
the task for QR decomposition
end

```
```

do
for i=1:L
the task for QR decomposition
the task for forward substitution on the former R
end
update the upper triagular matrix R
loop

```

In this method, the spectra we compute are based on the already arrived segments of \(L\) snapshots. That is, we perform the first QR decomposition on the first segment of snapshots and then we compute the second QR decomposition while computing the first spectrum based on the first upper triangular matrix \(\boldsymbol{R}\). We then repeat the same procedure of performing QR decomposition on the current data segment and computing the spectrum based on the last upper triangular matrix

Noticing that the upper triangular array \(R\) is updated when a new snapshot of sampling data passes through the systolic array, we can then define the PEs in another way as follows:
```

do
for i=1:L
the task for QR decomposition
the task for forward substitution over the updated R
end
loop

```

The latter method means that we compute the following spectrum
\[
\begin{equation*}
\hat{P}(\theta)=\frac{1}{s^{H}(\theta) \dot{C}_{x x}^{-1}(t) s(\theta)} \tag{9.23}
\end{equation*}
\]
where
\[
\begin{equation*}
\grave{C}_{x x}(t)=\frac{1}{t} \sum_{k=1}^{t} x(k) x^{H}(k) \tag{9.24}
\end{equation*}
\]
and the \(t\) represents the discrete time. Instead of using the estimate covariance matrix on the already coming \(L\) snapshots, we use the updated correlation matrix to compute the spatial spectrum.

We give the detailed PE definition for the latter method in Figure 9.7 while it is convenient to define the PEs for the former method.


Figure 9.7 PE definitions of the systolic array for Capon's method

\subsection*{9.5 Occam Simulation}

In this section, we discuss the simulation of the systolic architecture proposed in the last section. We use the parallel programming language occam 2 in a single transputer system to simulate the parallel architecture for Capon's method.

\subsection*{9.5.1 Simulation Method}

The Occam 2 programming language was designed for the transputer, a microprocessor as a building block of a interconnected network. Many different network topologies may be created with a system of transputers by connecting the four pair communication links of each transputer in different ways. The parallel algorithms developed in a single transputer by the occam 2 language can be directly configured or mapped to transputer networks. Thus the programming for the parallel architectures in transputers usually begins with the parallel architecture simulation in a single transputer system. After the parallel algorithms have been tested in a single transputer system, the algorithms may be distributed to a transputer networks for speedup.

We simulate the Capon's DOA estimation method by occam 2 in a single transputer system. We outline the simulation algorithm in a pseudo-code form as following.
```

PAR
.. signal generating (narrowband signals with Gaussian noises)
PAR
PAR i=0 FOR pe.c.and.s (the number of PE in triangular array)
irow, icol:= the position of the ith processing element
IF
irow=icol (the PE is in the diagonal of the array)
perform the function of the circle element
TRUE
perform the function of the square element
PAR i=0 FOR N
perform the function of rhombic element
... obtain result

```

Based on the above structure, we have simulated the parallel architecture of the Capon's method for DOA estimation. The simulation result is given below.

\subsection*{9.5.2 Simulation Result}

Figure 9.8 presents the Capon's spectra from the above simulation algorithm. The number of sensors is 8 . Two uncorrelated signal sources with the additive white Gaussian noise arrive at the array from the directions of \(-2^{\circ}\) and \(8^{\circ}\). The signal-tonoise ratio, SNR , is 10 dB for each signal. We use 1810 data snapshots to compute 10 spatial spectra scanning from \(-90^{\circ}\) to \(90^{\prime \prime}\) with \(1^{\circ}\) step. The 10 spectra are plotted in the same plot in an area graph format. In an area graph, a different pattern fills the area between each plotted range that corresponds to the amplitude of the spectral


Figure 9.8. Simulation result function. This kind of graph can emphasise broad trends and continuity in data over time. We choose the format in order to emphasize the spectral peaks. In an actual sonar system, the bearing-time record, which uses colours or grey levels to represent the signal level versus time and bearing, is often used to increase the probability of detection and to track moving targets. This kind of bearing-time display will provide a better performance for detecting a potential target in a noisy background while it demands more spectra to be computed in the given time. This shows that the performance improvement causes a large computational load and that, on the other hand, the large computational load usually brings the performance improvement.

\subsection*{9.6 Summary}

In this chapter, we propose a systolic architecture for Capon's DOA estimation method. The algorithm is highly efficient and the speed up is significant. Several points are mentioned below for the improvement of the systolic algorithm.

Firstly, the square-root-free QR decomposition algorithm is preferred to reduce the possible bottleneck occurring in the circle elements due to the square root operation. It also slightly reduces the computational load of the square element [Gentleman73]. This algorithm may bring higher efficiency for the architecture.

Secondly, there are several choices when deciding how to use the sampling data in QR decomposition. Two methods have been proposed in Sec. 9.4.4. Another method is applying appropriate weights to the sampling data. For instance, we can apply a forgetting factor to the sampling data so that the earlier data affect the system less than the later data.

Thirdly, the data rate for the spectra to be computed can be different from the sampling data rate although we assumed they are same in Sec. 9.4. For example, we can change the PE definitions so that, after updating the matrix \(R\) twice with two snapshots, the spectral data is computed once. This means that the steering vectors enter the array at a lower speed than the sampling data. In this way, we can select an appropriate data rate for spectral output.

Finally, the architecture is very suitable for VLSI implementation as only a few kinds of PEs are used in the array and all the communications are local. It can also be implemented in microprocessor level, for example, using a transputer network. The partition of the systolic algorithm may require to use every transputer efficiently.

In a real time array processing system, the system implementation is expensive due to the computationally intensive high resolution methods. The systolic architecture while implemented by VLSI/VHSIC certainly provides a possible solution for the problem.

\section*{Chapter 10}

\section*{A LINEAR SYSTOLIC ARRAY FOR CONJUGATE GRADIENT METHOD}

\subsection*{10.1 Introduction}

THE solution of an \(n \times n\) symmetric positive definite (SPD) system \(A x=b\) arises in many applications such as digital signal processing (DSP) algorithms and optimization theory. When used in DSP the matrix \(\boldsymbol{A}\) is often known as an auto-correlation matrix and \(\boldsymbol{b}\) is a cross correlation vector. The system can be solved by directed methods based on matrix decompositions such as Cholesky decomposition. The system can be also solved by iterative methods. Among different iterative methods, the conjugate gradient (CG) method is very attractive as it can guarantee a convergence within \(/ /\) iterations if precise computation can be carried out through the iterative process. The preconditioned conjugate gradient method (PCG) can further increase the convergence rate by constructing an appropriate preconditioner. The solution of \(x\) by the conjugate gradient (CG) method can be also used as a minimization technique [Gill81] [Modi88] [Golub89].

The CG and PCG algorithms are complex algorithms involving matrix-vector multiplication, vector dot product and other scalar operations. The data flow of such algorithms are complicated. In this chapter, we apply the SFG design method [KungSY89] to deriving a regular DG for the CG method. Based on the obtained DG, a novel efficient linear systolic array for the CG method is proposed. Furthermore, we incorporate a simple preconditioner into the CG method to form a linear systolic array for the PCG method.

The outline of this chapter is as follows. The CG algorithm for solving a SPD system of equation is reviewed in Sec. 10.2. Then we design a regularized dependence graph (DG) for the CG algorithm in Sec. 10.3. Based on the obtained DG, we design a linear systolic array for CG method in Sec. 10.4. Preconditioned conjugate gradient (PCG) method is discussed in Sec. 10.5 where a simple diagonal preconditioner strategy is implmented using systolic array. Finally Sec. 10.6 concludes this chapter.

\subsection*{10.2 Conjugate Gradient Method}

We consider to solve the system of linear equations as follows
\[
\begin{equation*}
A x=b \tag{10.1}
\end{equation*}
\]
where \(A\) is an \(n \times n\) positive definite and symmetric matrix, and \(x\) and \(b\) are \(n \times 1\) vectors. A matrix \(A\) is positive definite if \(\boldsymbol{x}^{T} \boldsymbol{A x}>0\) for all nonzero \(n \times 1\) vector \(\boldsymbol{x}\).

The starting point in the derivation of the conjugated gradient (CG) method is to minimize the objective function \(\phi(x)\), defined by
\[
\begin{equation*}
\phi(x)=\frac{1}{2} x^{T} A x-x^{T} b \tag{10.2}
\end{equation*}
\]

The minimum of \(\phi\) is \(-b^{T} A^{-1} b / 2\), achieved by setting \(\boldsymbol{x}=\boldsymbol{A}^{-1} \boldsymbol{b}\). Thus, minimizing \(\phi\) and solving Eq. (1) are equivalent problems.

One of the simplest strategies for minimizing \(\phi\) is the method of steepest descent [Golub89]. At a current point \(\boldsymbol{x}_{\boldsymbol{c}}\) the function \(\phi\) decreases most rapidly in the direction of the negative gradient \(-\nabla \phi\left(x_{c}\right)=\boldsymbol{b}-\boldsymbol{A} \boldsymbol{x}_{\boldsymbol{c}}\). We call
\[
\begin{equation*}
r_{c}=b-A x_{c} \tag{10.3}
\end{equation*}
\]
the residual of \(\boldsymbol{x}_{\boldsymbol{c}}\). If the residual is nonzero, then there exists a positive \(\alpha\) such that \(\phi\left(x_{c}^{\prime}+\alpha r_{c}\right)<\phi\left(x_{c}\right)\). In the method of steepest descent, we set \(\alpha=r_{c}^{r} r_{c} / r_{c}^{T} A r_{c}\) thereby minimizing \(\phi\left(x_{c}+\alpha r_{c}\right)\). This leads to the steepest descent algorithm:
```

Algorithm 1: The steepest descent method
$k=0 ; x_{0}=0 ; r_{0}=\boldsymbol{b}$;
while $\boldsymbol{r}_{k} \neq 0$
$k=k+1$
$\alpha_{k}=r_{k-1}^{T} r_{k-1} / r_{k-1}^{T} A r_{k-1}$
$x_{k}=x_{k-1}+\alpha_{k} r_{k-1}$
$r_{k}=\boldsymbol{b}-\boldsymbol{A} \boldsymbol{x}_{k}$
end

```

When the level curves of \(\phi\) are very elongated hyperellipsoids, the minimization corresponds to finding the lowest point on a relatively flat, steep-sided valley. In this situation, the steepest descent method will traverse back and forth across the valley rather than down the valley. To avoid this pitfall of steepest descent, we consider successive minimization of \(\phi\) along a set of directions \(\left\{\boldsymbol{p}_{1}, \boldsymbol{P}_{2}, \cdots\right\}\) that do not necessarily correspond to the residuals \(\left\{\boldsymbol{r}_{1}, r_{2}, \cdots\right\}\). It is easy to show that to minimize \(\phi\left(\boldsymbol{x}_{k-1}+\alpha p_{k}\right)\) with respect to \(\alpha\), we merely set
\[
\begin{equation*}
\alpha=\alpha_{k}=p_{k}^{T} r_{k-1} / p_{k}^{T} A p_{k} \tag{10.4}
\end{equation*}
\]

It can be shown that \(\boldsymbol{p}_{k}\) must not be orthogonal to \(\boldsymbol{r}_{k-1}\) in order to ensure a reduction in the value of \(\phi\). This leads to the following minimization strategy:
```

Algorithm 2: A general search strategy
$k=0 ; x_{0}=0 ; r_{0}=\boldsymbol{b}$,
while $\boldsymbol{r}_{\boldsymbol{k}} \neq 0$
Choose a direction $\boldsymbol{p}_{k}$ such that $\boldsymbol{p}_{k_{k-1}}^{T} \not{ }^{T} \neq 0$
$\alpha_{k}=P_{k}^{T} r_{k-1} / p_{k}^{T} A p_{k}$
$x_{k}=x_{k-1}+\alpha_{k} P_{k}$
$r_{k}=b-A x_{k}$
end

```

The problem is how to choose these search vectors so as to guarantee global convergence and at the same time to avoid the pitfalls of steepest descent. The conjugate gradient method chooses linearly independent vectors \(\boldsymbol{p}_{\boldsymbol{i}}\) with the property that each \(\boldsymbol{x}_{\boldsymbol{k}}\) in Algorithm 2 solves
\[
\begin{equation*}
\min _{x \in \operatorname{span}\left\{p_{1}, \cdots, F_{k}\right\}} \phi(x) \tag{10.5}
\end{equation*}
\]

This would guarantee not only global convergence but finite termination as well because we must have \(\boldsymbol{A} \boldsymbol{x}_{n}=\boldsymbol{b}\). Let \(\boldsymbol{P}_{\boldsymbol{k}}=\left[\boldsymbol{P}_{1}, \ldots \boldsymbol{P}_{k}\right] \in \mathrm{R}^{n \cdot k}\) be the matrix of search directions. If \(\boldsymbol{x} \in \operatorname{range}\left(\boldsymbol{P}_{\boldsymbol{k}}\right)\) then \(\boldsymbol{x}=\boldsymbol{P}_{\boldsymbol{k}-\boldsymbol{y}} \boldsymbol{y}+\alpha \boldsymbol{p}_{\boldsymbol{k}}\) for some \(\boldsymbol{y} \in \mathbf{R}^{k-1}\) and \(\alpha \in \mathbf{R}\). If \(\boldsymbol{x}\) has this form then it is easy to show that
\[
\begin{equation*}
\phi(x)=\phi\left(P_{k-1} y\right)+\alpha y^{T} P_{k-1}^{r} A p_{k}+\frac{\alpha^{2}}{2} p_{k}^{T} A p_{k}-\alpha p_{k}^{r} b \tag{10.6}
\end{equation*}
\]

The presence of the "cross term" \(\alpha y^{T} P_{k-1}^{T} A p_{k}\) complicates the minimization. Without it the minimization of \(\phi\) over range \(\left(P_{k}\right)\) would decouple into a minimization over range \(\left(P_{k-1}\right)\), whose solution \(\boldsymbol{x}_{\boldsymbol{k}-1}\) is assumed known, and a similar minimization involving the scalar \(\alpha\). One way to effect this decoupling is to insist that \(p_{k}\) is \(\boldsymbol{A}\)-conjugate to \(\boldsymbol{p}_{1}, \ldots, \boldsymbol{p}_{k-1}\) meaning that
\[
\begin{equation*}
P_{k-1}^{T} A p_{k}=0 \tag{10.7}
\end{equation*}
\]

Applying this search direction to Algorithm 2, we obtain
```

$k=0 ; x_{0}=0 ; r_{0}=\boldsymbol{b} ;$
while $\boldsymbol{r}_{\boldsymbol{k}} \neq 0$
$k=k+1$
Choose $\boldsymbol{p}_{k} \in \operatorname{span}\left\{A p_{1}, \ldots, A p_{k-1}\right\}^{\perp}$ such that $\boldsymbol{p}_{k^{2}}^{T} r_{k-1} \neq 0$
$\alpha_{k}=p_{k}^{T} r_{k-1} / p_{k}^{T} A p_{k}$
$x_{k}=x_{k-1}+\alpha_{k} p_{k}$
$r_{k}=b-A x_{k}$
end

```

Since our aim is to bring about the swift reduction in the size of residuals, it is natural to choose \(\boldsymbol{p}_{\boldsymbol{k}}\) to be the closest vector to \(r_{k-1}\) that is \(\boldsymbol{A}\)-conjugate to \(\boldsymbol{p}_{1}, \ldots, \boldsymbol{p}_{k-1}\). This defines the method of conjugate gradients:
```

$k=0 ; x_{0}=0 ; r_{0}=b ;$
while $\boldsymbol{r}_{k} \neq 0$
$k=k+1$
if $k=1$
$p_{1}=r_{0}$
else
Choose $p_{k}$ minimize $\left\|p-r_{k-1}\right\|_{2}$ over all vectors $p \in \operatorname{span}\left\{A p_{1}, \ldots, A p_{k-1}\right\}^{\perp}$
end
$\alpha_{k}=p_{k}^{T} r_{k-1} / p_{k}^{T} A p_{k}$
$x_{k}=x_{k-1}+\alpha_{k} p_{k}$
$r_{k}=b-A x_{k}$
end

```

An efficient algorithm to implement the conjugate gradient method was originally introduced by Hestenes and Stiefel [Hestenes52] [Golub89]. The algorithm has the following form:
```

Algortihm 3: Conjugate gradient (C.G) method
$k=0 ; \boldsymbol{x}_{0}=0 ; \boldsymbol{r}_{0}=\boldsymbol{b}$;
while $\boldsymbol{r}_{\boldsymbol{k}} \neq 0$
$k=k+1$
if $k=1$
$p_{1}=r_{0}$
else
$\beta_{k}=r_{k-1}^{T} r_{k-1} / r_{k-2}^{T} r_{k-2}$
$p_{k}=r_{k-1}+\beta_{k} p_{k-1}$
end
$\alpha_{k}=r_{k-1}^{T} r_{k-1} / p_{k}^{T} A p_{k}$
$x_{k}=x_{k-1}+\alpha_{k} p_{k}$
$r_{k}=r_{k-1}-\alpha_{k} A p_{k}$
end

```

The termination criteria in Algorithm 3 is unrealistic. Rounding errors leads to a loss of orthogonality among the residuals and finite termination is not mathematically guaranteed. Moreover, when the conjugate gradient method is applied, \(n\) is usally so large that \(O(n)\) iterations represents an unacceptable amount of work. As a consequence of these observations, it is customary to regard the method as a genuinely iterative technique with termination based upon an iteration maximum \(k_{\text {max }}\) and the residual norm. This leads to the following practical version of Algorithm 3.
```

Algortihm 4: Conjugate gradient (CG) method (practical implementation)
$k=0 ; \boldsymbol{x}=0 ; r=\boldsymbol{b} ; \rho_{0}=\|r\|_{2}^{2}$;
while $\sqrt{\rho_{k}>\varepsilon\|b\|_{2}} \begin{aligned} & \text { and } k<k+1 \\ & \text { max }\end{aligned}$
if $k=1$
$p=r$
else
$\beta_{k}=\rho_{k-1} / \rho_{k-2}$
$p=r+\beta_{k} p$
end
$\boldsymbol{w}=A p$
$\alpha_{k}=\rho_{k-1} / p^{T} w$
$x=x+\alpha_{k} p$
$\boldsymbol{r}=\boldsymbol{r}-\alpha_{2} \boldsymbol{k}$
$\rho_{k}=\|\boldsymbol{k}\|^{2}$
end

```

This algorithm requires one matrix-vector multiplication ( \(n^{2}\) multiplicative additive operations or flops) and another \(5 n\) flops per iteration.

\subsection*{10.3 DG Design for CG Method}

\subsection*{10.3.1 Algorithm Modification}

In order to design a DG (Dependence Graph) for the CG method, Algorithm 4 needs some modifications. First, the "if-else" for the searching direction \(p\) computation can be eliminated by setting \(\boldsymbol{P}_{0}=0\). Secondly the computation of \(\boldsymbol{\rho}_{k}\) can be performed in a different order. This will result in the following algorithm:
```

Algortihm 5: Conjugate gradient (CG) method
$x_{0}=0 ; r_{0}=b ; \rho_{0}=0 ; p_{0}=0$;
for $k=1,2, \ldots$
(1) $\rho_{k}=r_{k-1}^{T} r_{k-1}$

```
```

(2) $\beta_{k}=\rho_{k} / \rho_{k-1}$
(3) $p_{k}=r_{k-1}+\beta_{k} p_{k-1}$
(4) $w_{k}=A_{r} p_{k}$
(5) $\delta_{k}=\boldsymbol{p}_{k}^{T} \boldsymbol{w}_{k}$
(()) $\alpha_{k}=\rho_{k} / \delta_{k}$
(7) $x_{k}=x_{k-1}+\alpha_{k} p_{k}$
(8) $r_{k}=r_{k-1}-\alpha_{k} \boldsymbol{w}_{k}$
end

```

Then the single assignment code for Algorithm 5 can be obtained as follows:
Algorithm 6. Single assignment code for Algorithm 5
for \(\mathrm{k}=1,2\),
for \(i=1: n\)
(1) rho( \(i, k)=r h o(i-1, k)+r(i, k-1)^{*} r(i, k-1)\)
end
(2) \(\quad \operatorname{beta}(\mathrm{k})=\mathrm{rho}(\mathrm{n}, \mathrm{k}) / \mathrm{rho}(\mathrm{n}, \mathrm{k}-\mathrm{I})\)
for \(\mathrm{i}=1\) : \(n\)
(3) \(\quad \mathrm{p}(\mathrm{i}, \mathrm{k})=\mathrm{r}(\mathrm{i}, \mathrm{k}-\mathrm{l})+\mathrm{beta}(\mathrm{k})^{*} \mathrm{p}(\mathrm{i}, \mathrm{k}-\mathrm{l})\)
end
for \(\mathrm{i}=1 \mathrm{n}\)
\[
\text { for } j=1: n
\]
(4)
\[
w(i, j, k)=w(i, j-l, k)+a(i, j)^{*} p(j, k)
\]
end
(5) \(\quad \operatorname{delta}(i, k)=\operatorname{delta}(i-I, k)+p(i, k)^{*} w(i, n, k)\)

> end
(6) \(\quad \operatorname{alpha}(\mathrm{k})=\mathrm{rho}(\mathrm{n}, \mathrm{k}) / \operatorname{delta}(\mathrm{n}, \mathrm{k})\)
for \(\mathrm{i}=1: \mathrm{n}\)
(8)
\[
\begin{align*}
& x(i, k)=x(i, k-1)+a \operatorname{lpha}(k)^{*} p(i, k)  \tag{7}\\
& r(i, k)=r(i, k-1)-a \operatorname{lpha}(k)^{*} w(i, n, k)
\end{align*}
\]
end
end
with rho \((0, k)=0\), rho \((\mathrm{n}, 0)=\rho_{0}=0, w(\mathrm{i}, 0, \mathrm{k})=0\), delta \((0, \mathrm{k})=0\) for all \(i\) and \(k\).
Notice that, in Algorithm 5, the vector dot prodects and scalar operations in lines (2), (5) and (6) are separated by the matrix/vector operation in the remaining lines. This may cause a problem for the parallel implementation of the algorithm due to a longer critical path. We will analyze the critical path of the algorithm in next subsection and we now consider another variant of Algorithm 5. The modified algorithm has the following form:

Algortihm 7: Conjugate gradient (CG) method
\(x_{0}=0 ; r_{0}=b ; \rho_{0}=r_{0}^{T} r_{0} ; p_{0}=b\) :
for \(k=1,2, \ldots\)
(1) \(\boldsymbol{w}_{k}=A P_{k-1}\)
(2) \(\delta_{k}=p_{k-T}^{T} w_{k}\)
(3) \(\epsilon_{k}=w_{k}^{T} w_{k}\)
(4) \(\alpha_{k}=\rho_{k-1} / \delta_{k}\)
(5) \(\quad \beta_{k}=\alpha_{k} * \epsilon_{k} / \delta_{k}-1\)
(6) \(\rho_{k}=\rho_{k-1} * \beta_{k}\)
(7) \(x_{k}=x_{k-1}+\alpha_{k} p_{k}\)
(8) \(r_{k}=r_{k-1}-\alpha_{k} w_{k}\)
(()) \(P_{k}=r_{k}+\beta_{k} p_{k-1}\)
end
The formula for \(\beta_{k}\) in the above algorithm can be obtained from Algorithm 5 by using two rules used in the dirivation of the CG method. The first rule is that \(p_{k}\) is \(A\)-conjugate to \(p_{1}, \ldots, p_{k-1}\), i.e., \(\boldsymbol{P}_{k-1}^{T} \boldsymbol{A} \boldsymbol{P}_{\boldsymbol{k}}=\mathbf{0}\). The second rule is that the \(k\) th residual is othogonal to the \(k\) th searching matrix, i.e., \(\boldsymbol{P}_{k_{k}}^{T} r_{k}=0\). The single assignment code for Algorithm 7 then can be given as follows.

Algorithm 8. Single assignment code for Algorithm 7
for \(\mathrm{k}=1,2, \ldots\)
\[
\begin{aligned}
& \text { for } \mathrm{i}=1: n \\
& \quad \text { for } \mathrm{j}=1: \mathrm{n}
\end{aligned}
\]
\[
\begin{equation*}
w(i, j, k)=w(i, j-1, k)+a(i, j)^{*} p(j, k-1) \tag{1}
\end{equation*}
\]
end
(2) delta(i, \(k)=\operatorname{delta}(i-1, k)+p(i, k-I)^{*} w(i, n, k)\)
(3) epsilon(i,k)=epsilon(i-l, \()+w(i, n, k)^{*} w(i, n, k)\)
end
(4) \(\operatorname{alpha}(\mathrm{k})=\mathrm{rho}(\mathrm{k}-1) / \mathrm{delta}(\mathrm{n}, \mathrm{k})\)
(5) \(\quad \operatorname{beta}(\mathrm{k})=\) alpha \((\mathrm{k})^{*}\) epsilon \((\mathrm{n}, \mathrm{k}) /\) delta( \(\left.\mathrm{n}, \mathrm{k}\right)\)-l
(6) \(\quad\) rho \((k)=r h o(k-1) * \operatorname{beta}(k)\)
for \(\mathrm{i}=1\) : n
(7) \(\quad x(i, k)=x(i, k-1)+a \operatorname{lpha}(k)^{*} p(i, k-1)\)
(8) \(\quad r(i, k)=r(i, k-1)\)-alpha(k)* \(w(i, n, k)\)
(a) \(\quad p(i, k)=r(i, k)+b e t a(k)^{*} p(i, k-1)\)
end
end
with delta(i, 0\()=0\), epsilon( \(\mathrm{i}, 0)=0, \mathrm{w}(\mathrm{i}, 0, \mathrm{k})=0\) for all \(i\) and \(k\).

\subsection*{10.3.2 Dependece Graph Design}
10.3.2.1 DGs for the Two CG

\section*{Algorithms}

We consider the single assigment codes for the CG method in Algorithms 6 and 8 discussed earlier. Their dependece graphs for a single iteration (in \(k\) direction) can be easily obtained by connecting different


Figure 10.1 DG for Algorithm 6
loops together. Figure 10.1 shows the DG for Algorithm 6 .

In Figure 10.1, the different operations corresponding to those in Algorithm 6 are grouped and numbered accordingly. The DG for operation (1) receives the input \(r_{k-1}\) and accumulates the result \(\boldsymbol{\rho}_{\boldsymbol{k}}\). The computation of \(\beta_{k}\) is a simple scalar operation denoted by the node numbered as (2) in the DG. Operation (3) requires the input \(\boldsymbol{r}_{k-1}\), which is propagated from the nodes for operation (1), and the input \(\beta_{k}\), which is obtained from node (2) and propagated downwards. The result of the operation (3) is delivered to the top nodes for operation (4) while it is also propageted horizontally as required by operations (5) and (7). The DG for operation (4) is actually the one for matrix-vector multiplication. The elements of matrix \(A\) are sent to the individual notes although they are not drawn for the sake of simplicity. The result \(\boldsymbol{w}_{k}\) is available at the right side of the DG for operation (4). The computation of \(\delta_{k}\) is performed by the nodes in the group (5) where both \(p_{k}\) and \(w_{k}\) are provided as inputs and are propagated to the right for operations (7) and (8). The computation of \(\alpha_{k}\) is very straightforward as indicated by node (6). Operations (8) and (8), which update \(x\) and \(r\), are dependent on the value \(\boldsymbol{\alpha}_{\boldsymbol{k}}\). Finally, \(\boldsymbol{x}_{\boldsymbol{k}}\) and \(\boldsymbol{r}_{\boldsymbol{k}}\) are available at the nodes in groups (7) and (8) respectively.

Based on Algorithm 8, we can obtain its DG as shown below in Figure 10.2. The nodes in group (1) perform the matrix-vector multiplication while the nodes in groups (2), (3), (7), (8) and (9) perform the vector operations. Nodes (4), (5) and (6) perform the scalar operations.

Observing the two DGs in Figures 10.1 and 10.2 , we find that both DGs have a certain degree of regularity while some irregular connections exist indeed. Some modification is therefore required in order to obtain a DG more suitable for systolic mapping. Before we proceed further, we compare the critical paths, or


Figure 10.2 DG for Algorithm 8 the lengths of the longest paths, of two DGs. The DG for Algorithm 6 in Figure 10.1 has a critical
path of \(4 n+4\) while the DG for Algorithm 8 in Figure 10.2 has a critical path of \(3 n+3\). One of the longest paths for each DG is represented by thick arrows in Figures 10.2 and 10.3. Obviously, the DG for Algorithm 8 has a shorter critical path and it provides a potential for a better systolic design. In the following discussion, we will focus on the DG for Algorithm 8 in Figure 10.2.

\subsection*{10.3.2.2 Modification to the DG for CG Algorithm 8}

We now consider the DG in Figure 10.2 which has a shorter critical path than the DG in Figure 10.1. First, we observe that \(\boldsymbol{P}_{k-1}\) enters into the DG array from top side and \(\boldsymbol{p}_{k}\) is available at the right side. This makes it difficult to cascade DGs between adjacent iterations along \(k\) directions. We therefore rotate the group (1) DG. This rotation also causes \(\boldsymbol{w}_{k}\) being rotated. In order to inject \(\boldsymbol{w}_{\boldsymbol{k}}\) into the rest of the DG array, we plug in a new group of notes denoted by and this gives the new DG in Figure 10.3. The diagonal nodes in the group (10) transfer the vertical data to horizontal data thereby making \(\boldsymbol{w}_{\boldsymbol{k}}\) available in a desired way.


Figure 10.3 Modified DG for Algorithm 8

The modified DG in Figure 10.3 is nearly regular along the horizontal direction. There exists irregular connection between the top side of the array and the bottom side of the array. This irregularity can be alleviated by choosing an approprite projection direction when mapping to an SFG array.

Now we consider the irregularity of nodes (4), (5) and (©). More nodes with a function of only transmitting data can be attached in order to make the DG regularized. It is also observed that node (5) involves one multiplication and one division while all the other nodes involve only one multiplication or division. Thus node (5) may introduce some performance degradation when mapping to a systolic array. We divide node (5) into two operations or two nodes as given below:
(5) \(\quad \operatorname{mu}(\mathrm{k})=\mathrm{alpha}(\mathrm{k})^{*}\) epsilon(k)
(5)" \(\quad \operatorname{beta}(\mathrm{k})=m u(\mathrm{k}) / \operatorname{delta}(\mathrm{n}, \mathrm{k})-1\)

A more regularized DG with the above modifications is shown in Figure 10.4.


Figure 10.4 A more regularized DG for Algorithm 8

The DG in Figure 10.4 is still not fully regularized as the dependencies from bottom nodes to top nodes are irregular. Examing the top-left node in group (10), we find that the node has two indirect dependencies on the top-left node in group (1). These two indirect dependencies form two paths of different lengths in the DG. The horizontal path has a length of ( \(n-1\) ) while the vertical path together with the bottom-top connection has a length of \(n\). This suggests to us to plug in one column delay nodes between groups (1) and (10. Similarly, ( \(11-1\) )-column delay nodes can be added between groups (3) and (7). Finally we obtain a regularized DG as shown in Figure 10.5. This DG is ready for mapping if we choose a horizontal projection direction. All the non-local communication can be eliminated with this projection. In the following section, we will first map the DG into an SFG and then systolize the obtained SFG so as to design a systolic architecture for the CG method.


Figure 10.5 Regularized DG for Algorithm 8

\subsection*{10.4 Systolic Array Design for CG Method}

\subsection*{10.4.1 Mapping the \(D G\) onto \(S F G\)}

In order to exploit the regularity of the DG in Figure 10.5, a natural projection direction is the horizontal direction. By choosing such a projection direction, a regular SFG array can be obtained. Using the \(i-j\) coordinate system shown in Figure 10.5, the projection direction can be given as
\[
d^{T}=\left[\begin{array}{ll}
1 & 0 \tag{10.8}
\end{array}\right]
\]

We then choose a default schedule which can be represented by the schedule vector given as
\[
s^{T}=d^{T}=\left[\begin{array}{ll}
1 & 0 \tag{10.9}
\end{array}\right]
\]

The processor basis \(P\) corresponding to the above projection vector is as follows
\[
P^{T}=\left[\begin{array}{ll}
0 & 1 \tag{10.10}
\end{array}\right]
\]

The node mapping can be given by
\[
\boldsymbol{P}^{T}\left[\begin{array}{l}
i  \tag{10.11}\\
j
\end{array}\right]=\left[\begin{array}{ll}
0 & 1
\end{array}\right]\left[\begin{array}{l}
i \\
j
\end{array}\right]=j
\]

Eq. (10.11) means that the \((i, j)\) th node in the DG is mapped into the \(j\) th mode in the SFG, that is, each row of nodes in the DG is mapped into a single node in the SFG array. There are three types of arcs in the DG: horizontal arcs, vertical arcs and the bottom-to-top arcs, which can be represented by three vectors respectively as follows
\[
e_{h}^{T}=\left[\begin{array}{ll}
1 & 0
\end{array}\right], e_{v}^{T}=\left[\begin{array}{ll}
0 & 1
\end{array}\right] \text { and } e_{b t}^{T}=\left[\begin{array}{ll}
n+1 & -n \tag{10.12}
\end{array}\right]
\]

The arc mapping maps the above arcs of the DG to the edges of the SFG together with the number of delays on every edge. This mapping can be described as follows
\[
\left[\begin{array}{l}
s^{T}  \tag{10.13}\\
P^{T}
\end{array}\right]\left[\begin{array}{lll}
e_{h} & e_{v} & e_{b d f}=\left[\begin{array}{ll}
1 & 0 \\
0 & 1
\end{array}\right]\left[\begin{array}{ccc}
1 & 0 & (n+1) \\
0 & 1 & -n
\end{array}\right]=\left[\begin{array}{ccc}
1 & 0 & (n+1) \\
0 & 1 & -n
\end{array}\right], ~\left(\begin{array}{ll}
-n
\end{array}\right)
\end{array}\right.
\]

The second row of the right hand side of Eq. (10.13) represents the edges of the SFG corresponding to the three arcs in the DG. The first row of the the right hand side of Eq. (10.13) provides the number of delays for the three types of the edges in the SFG. The SFG array from the mapping given above is shown in Figure 10.6.

The SFG array consits of \((n+1)\) nodes with the head and the tail nodes connecting together to form a ring of nodes. The top \(n\) nodes perform the matrix and vector computations while the bottom node performs the scalar operations in Algorithm 8. The SFG array receives the input matrix \(\boldsymbol{A}\) from the left followed by a certain number of delays. Zeros are also injected to the SFG array at the top node. The vectors \(w, \boldsymbol{x}, \boldsymbol{r}\) and \(\boldsymbol{p}\) are stored in the


Figure 10.6 Selection of cut-set to systolize the SFG SFG array. The number of delays for each edge in the SFG array is also indicated in Figure 10.6. It is noticed that the SFG array is not systolized since the vertical edges have zero delays. Below we will systolize the SFG array in Figure 10.6 to derive a systolic array for the CG method.

\subsection*{10.4.2 Systolization of the \(\operatorname{SFG}\)}

Now we apply the cut-set retiming procedure to the the obtained SFG array in Figure 10.6. The objective of the cut-set retiming procedure is to convert the SFG into a temporally localized form so that all the edges between nodes have at least one delay element.

The cut-sets we choose are also depicted in Figure 10.6. By applying the cut-set retiming procedure (detailed in Chap. 4), we can obtain a temporally localized SFG. For each cut-set we add one delay to the inbound edges (downwards edges and rightwards edges) while substracting one delay to the outbound edge (the upwards edge). Therefore all the downwards edge will possess one delay. By counting the number of cut-


Figure 10.7 Temporally localized SFG array sets cutting the upwards edgs, \(I\) delays should be subtracted from the original delay. Thereby, the upwards edges will leave only one delay after retiming. The input data of matrix \(A\) is also skewed after retiming. The obtained temporally localized SFG is shown in Figure 10.7.

\subsection*{10.4.3 Systolic Arrays Design}

To convert the temporally and spatially localized SFG as shown in Figure 10.7 into a systolic form, it is only necessary to sucessfully introduce a delay into each of the operataion mudules. It is straightforward for us to obtain the systolic array from the SFG array. Figure 10.8 shows the systolic array for the CG method.

It is noticed that we show only one iteration of the CG method. However, the systolic array in Figure 10.8 is fully pipelinable. Thus more iterations of the CG method can be pipelined to pass through the systolic array again and again until a designed maximum number of iterations has been executed or the algorithm converges to the desired precision.

Now we consider the detailed definition of PEs in the systolic array. By observing the DG in Figure 10.5, we notice that during each iteration of the CG method, each PE of the systolic array should execute all the tasks of those nodes along the corresponding horizontal line in the DG. Since all the nodes in a sngle horizontal line in the DG are assigned to a single PE in the systolic array, the PE is required to perform all the tasks


Figure 10.8 Systolic array for the CG method represented by those nodes. It is clear from the DG in Figure 10.5 that each PE in the systolic array needs to perform different functions in the same iteration of the CG method. In Figure 10.5, we have divided the different functions into 11 phases. Suppose that we have a control signal propagating in the systolic array. Then PEs in the systolic array will perform appropriate functions according to processing phase indicated by the control signal. Figure 10.9 shows the systolic array with an extra vertical control channel as well as the necessary control sequence.

Another point requiring more elaboration is the operations involved in transposing vector \(w\) in phase 3 as shown in Figure 10.5. Because the diagonal nodes in group (il) have different functions from the other nodes in the same group of nodes, we should have a mechenism to specify whether the PE should change the data direction or not. Notice that the first


Figure 10.9 Systolic array for the CG method diagonal node in group (10) changes the vertical input data into horizontal output at the Ist time step of phase 3, and the second diagonal node in group (10) change the input data into horizontal output two time steps later, and so forth. We
synchronise these operations by introducing a counter to each PE. The counter in the first PE (top) is initialised with a value of 1 , the counter in the second PE is initialised with a value of two and so forth.

The detailed PE definitions taking into account the aforementioned two points are given in Table 10.1. The number of systolic steps for each phase is also presented in Table 10.1.

Table 10.1 PE definitions of systolic array for CG method
\begin{tabular}{|c|l|l|l|l|}
\hline Phase & & & \\
\hline
\end{tabular}

It takes, in total, \(3 n+5\) systolic steps for the systolic array to perform each iteration of the CG method. Therefore the proposed linear systolic array has a linear pipeline rate because each iteration has a complexity of \(O\left(r^{2}\right)\). It is also obvious that the systolic array can immediately start the next iteration after completing the previous iteration. Assuming that the total number of iteration required for CG
method is \(K\), the total time required for the CG method is \(K(3 \mu+5)\) plus a small amount of overhead due to the skewed format.

\subsection*{10.5 Preconditioned Conjugate Gradient (PCG) Method}

\subsection*{10.5.1 PCG Algorithm}

It is known that the method of conjugate gradients works well on matrices that are either well conditioned or have just a few distinct eigenvalues. If these conditions are not satisfied, the CG method may converge very slowly. The preconditioned conjugate gradient method is designed to accelerate the CG method by solving a revised equivalent system with a smaller condition number.

The idea behind PCG method is to apply the "regular" CG method to the transformed system
\[
\begin{equation*}
\tilde{A} \hat{x}=\tilde{b} \tag{10.14}
\end{equation*}
\]
rather than the original system in Eq. (10.1). Here \(\tilde{A}=C^{-1} A C^{-1}, \tilde{x}=C x, \tilde{b}=C^{-1} b\), and \(C\) is symmetric positive definite. If we define the preconditioner \(M\) by \(M=C^{2}\) (also positive definite), the PCG algorithm corresponding to the CG algorithm in Algorithm 7 can be written as follows.

Algortihm 9: Preconditioned conjugate gradient (PCG) method


An appropriate preconditioner for the above algorithm should have the following properties [Evans93c] [Evans92][Golub89]:
(a) \(\boldsymbol{M}\) is symmetric positive definite;
(b) \(\boldsymbol{M}\) is easy to construct;
(c) \(\boldsymbol{M}\) allows a fast computation of \(\epsilon_{k}=\boldsymbol{w}_{k}^{T} \boldsymbol{M}^{-1} \boldsymbol{w}_{k}\) as well as \(\boldsymbol{z}_{k}=\boldsymbol{M}^{-1} \boldsymbol{r}_{k}\); and
(d) "good" distribution of the eigenvalues of \(M^{-1} A\).

The simplest preconditioner is the diagonal matrix
\[
\begin{equation*}
M=\operatorname{diag}(A) \tag{10.15}
\end{equation*}
\]

Although the above diagonal preconditioner is extremely simple, it can sometimes bring about a significant improvement on convergence rate for certain problems. For example, we consider a linear system where \(A\) is a Pascal matrix. The Pascal matrix \(A\) is made up from Pascal's triangle and it is a positive definite and symmetric matrix. The Pascal matrix has the


Figure 10.10 Convergence rates of CG and PCG methods following form:
\[
A=\left[\begin{array}{cccccc}
1 & 1 & 1 & 1 & 1 & \ldots  \tag{10.16}\\
1 & 2 & 3 & 4 & 5 & \ldots \\
1 & 3 & 6 & 10 & 15 & \cdots \\
1 & 4 & 10 & 20 & 35 & \cdots \\
1 & 5 & 15 & 35 & 70 & \cdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \ddots
\end{array}\right]
\]

We apply both the CG and PCG methods to the 20th order Pascal system. Figure 10.10 shows the 2 -norm of the difference between the approximate solution and exact solution versus the number of iterations. As the system is ill-conditioned, the CG method does not converge to a desired precision within 20 (the order of the system) iterations. Figure 10.10 also clearely indicates that the PCG method may converge much faster than the C.G method even though a very simple preconditioner is applied.

\subsection*{10.5.2 Systolic Array for PCG Method}

In this section, we consider the systolic implementation for the simplest preconditioning strategy using a diagonal preconditioner while other appropriate preconditioning strategy can be also implemented.


Figure 10.11 Regularized DG for PCG method with a diagonal preconditioner

With the introduction of the diagonal preconditioner, Algorithm 9 will be slightly more complex than the original CG method in Algorithm 7. Similarly we can obtain a regularized DG for the PCG method with a diagonal preconditioner as shown in Figure 10.11. Accordingly the DG can be mapped into a spatially and temporally localized SFG, which is the same as the one shown in Figure 10.7. Then we can obtain a systolic array for the PCG method with a diagonal preconditioner. The array has a same structure as shown in Figure 10.9 except that the control sequence is slightly changed. The new PE definition for the PCG method with a diagonal preconditioner can be given as follows in Table 10.2. The diagonal elements of the inverse of the preconditioner is stored in the individual PEs. Care has been taken so that at every time step each PE performs operations having more or less a complexity of inner product step (ips).

Other types of preconditioners are also possible to be implemented with similar design method. Algorithm 9 may be modified so that computations in each iteration involve the application of one preconditioner[Golub89]. A general guideline for choosing a preconditioner is to allow the solution of \(\boldsymbol{M z}=\boldsymbol{r}\) in \(O(n)\) steps or less ( \(O(1)\) steps for the simple diagonal preconditioner) by using n linearly connected processor elements. In this way, the obtained systolic array can complete each iteration in \(O(n)\) steps and thus the systolic array has a linear speedup.

Table 10.2 PE definitions of systolic array for PCG method with diagonal preconditioner
\begin{tabular}{|c|c|c|c|}
\hline Phase &  &  & Steps \\
\hline 1 & \(\theta_{\text {oul }}=\theta_{\text {in }}+\mathrm{a}_{\text {in }} \times \mathrm{p}\) & NOP & " \\
\hline 2 & \(\mathrm{c}=\mathrm{c} 0\) & NOP & 1 \\
\hline 3 & \(\mathrm{c}=\mathrm{c}-1\); if \(\mathrm{c}=0\) then \(\mathrm{w}=\theta_{\text {in }}\) & NOP & n \\
\hline 4 & \(\theta_{\text {oul }}=\theta_{\text {in }}+\mathrm{p} \times \mathrm{w}\) & \(\delta=\theta_{\mathrm{in}} ; \alpha=\rho / \delta ;\) & 1 \\
\hline 5 & \(\mathrm{z}=\mathrm{w} \times \mathrm{w}\) & \(\theta_{\text {oul }}=\alpha\) & 1 \\
\hline 6 & \(\theta_{\text {our }}=\theta_{\text {in }}+\mathrm{t} \times \mathrm{m}\) & \(\mu=\alpha \times \theta_{\text {in }} ; \theta_{\text {out }}=\alpha\) & 1 \\
\hline 7 & NOP & \(\beta=\mu / \delta-1\) & 1 \\
\hline 8 & NOP & \(\rho=\rho \times \beta ; \theta_{\text {out }}=\beta\) & 1 \\
\hline 9 & NOP & NOP & n-3 \\
\hline 10 & \(\theta_{\text {out }}=\theta_{\text {in }} ; x=x+\theta_{\text {in }} \times p\) & NOP & 1 \\
\hline 11 & \(\theta_{\text {oul }}=\theta_{\text {in }} ; r=r-\theta_{\text {in }} \times \mathrm{w}\) & NOP & 1 \\
\hline 12 & \(\mathrm{z}=\mathrm{r} \times \mathrm{m}\) & NOP & 1 \\
\hline 13 & \(\theta_{\text {out }}=\theta_{\text {in }} ; p=z+\theta_{\text {in }} \times p\) & NOP & 1 \\
\hline
\end{tabular}

\subsection*{10.6 Summary}

In this chapter, we have proposed a linear systolic array for CG method. The original CG algorithm is analysed and modified. By comparing the critical length of two versions of CG algorithms, we find one of them is more suitable for parallel implementation as it has a shorter critical length. The DG of the CG algorithm is then designed based on the single assignment algorithm. Since the algorithm is complex and its dataflow is very irregular, we have transformed the DG into a more regular one suitable for systolic array mapping. The systolic array design method based on signal flow graph
(SFG) representation has been adopted to map the DG into an SFG and then a systolic array. The obtained systolic array employs \((n+1)\) processing elements. It can complete each CG iteration in \(3 n+5\) time steps. Adjacent iterations can be fully pipelined. The array can complete a CG algorithm in \(K(3 n+5)\) time steps. The array has a constant efficiency of about \(33 \%\) regardless the size of the problem. In addition, we have also designed a systolic array for the preconditioned conjugate gradient method by choosing a simple diagonal preconditioner. It is noticed that further work on the simulation of the algorithm is required in order to further verify the proposed systolic algorithm.

\section*{Chapter 11}

\section*{SUMMARY AND FURTHER RESEARCH}

\subsection*{11.1 Systolic Array Processing}

S
INCE the development of the first all-purpose, all electronic digital computer, the digital computer has evolved at extremely rapid pace. The computer performance has been improved tremendously in the last 50 years. However, engineers, scientists and researchers are still making continuous efforts to further improve the performance and reduce the cost of computer systems.

There are basically two ways to improve the computer performance in terms of computational speed. One way is to use faster devices (VLSI chips). Although faster and faster VLSI components have contributed a great deal on the improvement of computation speed, the breakthroughs in increasing switching speed and circuit densities of VLSI devices will be difficult and costly in future. The other way is to use parallel processing architectures which employ multiple processors to perform a computation task. When multiple processors are working together, an appropriate architecture is very important to achieve the maximum performance in a cost-effective manner. Systolic arrays are ideally qualified for computationally intensive applications with inherent massive parallelism because they capitalize on regular, modular, rhythmic, synchronous, concurrent processes that require intensive, repetitive computation.

The systolic array architecture in parallel processing came as a product of a certain environment, the means and the background knowledge for its realisation. The needs can be outlined as the everincreasing tendency for faster computations, especially in areas like real-time signal processing and large-scale scientific computation. The means were provided by the remarkable advances in VLSI technology and computer aided design (CAD). Finally the background includes the applications of parallel processing and the design of parallel computers of various architectures including pipelined vector processors, SIMD architectures, array processors, MIMD architectures and so on.

\subsection*{11.2 Summary of the Thesis}

This thesis can be divided into three parts. The first part is an introductory part containing Chap. 1 and Chap. 2. The second part concerns with the systolic design methodology. The third part deals with the several systolic array design for different applications.

\subsection*{11.2.1 Overview of Computer Architecture and Systolic Array Processing}

Computer technology has undergone a rapid evolution in the last fifty years. The performance of computer in terms of computational capability has been improved a very great deal. This owes much to the following two factors: (a) VLSI technology, and (b) computer architecture. With the advent of VLSI, the size and cost of processing logic, memory and communication hardware was dramatically reduced. The VLSI technology enables us to integrate into a single chip millions of parts or devices. High density and low cost VLSI chips allow us to construct high performance computers and also make multiprocessing practical. When many processing elements are put together to achieve higher performance, architecture, which defines how different processing elements cooperate and how data communicate, plays a very important role in the parallel computer system design. Many computer architectures have been developed such as coprocessors, pipelined vector processors, SIMD architectures, array processors, MIMD architectures and so on.

Systolic arrays combine features from many other parallel architectures in a massively parallel architecture. A systolic array can act as a coprocessor, can contain multiple processors, and can act as a pipeline (maybe \(n\)-dimensional) processor. It is the advance of the VLSI technology and the parallel processing which has brought about the introduction of the systolic array. To design a complicated computer system with multiprocessors using VLSI technology, the design cost and the data communication between different processing elements are major concerns. Systolic arrays featuring regularity, modularity, and local communication alleviate these design constraints. By regularity and modularity, systolic array architectures are based on a small number of basic component cells and it is an advantage over an architecture that is based on a large number of complex cells. The regular local communication considerably eases the arrangement of component in a VLSI chip design. Thus systolic architectures are best amenable to VLSI implementation to achieve massive parallelism with minimum communication overhead.

Ever since H.T. Kung [KungHT78] proposed the systolic architecture, its elegant solution to demanding problems and its potential performance have attracted great attention. Systolic arrays have found applications in matrix arithmetic, signal processing, image processing, solution of differential equations, data structure, graph algorithms, computer aided design (CAD), and so forth.

Many systolic algorithms have been designed for a great diversity of areas. Few problems resist attack from systolic arrays. A lot of efforts have been made to implement the systolic array in a VLSI chip, in a programmable processor array, or in a reconfigurable systolic array. While systolic arrays were originally used for fixed or special purpose architecture, the systolic array concept has been extended to general-purpose SIMD and MIMD architectures.

\subsection*{11.2.2 Systolic Array Design Methodologies}

One of the important design problems in systolic array processing is the development of systematic methodology for transforming an algorithm represented in some high-level constructs into a systolic architecture specified by the timing of data movement and the interconnection of processing elements such that the design requirements can be satisfied.

Among a number of systolic design methodologies, we have studied two types of systematic design methodologies. The first method starts from the RIA (regular iterative algorithm) representations of a given algorithm, extracts the data dependency from the RIA algorithm, and then map the algorithm into a systolic array by using algebraic approach. This method is applied to the matrix-matrix multiplication problem. We have obtained 19 different systolic designs for the matrix multiplication problem. A detailed performance analysis for all the 19 designs is presented. Based on the performance measures, we can choose the right design for given requirements.

The other design method we have studied uses the SFG (signal flow graph) of a given algorithm to design systolic arrays through graphic mapping and retiming. An algorithm is first represented by a DG (dependence graph). Then the DG is mapped into an SFG by a graph projection. Cut-set retiming procedure is then applied to the obtained SFG to derive a regular and temporally localized SFG, from which a systolic array design is immediately available. A number of systolic arrays for matrix triangularisation, including both LU decomposition and QR decomposition, have been designed using this method.

\subsection*{11.2.3 Systolic Arrays Design and Applications}

A number of systolic arrays have been designed in this thesis. Firstly, we designed a systolic array for linear and inverse matrix system. The systolic array combines LU (or QR) decomposition, backward and forward substitution, and matrix multiplication into a trapezoidal array. The array can be used to solve a variety of linear systems and other problems such as matrix multiplication. When QR decomposition is used, the systolic array can solve linear least squares problems or perform pseudoinverse computation. Then a systolic array is designed to compute the Schur complement which often occurs in the domain decomposition method and other problems in linear algebra such as the numerical solution of elliptic partial differential equations and sparse linear system. A systolic array architecture has been designed and simulated for QR decomposition of block structured sparse systems which find applications in areas such as photogrammetry, Doppler radar positioning, geodetic problems, cluster analysis and pattern matching. A novel systolic array is derived for computing matrix inversion by Newton Iteration which has an attractive feature that some simper initial approximations can always guarantee a convergence. A systolic array architecture for Capon's DOA estimation in sonar/radar sensor array processing is derived based on the QR decomposition technique. Finally a novel linear systolic array architecture is proposed for conjugate gradient (CG) algorithm.

\subsection*{11.3 Further Research}

CAD Tools for Systolic Array Design. One of the major applications for systolic arrays is the design of the application-specific imtegrated circuits (ASICs), particularly for signal and image processing. It is important to find methods that allow us rapidly to explore different possible solutions to a single problem, and to compare them according to different performance measurements. Chap. 3 and Chap. 4 introduce two popular methods for systematic systolic array synthesising. It is desirable to have Computer-Aided Design (CAD) tools to automatize the design procedures. Some software tools have been developed, for example, VACS (a VLSI Array Compiler System) [KungSY88c] [KungSY89], ADVIS [Moldovan87]. Systolic synthesis tools are still at early stages of development and a fully integrated system is still some way off [Megson92]. Continued development of CAD tools is a worthwhile and necessary task. Sophisticated CAD tools will be beneficial to a low design cost for systolic arrays, and hence will make it more feasible to use ASICs in those demanding applications.

Systolic Algorithms Design. Although CAD tools will continue to improve in sophistication, they are very difficult to solve all the systolic algorithms design problems. Firstly, many complex applications require complicated designs that may be not feasible to automatize by a CAD tools. Secondly, CAD tools only aid the design process and they cannot innovate or produce novel designs automatically. Therefore, systolic algorithms design is still an active research area.

Mapping Systolic Algorithms to General Purpose Parallel Architectures. As design and fabrication cost of systolic chips is still relatively high, many systolic algorithms are not feasible to implement in a special-purpose hardware. It becomes attractive to map and implement systolic algorithms efficiently in existing general purpose parallel architectures. Similar to vectorization in pipeline computers, systolization of algorithms may result an efficient implementation in the existing systems.

Programmable and Reconfigurable Systolic Array. The more specialized the hardware, the higher the performance; but cost per application also rise and flexibility decrease. Therefore general purpose systolic array architectures are very attractive in certain applications. Programmable and/or reconfigurable systolic arrays have received much attention recently.

Systolic array architecture is an formidable approach to exploiting massive parallelism with a minimum communication overhead. Featuring modularity, regularity and local communication, systolic arrays are amenable to VLSI implementation. VLSl technology advances, parallel processing and demanding scientific and engineering applications have made the systolic array architecture a leading approach for handling computationally intensive applications. Systolic arrays hold great promise to be a pervasive form of massive parallel processing.

\section*{REFERENCES}
[Ahmed82] H. Ahmed, J.M. Delosme, M. Morf, Highly Concurrent Computing Structures for Matrix and Signal Processing, Computer 1982, No. 1, pp 65-82
[Alia84] G. Alia, F. Barsi and E. Martinelli, A Fast Near Optimal VLSI Implementation of FFT Using Residue Number Systems, Integration, Vol. 2, 1984, pp 133-147
[Annarat87] M. Annaratone et al, Architecture of Warp, Proc. Compcon, IEEE CS Press, Los Alamitos, Calif., 1987, pp 264-267
[Anton87] H. Anton and C. Rorres, Elementary Linear Algebra with Applications, Anton Textbooks, Inc., 1987
[Apostolico84] A. Apostolico, A. Negro, Systolic Algorithms for String Manipulations, IEEE Trans. Comput., Col. 33, No. 4, 1984, pp 361-364
[Atkinson78] K.E. Atkinson, Introduction to Numerical Analysis, Wiley, 1978
[Baggeroer78] A.B. Baggeroer, Sonar Signal Processing, in Applications of Digital Signal Processing, Ed. A.V. Oppenheim, Prentice Hall, 1978
[Bartee72] Thomas C. Bartee, digital Computer Fundamentals, McGraw-Hill Kogakusha, Ltd., 1972, pp 1-18
[Baudet83] G.M. Baudet, F.P. Preparata, Area-Time Optimal VLSI Circuits for Convolution, IEEE Trans. Comput., vol. 32, No. 7, 1983, pp 684-688
[Bekakos80] M.P. Bekakos and D.J. Evans, The Exposure and Exploitation of Parallelism on Fifth Generation Computer Systems, In: Parallel Computing, M. Feilmeier et al. (eds), North-Holland, 1980, pp 425-442
[Bentley79] J.L. Bentley and H.T. Kung, A Tree Machine for Searching Problems, Proc. Int'I Conf. Parallel Processing, Aug. 1979, pp 257-266
[Björck90] \(\AA\). Björck, Least Squares Methods, in: Handbook of Numerical analysis, edited by P.G. Ciarlet and J.L. Lions, Elsevier Science Publishers B.V. (North-Holland), 1990
[Borkar88] S. Borkar et al., iWarp: An Integrated Solution to High-Speed Parallel Computing, Proc. Supercomputing, IEEE CS Press, Los Alamitos, Calif., 1988, pp 330-339
[Breant81] R.P. Brent and H.T. Kung, Systolic VLSI Arrays for Polynomial GCD Computation, IEEE Trans. Computer, Vol. 30, No. 8, 1981, pp 731-736
[Brent83b] R.P. Brent, F.T. Luk and C. Van Loan, Computation of the Generalized Singular Value Decomposition Using Mesh-Connected Processors, SPIE Vol. 431, Real Time Signal Processing VI, 1983, pp 66-71
[Brent83] R.P. Brent, H.T. Kung and F.T. Luk, Some Linear-Time Algorithms for Systolic Arrays, Information Processing, Vol. 83, 1983, pp 863-876
[Brent84] R.P. Brent, F.T. Luk, The solution of Singular Value Problems Using Systolic Arrays, SPIE, Vol. 495, Real Time Signal Processing VII, 1984, pp 7-12
[Brent85] R.P. Brent and F.T. Luk, Singular-Value and Symmetric Eigenvalue Problems, SIAM J. Sci. Stat. Comput., Vol. 6, No. 1, January 1985, pp 69-84
[Burger84] Robert Burger, R.K. Cavin III, William C. Holton and L.W. Sumney, The Impact of ICs on Computer Technology, IEEE Computer, Vol. 17. October 1984, pp 88-95
[Burleson88] W.P. Burleson and L.L. Scharf, A VLSI Implementation of a Cellular Rotator Array, Proceedings of IEEE 1988 Custom Integrated Circuits Conference, 1988, pp 8.1.18.1 .4
[Capon69] J. Capon, High Resolution Frequency-Wavenumber Spectrum Analysis, Proc. IEEE, Vol. 57, 1969, pp 1408-1418
[Cappello83] P.R. Cappello and K. Steiglitz, Unifying VLSI Array Designs with Geometric Transformations, In International Conference Parallel Processing, 1983, pp 448-457
[Cappello84] P.R. Cappello and K. Steiglitz, Unifying VLSI Array Design with Linear Transformations of Space-Time, In Advances in Computing Research, Vol. 2, JAI Press Inc., 1984, pp 23-65
[Cappello86] P. Cappello, G. Davisdon, A. Gersho, C. Koc and V. Somayazulu, A Systolic Vector Quantization Processor for Real Time Speech Coding, IEEE ICASSP, Tokyo, 1986, pp 2143-2146
[Chang90] L.-W Chang, J.-H Lin, Bit Level Systolic Arrays for Real Time Median Filters, Proceedings of IEEE ICASSP'90, Vol. 3, 1990, pp 1791-1794
[Chang91] L.-W. Chang, M.-C. Wu, A Unified Systolic Array for Discrete Cosine and Sine Transforms, IEEE Transactions on Signal Processing, Vol. 39. No. I, Jan 1991, pp 192-194
[Chapman85] R. Chapman, T.S. Durrani, and T. Willey, Design Strategies for Implementing Systolic and Wavefront Arrays Using Occam. In International Conference on ASSP, March 1985, pp 292-295
[Charot86] F. Charot, P. Frison and P. Quinton, Systolic Architectures for Connected Speech Recognition, IEEE Transactions on ASSP, Vol 34, 1986, pp 765-779
[Chazelle84] B. Chazelle, Computational Geometry on a Systolic Chip, IEEE Transactions on Computers, Vol. 33, No. 9, 1984, pp 774-785
[Chen86] M. Chen, Synthesizing VLSI Architectures: Dynamic Programming Solver, In International Conference on Parallel Processing, 1986, pp 776-784
[Cheng85] H.D. Cheng, Algorism Partition of General Context-Free Languages Using FixedSize VLSI Architecture, Proc. First International Conference on Supercomputing, 1985, pp 631-640
[Chuang85] H.Y.H. Chuang and C.C. Li, A Systolic Array Processor for Straight Line Detection by Modified Hough Transform, Proc. of CAPAIDM'85, 1985, pp 300-304
[Cok91] Ronald S. Cok, Parallel Programs for the Transputer, Printice-Hall Inc, 1991
[Comon87] P. Comon and Y. Robert, A Systolic Array for Computing BA \({ }^{-1}\), IEEE Transactions on Acoustics, Speech and Signal Processing, ASSP-35, No. 6, June 1987, pp 717723
[Cong90] D. Cong, S. Haykin, A Multi-Layer Neural Network Classifier for Radar Clutter, IEEE Int. Jt. Conf. Neural Networks IJCNN'90, pp 241-246
[Corry83] A. Corry and K. Patel, Architecture of a CMOS Correlator, In Proc. IEEE Int. Symp. on Circuits and Systems, 1983, pp 522-525
[Cosnard88] M. Cosnard, M.H. Barton and M. Vanneschi, ed., Parallel Processing, Elsevir, North Holland, 1988
[Dahlquist74] G. Dahlquist, A. Bjork, Numerical Methods, Pretince Hall, 1974
[Das90] S.R. Das, N.H. Vaidiy, L.M. Patnaik, P.C. Mathias, Systolic Algorithm for Hidden Surface Removal, Parallel Computing, Vol. 15, 1990, pp 277-289
[Davis84] R.H. Davis and D. Thomas, Systolic Array Chip Matches the Pace of High-Speed Processing, Electronic Design, October 1984, pp 207-218
[Delosme86b] J.M. Delosme and I.C.F. Ispen, Design Methodology for Systolic Arrays, In SPIE Advanced Algorithms and Architectures for Signal Processing, 1986, pp 245-259
[Delosme86] J.M. Delosme and I.C.F. Ispen, Efficient Systolic Arrays for the Solution of Toeplitz Systems: An illustration of a Methodology for the Construction of Systolic Architectures in VLSI, International Workshop on Systolic Arrays, University of Oxford, July 1986
[Duncan90] Ralph Duncan, A Survey of Parallel Computer Architectures, IEEE Computer, Vol. 23, February 1990, pp 5-16
[Engstrom87] B.R. Engstrom and P.R. Cappeilo, The SDEF Systolic Programming System, In S.K. Tewksbury, B.W. Dickinson and S.C. Schwartz, eds, Concurrent Computations, Plenum Press, New York, 1987, pp 263-301
[Ericson90] M.N. Ericson, C.L. Britton Jr., and D.W. Bouldin, A Virtual Zero-Time, Monolithic Systolic Sorting Array, IEEE SOUTHEASTCON, Vol. 2, 1990, pp 549-552
[Evans82] D.J. Evans, ed., Parallel Processing Systems, Cambridge University Press, 1982
[Evans83] D.J. Evans, R.C. Dunbar, The Parallel Solution of Triangular Systems of Equations, IEEE Trans. Comput., Vol. 32, No. 2, 1983
[Evans86b] D.J. Evans, G.M. Megson, Romberg Integration Using Systolic Arrays, Parallel Computing, Vol. 3, No. 4, 1986, pp 289-304
[Evans86] D.J. Evans, Parallel Processing: Its Uses in All Levels of the Processing Operation, Data Processing, Vol. 28, No. 10, Dec. 1986, pp 529-542
[Evans87] D.J. Evans and G.M. Megson, Construction of Extrapolation Tables by Systolic Arrays for Solving Ordinary Differential Equations, Parallel Computing, Vol. 4, 1987, pp 33-48
[Evans89] D.J. Evans and K. Margaritis, Systolic Designs for Eigenvalue-Eigenvector Computations Using Matrix Powers, Parallel Computing, Vol. 14, 1989, pp 77-87
[Evans91b] . D.J. Evans and G.M. Megson, Matrix Inversion by Systolic Rank Annihilation, in [Evans91]
[Evans91c] D.J. Evans and K. Margaritis, Systolic Computation of the Matrix Exponential and Other Matric Functions, in [Evans91]
[Evans91] D.J. Evans, Systolic Algorithms, Gordon and Breach Science Publishers, 1991
[Evans92] D.J. Evans, Edit., Preconditoned Iterative Methods, Intern. J.Comp. Maths., Vol 44, 1992, pp 1-364
[Evans93a] D.J. Evans and C.R. Wan, Parallel Direct Solution for P-Cyclic Computation, Intern. J. Computer. Math., Vol. 48, 1993, pp 105-116
[Evans93b] D.J. Evans and C.R. Wan, Systolic Array for Schur Complement Computation, Intern. J. Computer Math., Vol. 48, 1993, pp 105-116
[Evans93c] D.J. Evans and C.R. Wan, Preconditioned Conjugate Gradient Method for \(\mathrm{AX}+\mathrm{BX}=\mathrm{C}\), Intern. J. Computer Maths., Vol. 49, No. 3-4, 1993, pp 207-219
[Fang91] Wai-Chi Fang. Chi-Yung Chang and Bing J. Sheu, Systolic Tree-Searched Vector Quantizer for Real-Time Image Compression, in VLSI Signal Processing IV, ed. H.S. Moscovits, Kung Yao, R. Jain,IEEE Press, 1991, pp 353-361
[Finn82] A.M. Finn, F.T. Luk, and C. Pottle, Systolic Array Computation of the Singular Value Decomposition, Proc. SPIE, Vol. 34I, 1982, pp 74-83.
[Fisher82] A.L. Fisher, Systolic Algorithms for Running Order Statistics in Signal and Image Processing, Journal of Digital Systems, Vol. 4, 1982, pp 251-264
[Fisher84] A.L. Fisher, K. Sarocky and H.T. Kung, Experience with CMU Programmable Systolic Chip, Proc. SPIE Real Time Signal Processing VII, SPIE, 1984, pp 495
[Flynn66] M.J. Flym, Very high speed computing systems, Proceedings of IEEE, Vol. 54, 1966, pp 1901-1909
[Fortes85] J.A.B. Fortes and D.I. Moldovan, Parallelism Detection and Transformation Techniques Useful for VLSI Algorithms, Journal of Parallel and Distributed Computing, Vol. 2, 1985, pp 277-301
[Fortes88] J.A.B. Fortes, King-Sun Fu, and B.W. Wah, Systematic Design Approaches for Algorithmically Specified Systolic Arrays, in Veljko Milutinovic, ed., Computer Architectures, North-Holland, 1988, pp 454-494
[Foster80] M.J. Foster and H.T. Kung, The Design of Special-Purpose VLSI Chips, IEEE Computer Vol. 13, No. 1, 1980, pp 26-40
[Foulser87] D. Foulser and R. Scheiber, The Saxpy Matrix-1: A General-Purpose Systolic Computer, Computer, Vol. 20, No. 7. July 1987, pp 35-43
[Friedland87] B. Friedlander, Block Processing on a Programmable Systolic Array, Proc. Int'l Conf. Parallel Processing, IEEE CS Press, Los Alamitos, Calif., 1987, pp. 184-187
[Frison89] P. Frison et al., Micmacs: A VLSI Programmable Systolic Architecture, Systolic Array Processors, Prentice-Hall, Englewood Cliffs, N.J: 1989, pp 145-155
[Frumkin92] M.A. Frumkin, Systolic Comiputations, Kluwer Academic Publishers, 1992
[Fu85] K.S. Fu, K. Hwang, B.W. Wah, VLSI Multiprocessor for Image Processing, NATO ASI Ser. F, No. 18, 1985, pp 139-157
[Gachet86] P. Gachet, B. Joinnault, and P. Quinton, Synthesizing Systolic Arrays Using DIASTOL, In W. Moore, A. McCabe and R.R. Urauhart, eds., Systolic Array, Adam Hilger, Bristol, 1986, pp 25-36
[Gannon82] D. Gannon, Pipelining Array Computations for MIMD Parallelism: A functional specification, in International Conference on Parallel Processing, August 1982, pp 284-286
[Gentleman73] W.M. Gentleman, Least squares computation by Givens transformations without square roots, J. Inst. Maths Applics, 1973, 12, pp 329-336
[Gentleman81] W.M. Gentleman and H.T. Kung, Matrix Triangularization by Systolic Arrays, Proc. SPIE, Vol. 248, 1981, pp 19-26
[Gertner87] I. Gertner and M. Shamash, VLSI Architectures for Multidimensional Fourier Transform Processing, IEEE Trans. Comput., Vol. 36, No. 11, 1986, pp 1265-1274
[Gharachor85] N. Gharachorloo, Super Buffer: A Systolic VLSI Graphics Engine for Real Time Raster Image Generation, Ph.D. Thesis, Electrical Engineering Department, Cornell Univ., Ithaca, NY, August 1985
[Gharachor88] N. Gharachorloo, S. Gupta, E. Hokenek, A Million Transistor Systolic Array Graphics Engine, Proceedings of the Intern. Conf. on Systolic Arrays, 1988, pp 193202
[Gill81] P.E. Gill, W. Murray and M.H. Wright, Practical Optimization, Academic Press, 1981
[Gokhale90] M. Gokhale et al., Splash: A Reconfigurable Linear Systolic Array, Proc. Int'l conf. Parallel Processing, IEEE CS Press, Los Alamitos, Calif. 1990, pp 1526-1531
[Golub79] G.H. Golub, F.T. Luk and M. Pagano, A Sparse Least Squares Problem in Photogrammetry, in: Proceedings Computer Science and Statics: Twelfth Annual Conference on the Interface, Waterloo, Ont. , 1979
[Golub80] G.H. Golub, R.J. Plemmons, Large Scale Geodetic Least Squares Adjustment by Dissection and Orthogonal Decomposition, Linear Algebra Appl. Vol 34, pp 3-27, 1980
[Golub89] G.H. Golub and Charles F. Van Loan, Matrix Computations, 2nd Edition, The Johns Hopkins University Press, 1989
[Guibas79] L.J. Guibas, H.T. Kung and C.D. Thompson, Direct VLSI Implementation of Combinatorial Algorithms, Proc. Conf. Very Large Scale Integration: Architecture, Design, Fabrication, California Institute of Technology, Jan. 1979, pp 509-525
[Guibas82] L.J. Guibas and F.M. Liang, Systolic Stacks, Queues, and Counters, Proc. Conf. Advanced Research in VLSI, MIT, Jan. 1982
[Gunzinger92] A. Gunzinger, et. al., Achieving Super Computer Performance with a DSP Array Processor, Proceedings of Supercomputing, 1992, pp 543-550
[Gupta83] Amar Gupta and Hoo-min Toong, ed., Advanced Microprocessors, IEEE Press, 1983
[Gupta87] Amar Gupta, ed., Advance Microprocessors, II, IEEE Press, 1987
[Gwinn89] R.P. Gwinn, P.B. Norton, P.W. Goetz, The New Encyclopædia Britannica, 15th edition, Encyclopædia Britannica, Inc., Vol 16, 1989, pp 629-652
[Hageman81] L. Hageman, D. Young, Applied Iterative Methods, Academic Press, 1981
[Halem89] Milton Halem, Scientific Computing Challenges Arising from Space-Borne Observations, Proceedings of the IEEE, Vol. 77, No. 7, July 1989, pp 1061-1091
[Hargrave86] P.J. Hargrave, C.R. Ward and J. McWhirter, A Novel Algorithm and Architecture for Adaptive Digital Beamforming, IEEE Transactions on Antennas and Propagation, March 1986, pp 338-346
[Haykin92] S. Haykin, J.P. Reilly and E. Vertatschitsch, Some Aspects of Array Signal Processing, IEE Proc. F, Vol 139, Feb 1992
[Hayt78] W.H. Hayt and J.H. Kemmerly, Engineering Circuits Analysis, McGraw-Hill, New York, 1978
[Heller83] D.E. Heller and I.C.F Ipsen, Systolic Networks for Orthogonal Decompositions with Applications, SIAM J. Sci. Stat. Comput., V0I. 4, 1983, pp 261-269
[Hestenes52] M.R. Hestnenes and E. Stiefel, Methods of Conjugate Gradients for Solving Linear Systems, J. Res. Nat. Bur. Stand., Vol 49, 1952, pp 409-436
[Hoppe86] H.C. Hoppe and H. Muhlenbein, Parallel Adaptive Full Multigrid Methods on Message-Based Multiprocessors, Parallel Computing, Vol. 3, 1986, pp 269-287
[HPCR93] Supercomputers Break New Ground, High-Performance Computing Review, July/August 1993, pp 4-12
[Hurson86] A.R. Hurson, B. Shirazi, A Class of Systolic Multiplier Units for VLSI Technology, Intern. J. Comput. and Inform. Sci., Vol. 14, No. 5, 1986, pp 261-276
[Hurst92] S.L. Hurst, Custom VLSI Microelectronics, Prentice Hall, 1992
[Hwang84] Kai Hwang and F.A. Briggs, Computer Architecture and Parallel Processing, McGraw-Hill, Inc., 1984
[Hwang91] J. -N. Hwang, and J. A. Ritcey, Systolic Architecture for Radar CFAR Detectors, IEEE Transactions on Signal Processing, Vol. 39, No. 10. October 1991, pp 22862295
[IEEE89] \(\longrightarrow\), Supercomputer Hardware, IEEE Computer, Vol. 22, Iss. II, pp 63-8
[INMOS88a] INMOS Limited, Transputer Refenece Manual, Hemel Hempstead, U.K., Prentice Hall International (U.K.) Ltd, 1988
[INMOS88b] INMOS Limited, Occam 2 Refenece Manual, Hemel Hempstead, U.K., Prentice Hall International (U.K.) Ltd, 1988
[Ipsen84] I. Ipsen, Singular Value Decomposition with Systolic Arrays, SPIE Vol. 495, Real Time Signal Processing VII, 1984, pp 13-21
[Ishii88] M. Ishii, H. Sato, K. Murakami, M. Ikesaka and H. Ishihata, Cellular Array Processor CAP and Application, Proc. Int'l Conf. Systolic Arrays, IEEE CS Press, Los Alamitos, Calif., 1988, pp 535-544
[Jagadish87] H.V. Jagadish, S.K. Rao and T. Kailath, Array Architectures for Iterative Algorithms, Proceedings of the IEEE, Vol. 75, No. 9, September 1987, pp 1304-1321
[Joanblanq90] C. Joanblanq, F. Rothan and P. Senn, A 54 MHz Chip Set for HDTV Filtering, Proceedings of the IEEE 1990 Custom Integration Circuits Conference, 1990, pp 25.3.1-25.3.4
[Johnson93] K.T. Johnson and A.R. Hurson, General-Purpose Systolic Arrays, IEEE Computer, November 1993, pp 20-31
[Jones89] S. Jones, A. Spray and A. Ling, A Flexible Building Block for the Construction of Processor Arrays, in Systolic Array Processors, Prentice-Hall, Englewood Cliffs, N.J., 1989, pp 459-466
[Jordan79] H.F. Jordan and P.L. Sawyer, A Multiprocessor System for Finite Element Structural Analysis, Computers and Structures, Vol. 10, 1979, pp. 118-135
[Kalson85] S. Kalson and K. Yao, A Systolic Array fr Linearly Constrained Least Squares Filtering, In Proc. IEEE ICASSP'85, 1985, pp 977-980
[Kane87a] R. Kane and S. Sahni, Hardware Algorithm for Net Extraction, Computer-Aided Design, Vol. 19. No. 7, 1987. pp 347-354
[Kane87b] R. Kane and S. Sahni, A Systolic Algorithm for Rectilinear Polygons, ComputerAided Design, Vol. 19, No. I, 1987, pp 101-109
[Kane87c] R. Kane and S. Salmi, A Systolic Rule Checker, IEEE Trans. CAD, Vol. 6, No. 2, 1987, pp 22-23
[Kean89] T. Kean and J. Gray, Configurable Hardware: Two Case Studies of Micrograin Computation, in Systolic Array Processors, Prentice-Hall, Englewood Cliffs, N.J., 1989, pp 310-319
[Keyes79] R.W. Keyes, The Evolution of digital electronics towards VLSI, IEEE Transactions on Electron Devices, Vol 26, No. 4, 1979, pp 271-278
[Kittler86] J. Kittler, M.J.B. Duff, Eds, Image Processing System Architectures, New York, Wiley, 1986
[Kohn89] Les Kohn and Neal Margulis, Introducing the Intel i860 64-Bit Microprocessor, IEEE Micro, Vol. 9, Iss. 4, 1989, pp 15-30
[Koren83] I. Koren and G.M. Silberman, A Direct Mapping of Algorithms onto VLSI Processing Arrays Based on the Data Flow Approach, In International Conference on Parallel Processing, 1983, pp 335-337
[KungHT78] H.T. Kung, C.E. Leiserson, Algorithms for VLSI Processor Arrays, Sparse Matrix Proceedings, SIAM Press, 1978, pp 256-282
[KungHT79] H.T. Kung and C.E. Leiserson, Systolic Arrays (for VLSI), Sparse Matrix Proc. 1978, SIAM, 1979, pp 256-282
[KungHT80a] H.T. Kung, The structure of parallel algorithms, Advances in Computer, Vol. 19, 1980, pp 66-112
[KungHT80b] H.T. Kung, Special-Purpose Devices for Signal and Image Processing: An Opportunity in VLSI, Proc. SPIE Vol. 241, Real Time Signal Processing III, July 1980, pp 76-84
[KungHT80c] H.T. Kung and C.E. Leiserson, Systolic Array for VLSI, in [Mead80]
[KungHT81] H.T. Kung, L.M. Ruane and D.W.L. Yen, A Two Level Pipelined Systolic Arrays for Convolutions, In VLSI Systems and Computations, H.T. Kung, R.F. Sproull and G.L. Steele, Jr. (Eds.), Carnegie-Mellon University, Computer Science Press, Oct. 1981, pp 255-264
[KungHT82a] H.T. Kung, Why Systolic Architecture? Computer, Vol. 15, No. I, Jan. 1982, pp \(37-\) 45
[KungHT82b] H.T. Kung, Notes on VLSI Computations, in [Evans82], pp 339-356
[KungHT83] H.T. Kung and W.T. Lin, An Algebraic for Systolic Computation, Proc. Conference on Elliptic Problem Solvers, 1983, pp 141-160
[KungHT85] H.T. Kung, Two-Level Pipelined Systolic Arrays for Matrix Multiplication, Polynomial Evaluation and Discrete Fourier Transformation, In: Dynamical Systems and Cellular Automata, Ed. J. Demongeof et al., Academic Press, 1985, pp 321-330
[KungSY82b] S.Y. Kung and R.J. Gal-Ezer, Synchronous Versus Asynchronous Computation in Very Large Scale Integrated (VLSI) Array Processors, SPIE Vol. 341, Real Time Signal Processing V, 1982, pp 53-65
[KungSY82] S.Y. Kung, K.S. Arun, R.J. Gal-Ezer, and D.V. Bhaskar Rao, Wavefront Array Processor: Language, Architecture, and Application, IEEE Trans. Comput., Vol. 31, No. 11, November 1982, pp 1054-1066
[KungSY83] S.Y. Kung and Y.H. Hu, A Highly Concurrent Algorithm and Pipelined Architecture for Solving Toeplitz Systems, IEEE Transactions on ASSP. Vol. 31, No. 1, 1983, pp 66-76
[KungSY84] S.Y. Kung, On Supercomputing with Systolic/Wavefront Array Processors, Proceedings of the IEEE, Vol. 72, No. 7, July 1984, pp 867-884
[KungSY85a] S.Y. Kung, H.J. Whitehouse and T. Kailath, eds, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1985
[KungSY85b] S.Y. Kung, VLSI Signal Processing: From Transversal Filtering to Concurrent Array Processing, in [KungSY85a], pp 121-152
[KungSY88a] S.Y. Kung, VLSI Array Processors: Designs and Applications, 1988 IEEE International Symposium on Circuits and Systems Proceedings, Vol. 1, 1988, pp 313320
[KungSY88b] S.Y. Kung, VLSI Array Processors, Prentice Hall, 1988
[KungSY88c] S.Y. Kung and S.N. Jean, A VLSI Array Compiler System (VACS) for Array Design, In. VLSI Signal Processing, III, Nov. 1988, pp 495-508
[KungSY89] S.Y. Kung and S.N. Jean, Array Compiler Design for VLSI/WSI Systems, in International Conference on Systolic Arrays, May 1989
[KungSY91] S.Y. Kung and J.N. Hwang, Systolic Array Designs for Kalman Filtering, IEEE Transactions on Signal Processing, Vol. 39. No. I.Jan 1991, pp 171-182
[Lam85] M.S. Lam and J. Mostow, A Transformational Model of VLSI Systolic Design, IEEE Computer, Vol. 27, No. 2, February 1985, pp 42-52
[Lam91] S.P.S. Lam, A Systolic Implementation of the Jacobi Algorithm, Proc. of IEEE ICASSP 1991, Vol.2, pp 1021-1024
[Lang86] H.W. Lang, The Instruction Systolic Array: a Parallel Architecture for VLSI, Integration-the VLSI journal, Vol. 4, 1986, pp 65-74
[Law85] K.H. Law, Systolic Arrays for Finite Element Analysis, Comput. and Struct., Vol. 20, No. 1/3, 1985, pp 55-65
[Lee88] P. Lee and Z.M. Kedem, Synthesizing Linear Array Algorithms from Nested For Loop Algorithms, IEEE Transactions on Computers, Vol. 37, No. 12, December 1988, pp 1678-1598
[Leghar89] F. Leghart, Supercomputer Technology, Computer Technology Research Corp., 1989, pp 144-152
[Leighton92] F.T. Leighton, Parallel Algorithms and Architectures, Morgan Kanfmann, 1992
[Leiserson81] C.E. Leiserson, Area-efficient VLSI computation, Ph.D. Thesis, Carnegie-Mellon University, Computer Science Department, Pittsburgh, PA., October, 1981
[Lewis86] P.S. Lewis and S.Y. Kung, Dependence Graph Based Design of Systolic Arrays for Algebraic Path Problem, In Proc. 12th Annual Asilomar Conf. on Signals, Systems and Computers, November 1986
[Lewis90] P.S. Lewis, Systolic Architectures for Adaptive Multichannel Least Squares Lattice Filters, J. of VLSI Signal Processing, Vol. 2, No. I, 1990, pp 29-36
[Lewis91] P.S. Lewis amd S.Y. Kung, An Optimal Systolic Array for Algebraic Path Problem, IEEE Trans. on Computers, Vol. 40, No. I, Jan. 1991, pp 100-105
[Li83] G.J. Li and B.W. Wah, The Design of Optimal Systolic Arrays, Proc. Computer Software and Application Conference, IEEE, 1983, pp 310-319
[Li85] G.J. Li and B.W. Wah, The Design of Optimal Systolic Arrays, IEEE Transactions on Computers, Vol. 35, 1985, pp 66-77
[Lindscog86] B. Lindscog and P.E. Danielsson, PICAP3: A Parallel Processor Tuned for 3D Image Operations, Proc. Eighth Int'l Conf. Pattern Recognition, IEEE CS Press, 1986, pp 1248-1250
[Luoka88] B. Luoka, M. Tchuente, Dynamic Programming on 2D Systolic Arrays, in [Cosnard88], pp 265-274
[Ma91] W. Ma, 2-D DCT Systolic Array Implementation, Electronics Letters, Vol. 27, No. 3, Jan 1991, pp 201-202
[Malek89] M. Malek and E. Opper, The Cylindrical Banyan Multicomputer: A Reconfigurable Systolic Architecture, Parallel Computing, Vol. 10, No. 3, May 1989, pp 319-326
[Manneback85] P. Manneback, C. Murigande and P.L. Toint, A Modification o an Algorithm by Golub and Plemmons for Large Linear Least Squares in the Context of Doppler Positioning, IMA J. Numer. Anal. Vol 5, 1985, pp 221-234
[Manolakos91] E.S. Manolakos, H.M. Stellakis and D.H. Brooks, Parallel Processing for Biomedical Signal Processing: Higher Order Spectral Analysis-An Application, IEEE Computer, March 1991, pp 33-43
[Margaritis92] K. Margaritis and D.J. Evans, A Systolic Ring Architecture for Solving Polynomial Equations, in [Evans91], pp 55-67
[May92] David May, Roger Shepherd and Peter Thompson, The T9000 Transputer, IEEE 1992 International Conference on Computer Design: VLSI in Computers and Processors, ICCD '92, p209-212
[McCanny82] J.V. McCanny and J.G. McWhirter, Implementation of Signal Processing Functions using 1-bit Systolic Arrays, Electron. Letts., Vol. 18, 1982, pp 241-243
[McCanny84] J.V. McCanny, K.W. Wood, J.G. McWhirter and C.J. Oliver, The Relationship Between Word and Bit Level Systolic Arrays as Applied to Matrix and Matrix Multiplication, In Proc. SPIE, Tech. Symp. Real Time Signal Processing VI, 1983
[McCanny86] J.V. McCanny and J.G. McWhirter, The Derivation and Utilization of Bit Level Systolic Array Architectures, In Proc. of the International Workshop on Systolic Arrays, University of Oxford, July 1986, pp F1.1-FI. 12
[McEvoy91] Kevin McEvoy and Peter M. Dew, The Cut Theorem-A Tool for Design of Systolic Algorithms, in [Evans91], pp 21-5।
[McWhirter83] J.G. McWhirter, Recursive Least Squares Minimization Using a Systolic Array, SPIE, Real Time Signal Processing VI, 1983, pp 105-110
[McWhirter91] J.G. McWhirter, D.S. Broomhead, and T.J. Shepherd, Systolic Array for Nonlinear Adaptive Filtering and Pattern Recognition, Journal of VLSI Signal Processing, Vol. 3, No. 1-2, 1991, pp 69-75
[Mead80] C. Mead and L. Conway, Introduction to VLSI Systems, Addison-Wesley, Reading, Mass., 1980
[Megson90b] G.M. Megson and D.J. Evans, Orthogonal Systolic Design for the Assignment Problem, Parallel Computing, Vol. 16, No. 2-3, Dec. 1990, pp 253-267
[Megson90] G.M. Megson, Efficient Systolic String Matching, Electronics Letters, Vol. 26, No. 24, Nov. 1990, pp 2040-2042
[Megson92] G.M. Megson, An Introduction to Systolic Algorithm Design, Clarendon Press, Oxford, 1992
[Melhem83] R.G. Melhem, An Abstract Systolic Model and Its Application to the Design of Finite Element Methods, PhD Thesis, Pittsburg, 1983
[Miller89] E.K. Miller, The Computer Revolution, IEEE Potentials, May 1989, pp 27-31
[Miranker84] W.L. Miranker and A. Winkler, Space-time representations of computational structures, Computing, Vol. 32, 1984, pp 93-114
[Mitchell90] W.F. Mitchell, Systolic Array for Kalman Filtering With Algorithm-Based Fault Tolerance, SPIE Vol. 1384, Advanced Signal Processing Algorithms, Architectures, and Implementations, pp 450-461
[Modi88] J.J. Modi, Parallel Algorithms and Matrix Computation, Oxford: Claredon Press, 1988
[Moldovan83] D.I. Moldovan, On the Design of Algorithms for VLSI Systolic Arrays, Proceedings of the IEEE, Vol. 71, No. I, January 1983, pp 113-120
[Moldovan86] D.I. Moldovan and J.A.B. Forts, Partitioning and Mapping Algorithms into Fixed Size Systolic Arrays, IEEE Transactions on Computers, Vol. 35, No. I, January 1986, pp 1-12
[Moldovan87] D.I. Moldovan, ADVIS: A Software Package for the Design of Systolic Arrays, IEEE Transactions on Computer Aided Design, Vol. 6, 1987, pp 33-40
[Moonen91b] M. Moonen, P. Van Dooren, J. Vandewalle, Jocobi-Type Algorithms for LDU and Cholesky Factorization, Journal of Parallel and Distributed Computing, Vol. 13, No. 1, 1991, pp 111-117
[Moonen91] M. Moonen, J. Vandewalle, A Systolic Array for Recursive Least Squares Computations, Proc. of IEEE ICASSP 1991, Vol. 2, pp 1013-1016
[Moore75] G.E. Moore, Progress in Digital Integrated Electronics, Int'l Electro Devices Meeting, Dec. 1975, pp. 11-13
[Moreno92] J.H. Moreno and T. Lang, Matrix Computations on Systolic-Type Arrays, Kluwer Academic Publishers, 1992
[Morris86] L. R. Morris, Digital Signal Processing Microprocessors: Forward to the Past? IEEE Micro, Dec. 1986, pp 6-8
[Navarro86] J.J. Navarro, J.M. Llaberia and M. Valero, Computing Size-Independent Matrix Problems on Systolic Array Processors, Comput. Architectures News, 1986, Vol. 14, pp 271-279
[Nicolas85] D. Nicolas, J. Francis, S. Marc and D. Michel, VLSI Architecture for a One Chip Video Media Filter, Proc. IEEE ICASSP'85, Tampa, FL., 1985, pp 26.7.1-26.7.4
[Nielsen91] Richard O. Nielsen, Sonar Signal Processing, Artech House, Inc., 1991
[Nishizawa88] T. Nishizawa, T. Ohgi, K. Nagatomi, H. Kamiyama, K. Maenobu, and M. Fujiwara, A Hidden Surface Processor for 3-Dimensional Graphics, ISSCC, Feb 1988, pp 166167
[Norrie84] Chris Norrie, Supercomputers for Superproblems: An Architectural Introduction, IEEE Computer, Vol. 17, March 1984, pp 62-74
[Nudd85] G.R. Nudd, D.R. Etchells, J. Grinberg, Three Dimensional VLSI Architecture for Image Understanding
[O'Leary87] D.P. O'Leary, Systolic Arrays for Matrix Transpose and Other Reorderings, IEEE Trans. Computers, Vol. 36, No. 12, 1987, pp 117-122
[Oflazer83] K. Oflazer, Design and Implementation of a Single Chip I-D Median Filter, IEEE Transactions on ASSP, Oct. 1983
[Onaga86] K. Onaga, T. Takechi, A Distributed Computation for LU Decomposition of a Partitioned Matrix on VLSI Processor Arrays by Means of Wavefront Control Mechanism, VLSI '85, Amsterdam, Elsevier, North-Holland 1986, pp 197-206
[Pan85] V. Pan and J. Reif, Efficient Parallel Solution of Linear Systems, Proc. 17th Annual ACM Symposium on Theory of Computing, 1985, pp 143-152
[Panchana91] S. Panchanathan, M. Goldberg, A Systolic Array Architecture for Image Coding Using Adaptive Vector Quantization, IEEE Transactions on Circuits and Systems for Video Technology, Vol. 1, No. 2, Jan 1991, pp 222-229
[Peterson89] V. L. Peterson, J. Kim, T. L. Holst, G.S. Deiwert, D. M. Cooper, A. B. Watson and F. R. Bailey, Supercomputer Requirements for Selected Disciplines Important to Aerospace, Proceedings of the IEEE, Vol. 77, No. 7, July 1989, pp 1038-1055
[Peterson91] Craig Perterson, James Sutton and Paul Wiley, iWarp: A 100 MOPS LIW Microprocessor for Multicomputer, IEEE Micro, Vol. II, Iss. 3, 1991, pp 26-29,8187
[Preparata84] F.P. Preparata, VLSI Algorithms and Architectures, In M.P. Chytil and V. Koubex, eds., Lecture Notes in Computer Science 176, Springer-Verlag, 1984
[Przytula88] K.W. Przytula and J.B. Nash, Implementation of Synthetic Aperture Radar Algorithms on a Systolic/Cellular Architecture, Proc. Int'l Conf. Systolic Arrays, IEEE CS Press, Los Alamitos, Calif., 1988, pp 21-27
[Quinton84] P. Quinton, Automatic Synthesis of Systolic Arrays from Uniform Recurrent Equations, In IIth Annual Symposium on Computer Architecture, June 1984, pp 208-214
[Quinton91] P. Quinton and Y. Robert, Systolic Algorithms and Architectures, Masson and Prentice Hall International (UK) Ltd, 1991
[Rajopadhy90] S. Rajopadhye and B. Lisper, Matrix Permutations on Mesh-Connected Arrays, IEEE International Symposium on Circuits and Systems, 1990, Vol. 4, pp 2626-2629
[Ramakrish83] I.V. Ramakrishnan, D.A. Fussell and A. Silberschatz, On Mapping Homogeneous Graphs on a Linear Array Processor Model, In International Conference on Parallel Processing, 1983, pp 440-447
[Randell82] B. Randell and P.C. Treleaven, VLSI Architectures, Prentice Hall, 1982
[Rao84] S.K. Rao and T. Kailath, Orthogonal Digital Filters for VLSI Implementations, IEEE Trans. Circuits and Systems, Vol. 31, No. 11, 1984, pp 933-945
[Ra085] S.K. Rao, Regular Iterative Algorithms and Their Implementation on Processor Arrays, Ph.D. dissertation, Infomation System Lab., Standford University, CA 1985
[Rao88] S.K. Rao and T. Kailath, Regular Iterative Algorithms and Their Implementation on Processor Arrays, Proceedings of the IEEE, Vol. 76, No. 3, March 1988, pp 259-269
[Reilly86] J.P. Reilly, Systolic Array Evaluation of Polynomials with Application to Nonlinear Spectrum Estimation, Electronics Letters, Vol. 22, No. 24, 1986, pp 1300-1301
[Riganati84] John P. Riganati and Paul B. Schneck, Supercomputing, IEEE Computer, Vol. 17, October 1984, pp 97-113
[Robert85] Y. Robert and D. Trystram, An Orthogonal Systolic Array for the Algebraic Path Problem, Report 533 IMAG 1985, Informatique et Mathematiques Appliquée de Grenoble
[Robertson91] W. Robertson and W.J. Phillips, A System of Systolic Modules for the MUSIC Algorithm, IEEE Transactions on Signal Processing, Vol. 39, No. 11, 1991, pp 25242534
[Rote85] G. Rote, A Systolic Array Algorithm for Algebraic Path Problems (shortest paths, matrix inversion), Computing, vol. 34, 1985, pp 192-219
[Rutenbar88] R.A. Rutenbar and D.E. Atkins, Systolic Routing Hardware: Performance Evaluation and Optimization, IEEE Trans on CAD., Vol. 7, 1988, pp 397-410
[Sarkar90] S. Sarkar, A.K. Majumdar, Fast Fourier Transform Using Linear Tagged Systolic Array, 1990 IEEE Region 10 Conference on Computer and Communication Systems, IEEE TENCON'90, pp 289-293
[Schreiber82] R. Schreiber, Systolic Arrays for Eigen Value Computation, Proc. SPIE, Vol. 341, Real Time Signal Processing V, 1982, pp 27-34
[Schreiber84] R. Schreiber, Computing Generalized Inverses and Eigenvalues of Symmetric Matrices Using Systolic Arrays, Comput. Math. Appl. Sci. and Eng., 1984, pp 285295
[Schroder91] H. Schroder, V.K. Murthy and E.V. Krishnamurthy, Systolic Algorithm for Polynomial Interpolation and Related Problems, Parallel Computing, Vol. 17, No. 45, Jul 1991, pp 493-503
[Schwartz84] D.A. Schwartz and T.P.Barnwell, A Graph Theoretic Technique for the Generation of Systolic Implementations for Shift Invariant Flow Graphs, In International Conference on Acoustics, Speech and Signal Processing, March 1984, pp 8.3.1-8.3.4
[Schwiegel88] U. Schwiegelshohn, L.A. Thiele, Systolic Array for Assignment Problem, IEEE Trans. Computer, Vol. 37, No. 11, 1988, pp 1422-1425
[Scott86] D.S. Scott, M.T. Heath and R.C. Ward, Parallel Block Jacobi Eigenvalue Algorithms for Using Systolic Arrays, Linear Algebra and Its Applications, Vol. 77, 1986, pp 345-356
[Shanbhag91] N.R. Shanbhag, An Improved Systolic Architecture for 2-D Digital Filters, IEEE Transactions on Signal Processing, Vol. 39, No. 5, May 1991
[Sikstrom87] B. Sikstrom, L. Wanhammer, M. Afghahi, J.A. Penez, High Speed 2-D Cosine Transform Chip, Integration, Vol. 5, No. 2, 1987, pp 159-170
[Siu91] Wan-Chi Siu, Kar-Lik Wong, Realization of Discrete Fourier Transforms Using a Nesting Algorithm, International Journal of Electronics, Vol. 70, No. 4, Apr 1991, pp 671-689
[Smith90] J.E. Smith, W.C. Hsu, C. Hsiung, Future General Purpose Supercomputer Architectures, Proceedings of Supercomputer '90, pp 796-804
[Strang80] G. Strang, Linear Algebra and Its Applications, Academic Press, Inc., 1980
[Sung87] T. Sung and Y. Hu, Parallel VLSI Implementation of Kalman Filter, [EEE Transactions on Aerospace and Electronic Systems, AES, Vol. 23, 1987, pp 215-224
[Takao91] K. Takao and C.S. Boon, Two-Mode Adaptive Beamformer Using a Systolic Array, IEE Proceedings-F, Vol. 138, No. 5, October 1991, pp 483-488
[Tasic93] J. Tasic, D. Caf and M. Gusev, Solving Linear Systems of Equations, Signal Processing, Vol. 31, No. 1, March 1993, pp 91-96
[Torrero85] Edward A. Torrero, Ed., Next Genereation Computers, IEEE Press, 1985
[Toverud88] M. Toverud and V. Anderson, CESAR: A Programmable High-Performance Systolic Array Processor, Proc. Int'I Conf. Computer Design, JEEE CS Press, Los Alamitos, Calif., 1988, pp 414-417
[Treleaven82] P.C. Treleaven, D.R. Brownbridge, and R.P. Hopkins, Data-Driven and DemandDriven Computer Architectures, ACM Computer Surveys, Vol. 14, No. 1, March 1982, pp 93-143
[Tyagi90] A. Tyagi, M. Bayoumi, Systolic Array Implementation of Image Segmentation by a Directed Split and Merge Procedure, IEEE Proceedings of International Conference on Pattern Recognition, Vol. 2, pp 491-493
[Ulman84] J.D. Ulman, Computational Aspects of VLSI, Computer Science Press, 1984
[Umeo89] H. Umeo and T. Asano, Systolic Algorithms for Computational Geometry Problems-A Survey, Computing, Vol. 41, 1989, pp 19-40
[VanSwaaij90] Van Swaaij, F.X.B. Michael, F.V.M. Catthoor, and H.J. De Man, Deriving ASIC Architecture for the Hough Transform, Parallel Computing, Vol. 16, No. 1, Nov 1990, pp 113-121
[Varga62] R.S. Varga, Matrix Iterative Analysis, Prentice Hall, 1962
[Wan90a] C.R. Wan, Systolic Array Architecture for Matrix Triangularisation, Sonar Group Internal Report, Loughborough University of Technology, August 1990
[Wan90b] C.R. Wan, Systolic Architecture for Signal Processing, Sonar Group Internal Report, Loughborough University of Technology, May 1990
[Wan91a] C.R. Wan, Computational Loads and Hardware Requirement for Direction Finding Algorithms, Sonar Group Internal Report, Loughborough University of Technology, February 1991
[Wan9lb] C.R. Wan, Systolic Array for Singular Value Decomposition, Sonar Group Internal Report, Loughborough University of Technology, May 1991
[Wan9lc] C.R. Wan and J.W.R. Griffiths, Circular Array and High Resolution Algorithms, Sonar Group Internal Report, Loughborough University of Technology, September 1991
[Wan91d] C.R. Wan, Systolic Architecture for MUSIC Algorithm, Sonar Group Internal Report, Loughborough University of Technology, December 1991
[Wan92] C.R. Wan, A Transputer Based Experimental System for Parallel Architectures, Sonar Group Internal Report, Loughborough University of Technology, January 1992
[Wan93a] C.R. Wan and D.J. Evans, A Systolic Architecture for Capon's Directions-of-Arrival (DOA) Estimation Method, Parallel Algorithms and Applications, Vol. I, 1993, pp 99-114
[Wan93b] C.R. Wan and D.J. Evans, A Systolic Array Architecture for Linear and Inverse Matrix Systems, Parallel Computing, Vol. 19, 1993, p 303-321
[Wan94a] C.R. Wan and D.J. Evans, Systolic Array Architecture for QR Decomposition of Block Structured Sparse Systems, Parallel Computing, Vol. 20, No. 4, June 1994, pp 903-914
[Wan94b] C.R. Wan and D.J. Evans, Systolic Array for Matrix Inversion by Newton Iteration, 1995, submitted for publication
[Ward84] C.R. Ward, A.J. Robson, P.J. Hargrave, and J.G. McWhirter, Application of a Systolic Array to Adaptive Beamforming, IEE Proc., Vol. 131, Oct. 1984, pp 638645
[Weil71] R.L. Weil and P.C. Kettler, Rearranging Matrices to Block Angular Form for Decomposition (and other) Algorithms, Management Science, Vol 18, pp 98-108, 1971
[Weiser81] U. Weiser and A. Davis, A Wavefront Notation Tool for VLSI Array Design, in VLSI Systems and Computations, H.T. Kung, R.F. Sproull and G.L. Steele Jr. (eds.), Carnigie-Mellon University, Computer Science Press, Oct. 1981, pp 226-234
[Wenyang91] C. Wenyang, L. Yanda and J. Yue, Systolic Realization for 2D Convolution Using Configurable Functional Method in VLSI Parallel Array Designs, Proc. IEEE Computers and Digital Technology, Vol. 138, No. 5, Sep. 19.91, pp 361-370
[Whitehou85] H.J. Whitehouse, J.M. Speiser and K. Bromley, Signal Processing Applications of Concurrent Array Processor Technology, in [KungSY85a], pp 25-41
[Witbrock90] Michael Witbrock and Marco Zagha, An Implementation of Backpropagation Learning on GF11, A Large SIMD parallel Computer, Paraltel Computing, 14(3), 1990, pp 329-346
[Woods90] R.F. Woods, B.P. McGovern, J.V. McCanny, Systolic Array Architectures for Parameterised Multiplexed IIR Filters, IEE Colloquium No. 95, 1990
[Yaacoby88] Y. Yaacoby and P.R. Cappelo, Scheduling a System of Affine Recurrence Equations onto a Systolic Array, In International Conference on Systolic Arrays, May 1988, pp 373-382
[Young71] D.M. Young, Iterative Solution of Large Linear Systems, Academic Press, 1971
[Zah85] S.H. Zah and K. Hwang, Polynomial Division on Systolic Arrays, IEEE Trans. Comput. 1985, Vol. 34, No. 6, pp 577-578
[Zhang90] J.Y. Zhang, A Very Fast Kalman Filter for Image Restoration, Proceedings of IEEE International Symposium on Circuits and Systems, Vol. 1, 1990, pp 250-253```


[^0]:    - Systolic algorithm for QR decomposition
    -- Using Square-root-free Method

[^1]:    - File Name: "A:OR1"

