

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

COMMONS DEED
Attribution-NonCommercial-NoDerivs 2.5
You are free:
 to copy, distribute, display, and perform the work
Under the following conditions:
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 Legal Code (the full license).
<u>Disclaimer</u> 曰

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

BLDSC NO:- DX81054 LOUGHBOROUGH UNIVERSITY OF TECHNOLOGY LIBRARY AUTHOR/FILING TITLE MARGARITIS, KG ACCESSION/COPY NO. DIGG45/02 CLASS MARK VOL. NO. LOAN COPY 3.0. HUN 1989 5 .111 1991 - 6 JUL 1990 6 JUH 1990 - 2 JUL 1993 - 5 JUL 1991 18 061 1990 001 6645 02

• • .

A STUDY

OF SYSTOLIC ALGORITHMS FOR VLSI PROCESSOR ARRAYS AND OPTICAL COMPUTING

By

K.G.Margaritis, Dipl.Eng., M.Sc.

VOLUME II

A Doctoral Thesis Submitted in partial fulfilment of the requirements for the Award of Doctor of Philosophy of the Loughborough Univerity of Technology October, 1987.

Supervisor: Professor D.J. Evans, D.Sc., F.I.M.A., F.B.C.S.

(c) by K.G.Margaritis, 1987.

hanghborouses werstay	
of Taua crety	
Deep Mar 88	
Clust	
in NILLUS on	

•

. .

, . A STUDY OF SYSTOLIC ALGORITHMS FOR VLSI PROCESSOR ARRAYS AND OPTICAL COMPUTING

By K.G. Margaritis

ABSTRACT

This thesis presents some new systolic algorithms for numerical computation, that are suitable for implementation on VLSI processor arrays or optical processors.

Chapter 1 is an introduction to the environment for the development of the systolic approach, followed by an overview of major research areas in systolic systems. Chapter 2 contains basic mathematical definitions and a brief introduction to specific areas of numerical analysis. Chapter 3 starts with some basic definitions and terminology for systolic computing; then fundamental systolic algorithms are described. Following is a review of some transformation techniques and an introduction to systolic programming and soft-systolic simulation. Finally, systolic and optical computing are combined, and a framework for developing systolic algorithms is outlined.

Chapter 4 investigates systolic algorithms for the solution of polynomial equations, and the systolic calculation of the roots of the characteristic equation of certain matrices. Chapter 5 presents systolic algorithms for the efficient solution and the updating of the solution of linear systems of equations, using LU decomposition. Chapter 6 develops the concept of pipelining systolic arrays, as well as the combination of area and time expansion, in iterative solution of linear systems of equations, based on series of systolic matrix-vector multiplications. Chapter 7 further develops the idea of expanding iterative systolic algorithms in area and/or in time. The systolic implementation of successive matrix-matrix multiplications discussed and then a group of algorithms based on matrix is powering is studied. Chapter 8 presents some optical systolic algorithms. The direct mapping of VLSI systolic algorithms on optical processors is discussed, and then, the Product processor is used for the optical systolic Outer implementation of basic matrix computations.

Chapter 9 completes this thesis with some general conclusions, and suggestions for further research. A comprehensive list of references is also given, and an Appendix on the OCCAM programming language, and programs simulating some of the systolic designs presented.

KEYWORDS: parallel processing, systolic algorithms, VLSI processor arrays, optical computing, polynomial equations, linear systems of equations, matrix eigen-problem solution, matrix functions.

A STUDY

OF SYSTOLIC ALGORITHMS FOR VLSI PROCESSOR ARRAYS AND OPTICAL COMPUTING

ABSTRACT

This thesis presents some new systolic algorithms for numerical computation, under the framework of being suitable for implementation on to VLSI processor arrays or optical processors.

Chapter 1 gives an introduction to the environment and background for the development of the systolic approach, followed by an overview of the major research areas in systolic systems; finally the thesis organization is described.

Chapter 2 contains basic mathematical definitions and a brief introduction to specific areas of numerical analysis; further, the algorithms used in subsequent chapters are briefly discussed.

Chapter 3 starts with an example, through which basic definitions and terminology in systolic computing are intro-

duced; then some fundamental systolic algorithms are described. Following is a review of some techniques for deriving and/or modifying systolic systems; further the concepts of systolic programming, simulation, and the softsystolic paradigm are introduced. Finally, the combination of systolic and optical computing is discussed and a framework for developing systolic algorithms is outlined.

Chapter 4 investigates the systolic implementation of algorithms for the solution of polynomial equations. First, the derivation and operation of the systolic designs for two traditional methods are discussed in detail; then the systolic calculation of the roots of the characteristic equation of a symmetric tridiagonal matrix is described, as well as some other aspects of the systolic computation of certain types of characteristic equations. Finally, a general ring architecture, for the iterative solution of polynomial equations is proposed.

Chapter 5 presents systolic algorithms for the efficient solution of linear systems of equations, using LU decomposition. Initially, the efficiency of the basic algorithm is improved using mathematical techniques; then the problem of updating LU factors is discussed, in the context of Linear Programming. Further, the LU decomposition with partial pivoting is used for the systolic calculation of the eigenvectors of a symmetric tridiagonal matrix.

Chapter 6 develops the concept of pipelining systolic

- ii -

arrays, as well as the combination of area and time expansion, in iterative systolic algorithms based on matrixvector multiplications. Firstly, an improved systolic design for matrix-vector multiplication is presented; then, area and/or time efficient pipelines for the iterative solution of linear systems are described. Further, pipelined structures for cyclic reduction and multi-coloring techniques are investigated. Finally, an alternative matrix-vector multiplication design for area expansion applications is discussed.

Chapter 7 further develops the idea of expanding iterative systolic algorithms in area and/or in time. Initially the systolic implementation of successive matrix-matrix multiplications is discussed, and then a group of algorithms based on matrix powering is studied. Thus, the basic iterative methods of chapter 6 are modified, and three closely related methods solving the matrix eigenproblem are investigated. Further, systolic matrix polynomial computations are implemented, as well as the approximation of matrix functions.

Chapter 8 presents some optical systolic algorithms. Firstly, the direct mapping of VLSI systolic algorithms on optical processors is discussed, and the optical implementation of fundamental systolic algorithms is presented. Then, the Outer Product processor is introduced and modified for banded matrix computations; further, the same processor is

- iii -

used for a series of optical systolic algorithms, based on the Gauss Elimination process.

Chapter 9 completes this thesis with some general conclusions, and suggestions for further research. A comprehensive list of references is also given, and an Appendix on OCCAM programming language, and programs simulating some of the systolic designs presented.

CONTENTS

VOLUME I

	Page
ACKNOWLEDGEMENTS	i
ABSTRACT	iii
CONTENTS	vii
LIST OF FIGURES	xiii
LIST OF TABLES	xx
LIST OF PROGRAMS	xxi

.

CHAPTER 1

INTRODUCTION

1.1	ENVIRONMENT FOR DEVELOPMENT	OF SYSTOLIC APPROACH .	1
1.2	REVIEW OF SYSTOLIC SYSTEMS R	RESEARCH	15
1.3	ORGANIZATION OF THE THESIS .		26

CHAPTER 2

BASIC MATHEMATICAL DEFINITIONS

2.1	POLYNOMIAL EQUATIONS	32
	2.1.1 Solution of polynomial equations	37
2.2	MATRICES	44
	2.2.1 Eigenvalues and eigenvectors	51
	2.2.2 Matrix and vector norms	53

	2.2.3 Matrix functions	55
2.3	LINEAR SYSTEMS OF EQUATIONS	59
	2.3.1 Direct methods	60
	2.3.2 Iterative methods	67
2.4	MATRIX EIGENVALUE PROBLEM	77
	2.4.1 The Power method	77
	2.4.2 Characteristic polynomial methods	80
	2.4.3 Inverse iteration	83
2.5	MISCELLANEOUS ITEMS	85
	2.5.1 The Simplex method	85
	2.5.2 Differential equations	87

CHAPTER 3

SYSTOLIC ALGORITHMS AND ARCHITECTURES

		95
3.1	BASIC DEFINITIONS AND TERMINOLOGY	
	3.1.1 A simple example	95
	3.1.2 Measures and characteristics of systolic systems	104
	3.1.3 Framework for systolic algorithms for VLSI .	110
3.2	SOME BASIC SYSTOLIC ALGORITHMS	113
	3.2.1 Systolic matrix-vector multiplication	114
	3.2.2 Systolic matrix-matrix multiplication	120
	3.2.3 Systolic solution of linear systems	125
3.3	TRANSFORMATION TECHNIQUES	137
	3.3.1 Retiming method	137
	3.3.2 Cut Theorem	143
	3.3.3 Area-Time expansion	150
	3.3.4 Rotate and Fold (R+F) method	156

.

3.4	SYSTOLIC PROGRAMMING AND SIMULATION	167
	3.4.1 The Warp machine	169
	3.4.2 The Wavefront Array Processor (WAP)	172
	3.4.3 The INMOS Transputer	176
	3.4.4 The Soft-systolic approach	179
	3.4.5 Soft-systolic simulation using OCCAM	183
3.5	OPTICAL COMPUTING AND SYSTOLIC ARCHITECTURES	191
	3.5.1 Optical signal transmission	194
	3.5.2 Optical systolic architectures	197
	3.5.3 A general framework for systolic algorithms.	207

CHAPTER 4

SYSTOLIC SOLUTION OF POLYNOMIAL EQUATIONS

4.1	INTRODUCTION	211
4.2	SYSTOLIC DESIGNS FOR BERNOULLI'S METHOD	214
	4.2.1 Systolic design derivation	216
	4.2.2 Implementation details	221
4.3	SYSTOLIC DESIGNS FOR THE ROOT-SQUARING METHOD	227
	4.3.1 Systolic design derivation	229
	4.3.2 Implementation details	235
4.4	SYSTOLIC DESIGN FOR THE CALCULATION OF THE EIGEN- VALUES OF A SYMMETRIC TRIDIAGONAL MATRIX	242
	4.4.1 Sturm sequence pipeline	245
	4.4.2 Systolic eigenvalue solver	248
4.5	CONCLUSIONS	253
	4.5.1 Iterative methods	253
	4.5.2 Characteristic polynomial computation	254

- viii-

CHAPTER 5

SYSTOLIC LU DECOMPOSITION

5.1	INTRODUCTION	256
5.2	THE R+F METHOD ON SYSTOLIC BLOCK LU DECOMPOSITION	260
	5.2.1 Block R+F LU decomposition	265
	5.2.2 Systolic implementation of block R+F LU decomposition	272
5.3	SYSTOLIC LU FACTORISATION FOR SIMPLEX UPDATES	289
	5.3.1 LU updating method	292
	5.3.2 Systolic LU modification	295
5.4	SYSTOLIC DESIGNS FOR THE CALCULATION OF THE EIGEN- VECTORS OF A SYMMETRIC TRIDIAGONAL MATRIX	305
	5.4.1 Systolic design	308
5.5	CONCLUSIONS	313

VÔLUME II

i	• • •		•	•	• •	•••	•	•	••	•	•		•	• •	•	••	•	•	•	٠		• •	•	•	• •	•	••	•	• •		•	т	AC	ТF	ABS
v	• • •	• • •	•	•	• •	••	•	•	• •	•	•	• •	•	•	•	••	•	•	•	•	••	• •	•	•	• •	•	••	•	••	••	•	S	NT	TE	CON
xi		• • •	•	•	•	• •	•		• •	•	• •	• •	•	•	•	• •	•	• •	•	•	••	•	•	••	•	•		S	RĔ	GU	FI	' 1	OF	т	LIS
xviii	•••	• • •	•	•		• •	•	•	•	•	• •	•	•	•	•	••	•	• •	•	•	••	• •	•	••	•	•	• •	;	ES	BL	ТА	• •	OF	т	LIS
xix	• • •	• • •	•	•		• •	•		•	•	• •	•	•	•	•	••	•		•		••	•		••	•	•	s	M	RA	00	PR		OF	T	LIS

СНАРТЕК б

SYSTOLIC MATRIX VECTOR MULTIPLICATION PIPELINES

6.1	INTRODUCTION	315
6.2	IMPROVED SYSTOLIC MATRIX VECTOR MULTIPLICATION	318
	6.2.1 Systolic array derivation	320

6.3	IMPROVED SYSTOLIC DESIGNS FOR THE ITERATIVE SOLUT- ION OF LINEAR SYSTEMS	328
	6.3.1 Improved iterative systolic designs	329
	6.3.2 Unidirectional mvm array for J,JOR methods .	339
6.4	SYSTOLIC NETWORKS FOR ITERATIVE METHODS USING CYCLIC REDUCTION	346
	6.4.1 Systolic designs	352
6.5	P-CYCLIC MATRICES AND MULTI-COLORING TECHNIQUES	360
	6.5.1 P-cyclic matrices	360
	6.5.2 Multi-coloring techniques	370
6.6	CONCLUSIONS	378

CHAPTER 7

SYSTOLIC ALGORITHMS USING MATRIX POWERS

7.1	INTRODUCTION	385
7.2	SYSTOLIC DESIGNS FOR SUCCESSIVE MATRIX SQUARING	389
	7.2.1 Systolic pipeline designs	389
	7.2.2 Systolic iterative designs	398
7.3	SYSTOLIC ITERATIVE SOLUTIONS OF LINEAR SYSTEMS USING MATRIX POWERS	404
	7.3.1 Systolic designs	406
7.4	SYSTOLIC DESIGNS FOR EIGENVALUE-EIGENVECTOR COMPU- TATION USING MATRIX POWERS	412
	7.4.1 Systolic design considerations	415
	7.4.2 Systolic designs	417
7.5	SYSTOLIC COMPUTATION OF THE EXPONENTIAL OF A MATRIX	425
	7.5.1 Systolic designs	426
7.6	CONCLUSIONS	436
	7.6.1 Systolic inversion using matrix powers	437
	7.6.2 Systolic computation of matrix functions	439

CHAPTER 8

OPTICAL SYSTOLIC ALGORITHMS

8.1	INTRODUCTION	442
8.2	OFTICAL SYSTOLIC BANDED MATRIX MULTIPLICATION	446
	8.2.1 Mapping of a R+F algorithm on an optical processor	446
	8.2.2 Mapping of the unidirectional mmm array on an optical processor	453
8.3	OPTICAL SYSTOLIC LU DECOMPOSITION AND SOLUTION OF TRIANGULAR SYSTEMS	456
	8.3.1 Optical LU decomposition	456
	8.3.2 Optical solution of triangular systems	464
8.4	OPTICAL SYSTOLIC ALGORITHMS USING OUTER PRODUCTS .	470
	8.4.1 Banded matrix multiplication	470
	8.4.2 Banded matrix LU decomposition	479
8.5	OPTICAL GAUSS ELIMINATION USING OUTER PRODUCTS	490
	8.5.1 Optical implementation	498
8.6	CONCLUSIONS	506

CHAPTER 9

CONCLUSIONS

9.1 THESIS SUMMARY	511
9.2 SOME FURTHER SUGGESTIONS	523
REFERENCES	528
APPENDIX	552
I. Brief introduction to OCCAM	552
II. Loughborough implementation of OCCAM	561
III. Soft-systolic simulation programs	570

LIST OF FIGURES

.

·	Page
Fig.1.1.1. Systolic system as special-purpose device	4
Fig.1:1.2. Signal processing application design	4
Fig.1.1.3. Hardware library design	4
Fig.1.1.4. Linear systolic array	7
Fig.1.1.5. Systolic system communication geometries	7
Fig.3.1.1. Polynomial multiplication array with broadcasting	97
Fig.3.1.2. Polynomial multiplication array with bidirectional dataflow	99
Fig.3.1.3. Polynomial multiplication array with unidirectional dataflow	101
Fig.3.1.4. IPS cell designs	103
Fig.3.2.1. Full matrix-vector multiplication array	116
Fig.3.2.2. Banded matrix-vector multiplication array	118
Fig.3.2.3. Full matrix-matrix multiplication array	122
Fig.3.2.4. Banded matrix-matrix multiplication array	124
Fig.3.2.5(a). Full matrix triangularization array	127
Fig.3.2.5(b). Cell specification	128
Fig.3.2.6. Banded matrix LU decomposition array	131
Fig.3.2.7. Triangular system solver	134
Fig.3.3.1. Application of the retiming method	142
Fig.3.3.2. Fault-tolerant array	145
Fig.3.3.3. Two-level pipelined array	145

Fig.3.3.4. Application of the cut theorem	148
Fig.3.3.5. Systolic ring architecture	148
Fig.3.3.6. Application of area-time expansion	153
Fig.3.3.7. Application of R+F method on LU decomposotion of tridiagonal matrices	160
Fig.3.3.8. Application of R+F method on triangular bidiagonal system solution	163
Fig.3.3.9. Systolic arrays for R+F method	165
Fig.3.4.1. Warp machine architecture	171
Fig.3.4.2. WAP architecture	174
Fig.3.4.3. Transputer architecture	178
Fig.3.4.4. Logical structure of soft-systolic simulation programs in OCCAM	185
Fig.3.5.1. Optical signal transmission	195
Fig.3.5.2. Optical processor for systolic matrix-vector multiplication	198
Fig.3.5.3. Operation of optical processor	· 203
Fig.4.2.1. Dataflow for bidirectional array design	218
Fig.4.2.2. Dataflow for systolic ring design (n=5)	220
Fig.4.2.3. Dataflow for systolic ring design (n=4)	222
Fig.4.2.4. Input-output for systolic ring	224
Fig.4.2.5. Systolic system for Bernoulli method	226
Fig.4.3.1. A simple systolic design for the calculation of the coefficients, b _i , i=0,1,2,,n	232
Fig.4.3.2. Data flow for semi-systolic array design	234
Fig.4.3.3. A 'purely' systolic array design	236
Fig.4.3.4. Final systolic array design	238
Fig.4.3.5. A systolic system for the Graeffe root squaring method	240
Fig.4.4.1. Roots for Sturm sequence polynomial	243

Fig.4.4.2. Sturm sequence pipeline	246
Fig.4.4.3. Systolic system overview	249
Fig.5.2.1(a). Block (2x2) LU, LDU decomposition array .	261
Fig.5.2.1(b). Cell definitions	262
Fig.5.2.1(c). Cell definitions	263
Fig.5.2.2. Preprocessor array	264
Fig.5.2.3(a). Block (2x2) tridiagonal system	266
Fig.5.2.3(b). LU, LDU decomposition of a block (2x2) tridiagonal matrix	266
Fig.5.2.4(a). Block (2x2) R+F LU decomposition	269
Fig.5.2.4(b). Block (2x2) R+F LDU decomposition	270
Fig.5.2.5(a). Preprocessor array and i/o format for k=5 (odd)	273
Fig.5.2.5(b). I/O format for $k=4$ (even)	275
Fig.5.2.6. Block (2x2) R+F LU, LDU decomposition array.	276
Fig.5.2.7. Block (2x2) R+F triangular system solution .	282
Fig.5.3.1. Matrix configurations for the modification of LU factors	290
Fig.5.3.2. Major steps of the modification of LU factors (n=5)	294
Fig.5.3.3. Parallel modification of LU factors	296
Fig.5.3.4(a). Rectangular array configuration	298
Fig.5.3.4(b). Cell definitions	299
Fig.5.3.5(a). Linear array configuration	301
Fig.5.3.5(b). Cell definitions	302
Fig.5.3.6. General case of modification of LU factors .	303
Fig.5.4.1. Step 4 of Gaussian Elimination with partial pivoting	306
Fig.5.4.2(a). Systolic array for Gaussian Elimination of a symmetric tridiagonal system	309
Fig.5.4.2(b). Cell definitions	310

.

Fig.6.2.1. Dataflow for the bidirectional mvm array ... 319 Fig.6.2.2. A simple systolic design for mvm 321 Fig.6.2.3. Dataflow for a semi-systolic mvm array 323 Fig.6.2.4. Dataflow for unidirectional mvm (y is 324 delayed) Fig.6.2.5. Dataflow for unidirectional mvm (x is delayed) 326 327 Fig.6.2.6. Systolic array for unidirectional mvm Fig.6.3.1. Pipeline block for J method; w=5, p=q=3 330 Fig.6.3.2. Pipeline block for JOR method 331 Fig.6.3.3. Pipeline block for GS method 332 Fig.6.3.4. Pipeline block for SOR method 333 Fig.6.3.5. Special cell definitions for pipeline blocks 334 in Fig.6.3.1-4 Fig.6.3.6. Modified pipeline block for J, JOR methods .. 337 Fig.6.3.7. Modified pipeline block for GS, SOR methods. 338 Fig.6.3.8. Preprocessor for J, JOR methods 341 Fig.6.3.9. Preprocessor for GS, SOR methods 342 Fig.6.3.10. Pipeline block for J, JOR methods using the 343 unidirectional mvm array Fig.6.3.11. Preprocessor for the pipeline in Fig.6.3.10 344 Fig.6.4.1(a). 2-cyclic ordering of tridiagonal matrices 347 for n=4 (even) and n=5 (odd) Fig.6.4.1(b). 2-cyclic ordering of the Jacobi matrices 348 Fig.6.4.2. Pipeline block for J method (2-cyclic 353 matrices) Fig.6.4.3. Pipeline block for JOR method (2-cyclic 354 matrices) Fig.6.4.4. Pipeline block for GS method (2-cyclic 355 matrices) Fig.6.4.5. Overall pipeline configuration for the 359 iterative solution of 2-cyclic systems

- xiv -

Fig.6.5.1(a). 2-cyclic ordering using 3 and 5 -point stencils	362
Fig.6.5.1(b). 3 and 4 -cyclic ordering using 4-point stencils	363
Fig.6.5.2(a). Systolic network for J method (p-cyclic matrices)	366
Fig.6.5.2(b). Systolic network for JOR method (p-cyclic matrices)	366
Fig.6.5.3(a). Systolic network for GS method (p-cyclic matrices)	367
Fig.6.5.3(b). Systolic network for SOR method (p-cyclic matrices)	367
Fig.6.5.4. 2-color ordering using 5-point stencil	372
Fig.6.5.5. 3-color ordering using 7-point stencil	372
Fig.6.5.6. 4-color ordering using 9-point stencil	373
Fig.6.5.7. Systolic networks for r-color ordering	375
Fig.6.6.1. Banded-full mmm systolic array	379
Fig.6.6.2. Time and area expansion for mvm computation.	381
Fig.6.6.3. Re-usable mvm array	382
Fig.7.2.1. Time and area expansion for matrix squaring computation	390
Fig.7.2.2(a). Banded matrix multiplication on unidire- ctional hex-array	391
Fig.7.2.2(b). Banded matrix squaring on a unidirection- al hex-array $w_A = 3$, $p_A = q_A^{=2}$	392
Fig.7.2.3. Banded matrix squaring $w_A = 4$, $p_A = 3$, $q_A = 2$	395
Fig.7.2.4. Matrix squaring for a banded matrix A with bandwidth w=5, p=q=3	397
Fig.7.2.5. Dense matrix multiplication on unidirection- al hex-array, n=3	399
Fig.7.2.6. Matrix squaring pipeline block for a full (nxn) matrix A, with n=3	400
Fig.7.2.7. Re-usable matrix multiplication array, $n=3$.	401
Fig.7.2.8. Iterative array configuration for successive matrix squaring	403

Fig.7.3.1.	Iterative array configuration for the J,JOR- Hotelling methods	407
Fig.7.3.2.	Iterative array configuration for the GS, SOR-Hotelling methods	408
Fig.7.3.3.	Matrix squaring and matrix-vector inner pro- duct step for a banded matrix C, w=5, $p=q=3$.	410
Fig.7.4.1.	Power method pipeline block for a banded ma- trix A with bandwidth w=5, p=q=3	419
Fig.7.4.2.	Matrix Squaring method pipeline block for a banded matrix A with bandwidth $w=5$, $p=q=3$	420
Fig.7.4.3.	Iterative array configuration for Matrix Squaring	423
Fig.7.5.1.	Time and area expansion for mmips operation.	429
Fig.7.5.2.	Iterative array configuration	430
Fig.7.5.3.	Pipeline configuration for $k=2$, $j=1$, $w=3$	431
Fig.7.5.4(a	a). Banded matrix multiplication, $w_A = 4$, $p_A = 3$, $q_A = 2$, $w_B = 3$, $p_B = 2$, $q_B = 2$	433
Fig.7.5.4(1	b). Banded matrix multiplication, $w_A = 3$, $p_A = 2$, $q_A = 2$, $w_B = 4$, $p_B = 3$, $q_B = 2$	434
Fig.8.2.1.	R+F matrix multiplication	447
Fig.8.2.2.	R+F matrix multiplication array	448
Fig.8.2.3.	Details of optical processor for R+F mmm	450
Fig.8.2.4.	Matrix multiplication optical processor	451
Fig.8.2.5.	Details of optical processor for unidirect- ional mmm	454
Fig.8.3.1.	Optical processor for LU decomposition	458
Fig.8.3.2.	Details of optical processor	459
Fig.8.3.3.	Optical processor for LU decomposition of a tridiagonal matrix	462
Fig.8.3.4.	Operation of optical processor for R+F LU decomposition	462
Fig.8.3.4.	(continued)	463
Fig.8.3.5.	Optical processor for triangular system solution	465

.

. .

.

Fig.8.3.6. Optical processor for bidiagonal system solution	467
Fig.8.3.7. Optical processor for R+F triangular system solution	468
Fig.8.4.1. Matrix multiplication using outer products .	471
Fig.8.4.2. Banded matrix multiplication using outer products	472
Fig.8.4.3. Optical processor for matrix multiplication.	474
Fig.8.4.4. Optical processor for banded matrix multi- plication	476
Fig.8.4.5. Phases of a matrix multiplication	477
Fig.8.4.6. LU decomposition using outer products	480
Fig.8.4.7. Banded LU decomposition using outer pro- ducts	482
Fig.8.4.8. LU decomposition using outer product	484
Fig.8.4.9. Optical processor for full matrix LU deco- mposition	485
Fig.8.4.10. Optical processor for banded LU decompo- sition	487
Fig.8.4.11. Phases of a full step of LU decomposition .	488
Fig.8.5.1. Triangularization of A	492
Fig.8.5.2. Gauss Elimination for A, <u>b</u> \dots	494
Fig.8.5.3. Back substitution for A, \underline{b}	495
Fig.8.5.4(a). Gauss-Jordan method for matrix inversion.	496
Fig.8.5.4(b). Gauss-Jordan method for matrix inversion.	497
Fig.8.5.5. Optical processor for matrix triangularizat- ion	499
Fig.8.5.6. Optical processor for Gauss Elimination	499
Fig.8.5.7. Optical processor system for Gauss-Jordan method	502
Fig.8.5.8. Optical processor for matrix inversion	504

.

,

LIST OF TABLES

able	1.1.1.	Selection of major applications of systolic systems	5
able	3.5.1.	Types of systolic algorithms	209
able	6.3.1.	Area-time requirements for systolic pipelines in Fig.6.3.1-5	335
able	6.3.2.	Area-time requirements for systolic pipelines in Fig.6.3.6-9	340
able	6.5.1.	Comparison of the area-time requirements of the pipelines for normal and p-cyclic ordering	368
able	6.5.2.	Comparison of the area-time requirements of the pipelines for natural and r-colored ordering	377
Table	7.5.1.	Optimum (k,j) for given ε and $ A _1 \cdots$	427

.

Page

LIST OF PROGRAMS

• ·

	Page
A.1.1. Bernoulli's method	574
A.1.2. Graeffe (Root Squaring) method	576
A.1.3. Sturm sequence method	578
A.1.4. Horner's scheme	581
A.1.5. Bairstow method	582
A.1.6. Characteristic polynomial of a lower Hessenberg matrix	584
A.2.1. Preprocessor for block (2x2) R+F LU/LDU decomposition	585
A.2.2. Block (2x2) R+F LU/LDU decomposition	586
A.2.3. Block (2x2) R+F triangular system solution	589
A.2.4. LU updating on orthogonal array	591
A.2.5. LU updating on linear array	593
A.2.6. LU decomposition with partial pivoting	595
A.2.7. Backsubstitution for Inverse Iteration	596
A.2.8. Linear array for Inverse Iteration	597
A.3.1. Unidirectional mvm array	602
A.3.2. Pipeline of mvm arrays for J,JOR methods	603
A.3.3. Preprocessor for iterative methods	604
A.3.4. Pipeline for J method (Cyclic reduction)	606
A.3.5. Pipeline for JOR method (Cyclic reduction)	607
A.3.6. Unidirectional mvm array with local memory	609

•

A.3.7.	Iterative mvm array (time expansion)	611
A.4.1.	Unidirectional mmm array	613
A.4.2.	Pipeline of mmm arrays	614
A.4.3.	Iterative array for J,JOR method	615
A.4.4.	Iterative array for GS, SOR method	617
A.4.5.	Pipeline of mvm and mmm arrays for J, JOR methods	618
A.4.6.	Pipeline for Power method	620
A.4.7.	Pipeline for Matrix Squaring method	622
A.4.8.	Iterative array for Power method	623
A.4.9.	Iterative array for Matrix Squaring method	625
A.4.10	. Iterative array for matrix exponential	627
A.4.11	. Pipeline for matrix polynomial	629
A.5.1.	Optical systolic mvm using Inner Products	631
A.5.2.	Optical systolic mvm using Outer Products	632
A.5.3.	Optical systolic mmm using Inner Products	633
A.5.4.	Optical systolic mmm using Outer Products	634
A.5.5.	DMAC algorithm	635
A.6.1.	Library routines for soft-systolic simulation of hard/hybrid/soft - systolic algorithms	636
A.6.2.	Library routines for soft-systolic simulation of optical - systolic algorithms	640

.

.

CHAPTER 6

SYSTOLIC MATRIX VECTOR MULTIPLICATION PIPELINES

6.1 INTRODUCTION

The matrix vector multiplication (mvm) computation is probably the single most useful operation in signal processing, since many basic processes, such as convolution, FIR filtering, Discrete Fourier Transform, can be regarded as special mvm cases [160], [287]. The original systolic mvm algorithm in [181] (see section 3.2) has been applied in many varied problems and has undergone numerous modifications.

For example, a two-level pipelined algorithm is proposed in [158] and the bit-level hardware implementation of a mvm systolic array is discussed in [195-196], where several alternative arrays are described. The same basic algorithm is used for the description of a series of techniques for the derivation and mapping of recurrence equations on-to systolic arrays in [47], [184] and [238]. The mvm algorithm was the first to be considered for optical systolic implementation in [58]. In [19-20] the R+F method is applied in order to improve the efficiency of the mvm array. For the same reason the 'double pipe' concept is introduced in [200-201]. The partitioning of the coefficient matrix in triangular factors, so that a small array can accommodate bigger mvm problems is addressed in [232]. The same method is further elaborated in [215], whilst an alternative, more general approach is given in [209].

An important application of the mvm array in Numerical Analysis is concerned with the iterative solution of linear systems of equations. Some well known iterative methods are used for the solution of linear systems derived from the discrete approximation of ordinary and partial differential equations [282], [289], [300] (see also sections 2.3. and 2.5). These methods are based on a series of mvm operations, so that they can be readily implemented systolically as long as a mvm systolic array is available.

The systolic realisation of iterative methods has been discussed in [80], where a pipeline for the Jacobi method, for sparsely banded matrices, is described in detail. In [45] the Gauss-Seidel method is investigated. A more general approach is given in [25], where a number of iterative methods are considered. The optical implementation of the same algorithms is described in [51], while in [53] the concept is extended to non-linear systems of equations. Finally a hybrid optical-digital architecture is proposed in [2].

- 316 -

In section 6.2, the derivation of an improved systolic mvm array is given, while in the subsequent section the systolic pipelines for the iterative solution of linear systems are reviewed and some modified pipelines are proposed, partially based on the improved mvm array.

Some special techniques are discussed in sections 6.4 and 6.5. Firstly, the solution of tridiagonal sytems using Cyclic Reduction is considered. Then the same method is generalised and a Multi-Coloring scheme is also applied. In the final section of this chapter, some additional applications for the improved mvm array are briefly discussed. Furthermore, alternative methods for the systolic implementation of the iterative methods for solving linear systems are also considered.

6.2 IMPROVED SYSTOLIC MATRIX VECTOR MULTIPLICATION *

The linear systolic array for banded matrix-vector multuplication (mvm), originally proposed in [181], (see section 3.2), can be improved in the following aspects:

The fact that the data sequence is not compact leads to a sub-optimal computation time since the array requires 2n+w IPS cycles to complete its computation on a (nxn) matrix A with bandwidth w; the processor utilisation is 1/2, as well as its throughput; i.e. in general the efficiency of the array is 1/2.

The dataflow is bi-directional along the array, although there are no feedback cycles, i.e. none of the values of any data stream depends on the preceding values of the same data stream. This bi-directional dataflow complicates the application of fault-tolerance techniques; furthermore, drainage and fillup cycles are also necessary (see Fig. 6.2.1).

The interconnection of the array with other systolic arrays requires the reformulation of the data sequences: e.g. the banded matrix-matrix multiplication (mmm) array, also proposed in [181], (see section 3.2), has two dummy

^{*} A shortened version of this section has been presented as part of lectures in the Workshop on VLSI Computation, Univ. of Leeds, 16 Feb. 1987, and in the Parallel Architectures and Computer Vision Workshop, Univ. of Oxford, 30 March 1987.



Fig.6.2.1. Dataflow for the bidirectional mvm array.

elements between two successive data items and therefore the interconnection of a mvm and a mmm array requires either some intermediate processing of the data sequences or the slowing down of the mvm array.

A unidirectional dataflow combined with a compacted data sequence format can improve the systolic design in all the points discussed above. As is well known, there are alternative systolic designs for a given recurrence and the optimal design is determined in accordance to the constraints imposed by the specific application. These alternative systolic designs can be derived by means of more or less formal techniques (see chapters 1,3). Herein the 'retiming' technique is used for the derivation of the improved systolic array design, in a way similar to that described in section 4.3.

6.2.1 SYSTOLIC ARRAY DERIVATION

The mvm computation Ax=y, in the case of the example of Fig.6.2.1, can be written as $y_1 = 0.0 + a_{11}x_1 + a_{12}x_2 + a_{13}x_3$ $y_2 = a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + a_{24}x_4$ $y_3 = a_{32}x_2 + a_{33}x_3 + a_{34}x_4 + a_{35}x_5$ $y_4 = a_{43}x_3 + a_{44}x_4 + a_{45}x_5 + 0.0$ $y_5 = a_{54}x_4 + a_{55}x_5 + 0.0 + 0.0$ (6.2.1) This algorithm can be straighforwardly implemented as shown

in Fig.6.2.2 : in any cycle during the computation each cell

- 320 -







Fig.6.2.2. A simple systolic design for mvm.

- 321 -

performs a multiplication and all products are summed up by means of a systolic tree adder. The dataflow is also given in Fig.6.2.2 and can be readily derived from (6.2.1); \underline{x} moves to the left, while the diagonals of matrix A are accessed through vertical channels; the entries of \underline{y} are produced as an output of the tree adder one for each cycle. An initial delay is required for the fillup of the array and for the summation through the systolic tree-adder.

Now if the computations in each cell are delayed by d cycles, where d is the distance of a cell from the leftmost cell, then the computation for \underline{y} can be pipelined, as illustrated in Fig.6.2.3. The tree adder has been avoided and each cell performs a full IPS computation; no fillup delay is necessary; however, the elements of \underline{x} must be broadcast to all cells. The computation cycle is now 1 IPS, and the initial delay for the first result to be produced is equal to the bandwidth w of matrix A.

Finally, if the calculations are 'retimed' for an additional cycle then both \underline{x} and \underline{y} travel in the same direction but at different speeds, as shown in Fig.6.2.4. All long interconnection characteristics have been removed, and the initial delay is now equal to 2w-1; the only additional hardware required is w-1 delays in the data stream of \underline{y} . Notice the unidirectional dataflow and the compactness of the data sequences. Thus the processor utilisation is nearly 1 and no fillup and drainage cycles are required between

- 322 --





Fig.6.2.3. Dataflow for a semi-systolic mvm array.



Fig.6.2.4. Dataflow for unidirectional mvm (\underline{y} is delayed).

- 324 -
successive mvm computations.

The relation (6.2.1) can also be rewritten as a backward recurrence, i.e. the inner products are collected in the reverse order of that given in (6.2.1). If the data stream of \underline{x} is now delayed, then the resulting dataflow is given in Fig.6.2.5. The computation time is now n+w+p-1 IPS cycles, where n is the order of the matrix and p is the size of the upper semiband of matrix A, i.e. w=p+q-1. A systolic array implementing the dataflow of Fig.6.2.5 is shown in Fig.6.2.6 and a soft-systolic simulation program in OCCAM is given in A.3.1.

unidirectional mvm systolic designs The proposed herein allow for compact data sequences, and therefore optimal computation time; furthermore fault-tolerance techniques can be applied to yield a network that degrades gracefully with respect to the number of consecutive failed cells. They also allow for extensive pipelining, since all data streams move in the same direction and no drainage and fillup cycles are required; more importantly the direct interconnection of mvm and mmm systolic arrays is now easily implementable since the mmm hex-array proposed in [184], [295] has the same characteristics with the mvm arrays discussed herein.

- 325 -



Fig.6.2.5. Dataflow for unidirectional mvm (\underline{x} is delayed).

٠



Fig.6.2.6. Systolic array for unidirectional mvm.

- 327 -

6.3 <u>IMPROVED SYSTOLIC DESIGNS FOR THE ITERATIVE SOLUTION</u> OF LINEAR SYSTEMS *

Given a linear system of equations, $A\underline{x}=\underline{b}$, which, without loss of generality has been so ordered that $a_{ii}\neq 0$, for all i. The system can be rewritten as

$$\underline{\mathbf{x}} = \mathbf{M}\underline{\mathbf{x}} + \underline{\mathbf{g}} , \qquad (6.3.1)$$

where M is the Jacobi matrix of A, with

$$m_{ij} = \begin{cases} 0, & i=j \\ -a_{ij}/a_{ii}, & i\neq j \end{cases}, g_{i} = b_{i}/a_{ii}, & i=1,2,...,n.$$
(6.3.2)

Equations (6.3.1,2) are the basis of many iterative methods for the solution of linear systems discussed in section 2.3: Some of these methods are

(i) Jacobi method (J)

$$\underline{x}^{(k+1)} = M\underline{x}^{(k)} + \underline{g} , \qquad (6.3.3)$$

(ii) Jacobi overrelaxation method (JOR)

$$\underline{x}^{(k+1)} = \omega (\underline{Mx}^{(k)} + \underline{g}) + (1 - \omega) \underline{x}^{(k)} , \qquad (6.3.4)$$

where is ω the overrelaxation factor.

(iii) Gauss-Seidel method (GS) $\kappa^{(k+1)} = L\kappa^{(k+1)} + U\kappa^{(k)} + g$

(6.3.5)

^{*} A shortened version of this section has been presented as part of lectures in the Workshop on VLSI Computation, Univ. of Leeds, 16 Feb. 1987. This section is also part of a paper presented at the Numeta 87 conference, University of Wales, Swansea.

where L(U) is a strictly lower(upper) triangular matrices with L+U=M.

(iv) Successive overrelaxation method (SOR)

$$\underline{x}^{(k+1)} = \omega (\underline{Lx}^{(k+1)} + \underline{Ux}^{(k)} + \underline{g}) + (1 - \omega) \underline{x}^{(k)}$$
(6.3.6)

Methods (6.3.3-6) have been considered for systolic implementation in [25], [45], [80]; Fig.6.3.1-4 illustrate linear arrays performing one iteration of the methods (6.3.3-6). In the same figures the data sequence format is shown as well as the synchronisation delays that are necessary between two successive iterative steps. In general, all the designs can be analysed in : one (or two) mvm arrays and a special cell performing the computations described in Fig.6.3.5.

Table 6.3.1 summarises the area and time requirements for the iterative systolic designs, for k iterative steps, and for a (nxn) banded matrix A, with bandwidth w=p+q-1. Notice that the time complexity for the special cell in the case of SOR is 1 IPS + 1 ADD and therefore the time cycle for this method must be prolonged accordingly.

6.3.1 IMPROVED ITERATIVE SYSTOLIC DESIGNS

The following observations can be made on the pipeline designs of Fig.6.3.1-4. Firstly, the quantities

$$-\frac{a_{ij}}{a_{ii}}, -\omega \frac{a_{ij}}{a_{ii}}, i \neq j \text{ and } \frac{b_i}{a_{ii}}, \omega \frac{b_i}{a_{ii}}$$
(6.3.7)

are calculated repetitively in each step of the iterative



Fig.6.3.1. Pipeline block for J method; w=5, p=q=3.



Fig.6.3.2. Pipeline block for JOR method.

L



Fig.6.3.3. Pipeline block for GS method.

1



Fig.6.3.4. Pipeline block for SOR method.



 $xl_{in} \qquad xl_{out} \qquad$





xlout:=xlin x2out:=x2in/ain aout^{:=a}in

xlout:=(xlin+x2in+bin)/ain aout:=ain bout:=bin

xlout:=(xlin+x2in+bin)/ain+xwin
 (x:=x3in delayed 2(p-1) cycles)
aout:=ain
bout:=bin

Fig.6.3.5. Special cell definitions for pipeline blocks in Fig.6.3.1-4.

Method	J	JOR	GS	SOR
Area Time	k((W-1)IPS+DIV) 2(n-1)+(k-1)(2p+1) +(W+1)	k(wIPS+DIV) as for J	k((w-1)IPS+DIV+ADD) 2(n-1)+(k-1)2p+p	k(WIPS+DIV+ADD) as for GS

Table 6.3.1. Area-time requirements for systolic pipelines in Fig.6.3.1-5. process, while they need to be calulated only once in the beginning of the computation. Thus, the complex special cells will be avoided at the expense of some initial preprocessing. Furthermore, for the JOR and SOR methods the addition of vectors $(1-\omega)\underline{x}^{(k)}$ can be interpreted as an additional IPS in the mvm, i.e. (6.3.4,6) can be rewritten as

$$\underline{x}^{(k+1)} = (\omega L + (1-\omega) I + \omega U) \underline{x}^{(k)} + \omega \underline{g} , \qquad (6.3.8)$$

$$\underline{x}^{(k+1)} = \omega \underline{L} \underline{x}^{(k+1)} + ((1-\omega)\mathbf{I} + \omega \mathbf{U}) \underline{x}^{(k)} + \omega \underline{g} . \qquad (6.3.9)$$

Thus, a more uniform implementation of the four methods can be achieved since for $\omega=1$, JOR yields the J and SOR yields the GS methods. Fig.6.3.6-7 illustrate the improved systolic arrays for one iterative step: the linear array of Fig.6.3.6 is a simple mvm array implementing a matrix-vector inner product step (mvips) as in (6.3.1). In Fig.6.3.7 there are again two mvm arrays but now the middle cell is a three-input adder.

The preprocessing elements required for the formulation of the data sequences are shown in Fig.6.3.8,9. The main diagonal of matrix A enters a divider cell calculating ω/a_{ii} ; this quantity is propagated to the remaining cells while $(1-\omega)$ is the main diagonal entry of the output matrix M. The other cells calculate $-(\omega/a_{ii})a_{ij}$, except for the cell corresponding to <u>b</u> that only calculates $(\omega/a_{ii})b_i$. Some reformatting delays are necessary for the output of the preprocessing elements to conform with the data sequence



Fig.6.3.6. Modified pipeline block for J, JOR methods.



Fig.6.3.7. Modified pipeline block for GS, SOR methods.

- 338 -

formats of the iterative systolic designs; these delays are also shown in Fig.6.3.8,9.

Table 6.3.2 summarises the area and time requirements for the improved iterative systolic designs. By comparing Tables 6.3.1,2 we can observe that, for the J,JOR methods, the special cell in each iteration is replaced by a simple IPS cell, which, in the case of the J method can be replaced by simple delays. On the other hand, a new preprocessing array is introduced; thus, the main pipeline becomes simpler and more regular at the expense of w IPS cells and 1 divider of the preprocessor. The computation times do not differ significantly.

For the GS, SOR methods, again the main pipeline is simpler and for the GS method the main diagonal cell can be removed giving better area results. The computation times are nearly the same. In general, the improved designs offer simpler and more regular pipelines with approximately the same time complexity; furthermore some unification is achieved in the treatment of the different methods, i.e. the same systolic design can be used for both J and JOR, or for GS and SOR methods.

6.3.2 UNIDIRECTIONAL MVM ARRAY FOR J, JOR METHODS

The basic building block of the iterative systolic networks discussed is the mvm array, which can be improved as proposed in section 6.2. Thus, for the J,JOR methods the

Method	J,JOR	GS,SOR
Area Time	kwIPS+(w-1)IPS+DIV+MUL 2(n-1)+(k-1)(2p-1)+w+ (4q-2)	k (wIPS+ADD) + (w-1) IPS+DIV+MUL 2 (n-1) + (k-1) 2p+p+ (2q-1)

Table 6.3.2. Area-time requirements for systolic pipelines in Fig.6.3.6-9.

,



Fig.6.3.8. Preprocessor for J, JOR methods.



Fig.6.3.9. Preprocessor for GS, SOR methods.

342

L



Fig.6.3.10. Pipeline block for J, JOR methods using the unidirectional mvm array.

•

.





5

- 344 -

pipeline block can take the form shown in Fig.6.3.10, while the preprocessing element is given in Fig.6.3.11. Softsystolic simulation programs for the J,JOR pipeline and its preprocessor are given in A.3.2, A.3.3.

The area requirements are nearly the same; the only additional complexity is the line crossings that are necessary for M, g to be fed into the mvm array. The time requirements have changed significantly, since the data sequence is compact and only n, instead of 2n IPS, cycles required for the result to be produced; on the other are hand, however, the unidirectional mvm array imposes w+p-1 IPS cycles delay per pipeline block, instead of 2p-1 for the design of Fig.6.3.7. Therefore the gain in computation time can be expressed as n-k(w-p) IPS cycles; so that the gain is significant for n>>kw.

In the case of the GS,SOR methods the unidirectional array gives no better results than those achieved by the design of Fig.6.3.8. This is because the benefit of the compactness of the data sequence is cancelled by the need for idle cycles between the computation of two successive entries of \underline{x} . Some alternative ways for improving the performance of the systolic GS,SOR methods are discussed in the subsequent sections.

6.4 <u>SYSTOLIC NETWORKS</u> FOR ITERATIVE <u>MEHODS</u> <u>USING</u> <u>CYCLIC</u> REDUCTION

An important class of linear systems of equations are the systems obtained as the discrete finite difference approximation to general second order ordinary differential equations. A simple case is the equation (see section 2.5)

$$-\frac{d^2 x(z)}{dz^2} + \sigma(z) x(z) = \phi(z)$$
(6.4.1)

subject to boundary and continuity conditions given in [289],[300]. The discrete approximation vector \underline{x} is given by a system of the form $\underline{A\underline{x}=\underline{b}}$, where A is a tridiagonal, irreducible, diagonal dominant matrix, with

$$a_{ij} > 0, a_{ij} < 0, i \neq j, i = 1, 2, ..., n.$$
 (6.4.2)

The corresponding Jacobi matrix M is tridiagonal with zero main diagonal entries, and non-negative off-diagonal enties (see Fig.6.4.1). As shown in the same figure, A and M can be rewritten as

$$A = \begin{bmatrix} D_1 & A_1 \\ A_2 & D_2 \end{bmatrix} \qquad M = \begin{bmatrix} O & F_1 \\ F_2 & O \end{bmatrix} \qquad (6.4.3)$$

Thus, the Jacobi matrix produced by the coefficient matrix A of the discrete approximation to the solution of a general second order ordinary differential equation as in (6.4.1), can take the form of (6.4.3) and it is said to be a weakly cyclic matrix of index 2. Alternatively, A is defined as a

	1	2	3	4	
1	^a 11	^a 12			
2	a ₂₁	^a 22	a ₂₃		
3		a ₃₂	a 33	a 34	
4			^a 43	a ₄₄	







Fig.6.4.1(a). 2-cyclic ordering of tridiagonal matrices for n=4 (even) and n=5 (odd).



Fig.6.4.1(b). 2-cyclic ordering of the Jacobi matrices.

F₂

2-cyclic matrix relative to the partitioning of Fig.6.4.1.

Especially in the case of second order ordinary differential equation in (6.4.1), matrix A is symmetric, thus yielding submatrices $H_2=H_1^T$ and $F_2=F_1^T$. In general, F_1 is lower triangular bidiagonal, and F_2 is upper triangular bidiagonal matrix, of size ([(n+1)/2]x[(n+1)/2]); for n odd the addition of a dummy column (row) is necessary.

Consider the solution of the linear system of equations $A\underline{x}=\underline{b}$, using the iterative methods discussed in section 6.3. From (6.3.1) it is obvious that both \underline{x} and \underline{g} can be partitioned as

$$\begin{bmatrix} \underline{x}_1 \\ \underline{x}_2 \end{bmatrix} = \begin{bmatrix} 0 & F_1 \\ F_2 & 0 \end{bmatrix} \begin{bmatrix} \underline{x}_1 \\ \underline{x}_2 \end{bmatrix} + \begin{bmatrix} \underline{g}_1 \\ \underline{g}_2 \end{bmatrix}$$
(6.4.4)

Thus, method J (6.3.3) can be written as

$$\frac{x_{1}^{(k+1)} = F_{1} \underline{x}_{2}^{(k)} + \underline{g}_{1}}{\underline{x}_{2}^{(k+1)} = F_{2} \underline{x}_{1}^{(k)} + \underline{g}_{2}}$$
(6.4.5)

i.e. a pair of decoupled equations is created, where $\underline{x_1}^{(k+1)}$ depends on only $\underline{x_2}^{(k)}$ and vice versa:



Since $\underline{x}_1, \underline{x}_2$ have both $\lceil (n+1)/2 \rceil$ entries, the two systems can

be solved in parallel. The JOR method (6.3.4) can be suitably modified as follows :

$$\frac{x_{1}^{(k+1)} = \omega (F_{1} \frac{x_{2}^{(k)} + g_{1}) + (1 - \omega) \frac{x_{1}^{(k)}}{1}}{x_{2}^{(k+1)} = \omega (F_{2} \frac{x_{1}^{(k)} + g_{2}) + (1 - \omega) \frac{x_{2}^{(k)}}{2}}$$
(6.4.7)

In this case, $\underline{x}_1^{(k+1)}, \underline{x}_2^{(k+1)}$ depend on both $\underline{x}_1^{(k)}$ and $\underline{x}_2^{(k)}$.



Applying the GS method (6.3.5) the vector iterates are defined as

$$\frac{\mathbf{x}_{1}^{(k+1)} = \mathbf{F}_{1} \mathbf{x}_{2}^{(k)} + \mathbf{g}_{1}}{\mathbf{x}_{2}^{(k+1)} = \mathbf{F}_{2} \mathbf{x}_{1}^{(k+1)} + \mathbf{g}_{2}}$$
(6.4.9)

Alternatively, since

$$\begin{bmatrix} 0 & F_1 \\ F_2 & 0 \end{bmatrix}^2 = \begin{bmatrix} F_1 F_2 & 0 \\ 0 & F_2 F_1 \end{bmatrix}$$
(6.4.10)

the solution of (6.4.8) is equivalent to the solution of the uncoupled equations

$$\frac{\mathbf{x}_{1}^{(k+1)}}{\mathbf{x}_{2}^{(k+1)}} = \mathbf{F}_{1}\mathbf{F}_{2}\frac{\mathbf{x}_{1}^{(k)}}{\mathbf{x}_{2}} + (\mathbf{F}_{1}\underline{g}_{2}+\underline{g}_{1}) \\ = \mathbf{F}_{2}\mathbf{F}_{1}\frac{\mathbf{x}_{2}^{(k)}}{\mathbf{x}_{2}} + (\mathbf{F}_{2}\underline{g}_{1}+\underline{g}_{2})$$
(6.4.11)

Now $\underline{x_1}^{(k+1)}$ depends only on $\underline{x_1}^{(k)}$ and $\underline{x_2}^{(k+1)}$ on $\underline{x_2}^{(k)}$ (Cyclic Reduction):



Furthermore from (6.4.8) and (6.4.11) the following equivalent system can be formed:

$$\begin{array}{c} \underline{x}_{1} = F_{1} \underline{x}_{2} + \underline{g}_{1} \\ \underline{x}_{2} = F_{2} F_{1} \underline{x}_{2} + (F_{2} \underline{g}_{1} + \underline{g}_{2}) \end{array} \right\}$$
(6.4.13)

In this way only half of the iterative computation is performed, while \underline{x}_1 is calculated by means of the final solution vector for \underline{x}_2 . Therefore we have:



Thus, \underline{x}_2 can be computed as soon as F_2F_1 and $(F_2g_1+g_2)$ are formed. These two calculations can be combined by squaring the augmented matrix

$$\begin{bmatrix} 1 & \underline{o}^{\mathrm{T}} \\ \underline{o} & M \end{bmatrix}^{2} = \begin{bmatrix} 1 & \underline{o}^{\mathrm{T}} \\ \underline{g} + M \underline{g} & M^{2} \end{bmatrix}, \qquad (6.4.15)$$

with M^2 as in (6.4.10) and g+Mg analysed as

$$\begin{bmatrix} \underline{g}_1 \\ \underline{g}_2 \end{bmatrix} + \begin{bmatrix} \mathbf{0} & \mathbf{F}_1 \\ \mathbf{F}_2 & \mathbf{0} \end{bmatrix} \begin{bmatrix} \underline{g}_1 \\ \underline{g}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{F}_1 \underline{g}_2 + \underline{g}_1 \\ \mathbf{F}_2 \underline{g}_1 + \underline{g}_2 \end{bmatrix} .$$
(6.4.16)

For the SOR method (6.3.6) the system is described by

$$\frac{x_{1}^{(k+1)} = \omega (F_{1} x_{2}^{(k)} + g_{1}) + (1 - \omega) x_{1}^{(k)} }{x_{2}^{(k+1)} = \omega (F_{2} x_{1}^{(k+1)} + g_{2}) + (1 - \omega) x_{2}^{(k)} }$$
(6.4.17)

It is obvious that no significant gain is achieved by this partitioning, and furthermore the expression of $\underline{x_1^{(k+1)}}$ in terms of $\underline{x_1^{(k)}}$ and $\underline{x_2^{(k)}}$ introduces excessive complexity in the solution of the system.

6.4.1 SYSTOLIC DESIGNS

Fig.6.4.2-4 illustrate schematically pipeline stages that implement the recurrence relations derived from equations (6.3.3-5) when M is a weakly cyclic matrix of index 2. Fig.6.4.2 shows a systolic design for the realisation of the kth iteration of (6.4.5,6) for the case of matrix M as defined in Fig.6.4.1, and n=5. The case of n even is not discussed as it is covered adequately by the odd case. Two parallel pipelines are developed, where the output of stage k of one pipeline is the input of stage k+1 of the other pipeline. The basic component of the pipeline stages is the mvm array discussed in section 6.2. The two mvm arrays have area w_1, w_2 IPS cells, where w_1, w_2 are the bandwidths of F_1, F_2 respectively; as illustrated in Fig.6.4.2-4, where $w_1=2$, $w_2=2$, with $p_1=1$, $q_1=2$, and $p_2=2$, $q_2=1$.

The input data sequence formats and the synchronisation delays required are also shown in Fig.6.4.2. The output vec-



Fig.6.4.2. Pipeline block for J method (2-cyclic matrices).



•

Fig.6.4.3. Pipeline block for JOR method (2-cyclic matrices).

354 -

I.



Fig.6.4.4. Pipeline block for GS method (2-cyclic matrices).

355 I

1

tors are synchronized by adding the appropriate delays in the data stream of the 'faster' output, i.e. in the side with smaller (w_i+p_i-1) , i=1,2. Thus, both output streams encounter a delay of max $((w_1+p_1-1), (w_2+p_2-1))$ IPS cycles. The same number of delays is placed on the data stream of F_1, F_2 and g_1, g_2 .

For the case of the tridiagonal matrix A, the Jacobi matrix M has two significant diagonals. Using a systolic network as in Fig.6.3.10 for the solution of the non-cyclic system it will require pipeline stages of $w=w_1=w_2=2$ IPS cells and a computation time of the order n+kw IPS cycles, where k is the number of iterations. Using the systolic design of Fig.6.4.2 the solution will require pipeline stages of area 2w and computation time of order (n/2)+kw.

Fig.6.4.3 depicts the implementation of equations (6.4.7,8): again the pipeline stage can be analysed in two parallel pipelines. However, in this case stage k+1 of one k of both pipelines; pipeline accepts input from stage furthermore, the computation is complex since more $(1-\omega)\underline{x}_1$ or $(1-\omega)\underline{x}_2$ should be added to the result of the mvips operation. This is accomplished by means of an additional cell performing the calculation IPS $\omega \underline{q}_1 + (1-\omega) \underline{x}_1$ or $\omega \underline{q}_2 + (1-\omega) \underline{x}_2$.

The input data sequence format is the same as in Fig.6.4.2. Some delays must be added in the input streams: \underline{x}_1 has to be delayed by (p_1-1) cycles before the beginning

- 356 -

of the calculation of $g_1 + (1-\omega) \underline{x}_1$; similarly \underline{x}_2 is delayed by (p_2-1) cycles. Furthermore, the additional IPS cell imposes a one-cycle delay in all input streams that are not involved in its computation. Since the computation time for the pipeline block is increased by one cycle, the output delays for data streams F_1, F_2, g_1, g_2 must also be increased by 1 cycle.

Comparing the design of Fig.6.4.3 with the corresponding pipeline implementing the JOR method in section 6.3, it is observed that, as in the case of the J method, the area has been doubled. The interconnections in the pipeline of Fig.6.4.3 are more complex, mainly because the computation involving $(1-\omega)$ cannot be performed in the main mvm array as in Fig.6.3.7. The computation time is of order (n/2)+kw, instead of n+kw which is a significant gain, especially if n>>kw.

Fig.6.4.4 illustrates the realisation of equations (6.4.13,14): only \underline{x}_2 is produced, while \underline{x}_1 can be calculated separately using the final result for \underline{x}_2 . This design can be duplicated, so that two parallel and unconnected pipelines calculate \underline{x}_1 and \underline{x}_2 . The kth stage of this pipeline is a simple mvm array. For the tridiagonal matrix A, the bandwidth of F_1F_2 is equal to the bandwidth of the original matrix A, while matrix M has only two significant diagonals.

Comparing the design of Fig.6.4.4 with the systolic pipeline for the GS method discussed in section 6.3, it is noticed that the area has been increased by 1/2 and the time

has been reduced from approximately 2n+kw to (n/2)+kw. However only \underline{x}_2 is produced and some pre- and postprocessing are necessary to calculate F_1F_2 and $F_2g_1+g_2$. If the double, uncoupled pipeline is used, the area is three times more while the time reduction is the same.

Block diagrams illustrating the overview of linear system solvers based on the pipeline designs discussed are given in Fig.6.4.5. The preprocessor, common to all system solvers, is similar to that discussed in section 6.3, i.e for given $H_1, H_2, \underline{b}_1, \underline{b}_2$ and ω it produces F_1, F_2, g_1, g_2 . The designs are similar to that in Fig.6.3.11; what differs is the arrangement of the data streams and reformatting delays that the output sequence format of the postprocessor is so identical to that required by the next block of the system. designs 3 and 4, the next block is a combination of mmm For the calculations in performs that mym arrays and (6.4.15,16). The mmm array used is the unidirectional array in [184],[295]. Finally, for design 3, there is a postprocessing element performing a mvips computation. Softsystolic simulation programs in OCCAM, for the pipeline designs of Fig.6.4.2 and Fig.6.4.3 are given in A.3.4 and A.3.5, respectively.



Fig.6.4.5. Overall pipeline configuration for the iterative solution of 2-cyclic systems.

- 359 -

6.5 P-CYCLIC MATRICES AND MULTI-COLORING TECHNIQUES

6.5.1 P-CYCLIC MATRICES

The results of the previous section are now extended in the more general case of p-cyclic matrices. A (nxn) matrix A is defined to be p-cyclic, where $p \ge 2$, if by a permutation of rows and columns it can be partitioned into the form:



where the subdiagonal matrices A_{ii} are non-singular, square matrices and the off-diagonal submatrices A_{ij} are rectangular matrices. The Jacobi matrix corresponding to A has the form



(6.5.2)

where

$$M_{1,p} = -A_{11}^{-1}A_{1,p}$$

$$M_{r,r-1} = -A_{rr}^{-1}A_{r,r-1}, \quad r=2,...,p$$
(6.5.3)
Matrix M is said to be weakly cyclic of index p; if all A_{ii} are diagonal matrices then (6.5.3) can be rewritten as:

$$m_{ij} = -a_{ij}/a_{ii}$$
 (6.5.4)

As discussed in section 6.4, 2-cyclic matrices arise naturally in the discrete approximation by finite differences of elliptic or parabolic differential equations. Examples of problems producing p-cyclic matrices for p>2 can be found in [280],[282] and can be classified in two general categories. Firstly, the use of 4-point stencils on triangular or rectangular nets for the finite-difference approximation of elliptic differential equations (see Fig.6.5.1) yield pcyclic matrices for 'p=3 or 4. Then, finite-difference approximation of parabolic differential equations with periodic boundary conditions yield p-cyclic matrices with p arbitrarily large.

It is assumed that A_{ii} are diagonal matrices and therefore the elements of the corresponding Jacobi matrix is given by (6.5.4). The bandwidth of the submatrices A_{ij} is (s-1) where s is the number of points that are covered by the stencil, while the bandwidth of matrix A in natural ordering is s, if the null diagonals are not taken into account.

The linear system of equations Ax=b, for A a p-cyclic matrix, takes the form:



p=2, s=3



p=2, s=5

Fig.6.5.1(a). 2-cyclic ordering using 3 and 5 -point stencils.

- 362 -



p=3, s=4





p=4, s=4

Fig.6.5.1(b). 3 and 4 -cyclic ordering using 4-point stencils.



and can be rewritten as,



where m_{ij} is given by (6.5.4) and $g_i = b_i/a_{ii}$. Thus, the J method (6.3.3) is partitioned as

$$\frac{x_{1}^{(k+1)}}{x_{1}} = M_{1,p} \cdot \frac{x_{p}^{(k)}}{p} + \frac{g_{1}}{g_{1}}$$

$$\frac{x_{i}^{(k+1)}}{x_{i}} = M_{i,i-1} \cdot \frac{x_{i-1}^{(k)}}{q_{i}} + \frac{g_{i}}{q_{i}}, i=2,...,p$$
(6.5.7)

Similarly, the JOR method (6.3.4) has the form

$$\frac{x_{1}^{(k+1)}}{x_{1}^{(k+1)}} = \omega (M_{1,p} \cdot \frac{x_{p}^{(k)}}{p} + \frac{g_{1}}{p}) + (1 - \omega) \frac{x_{1}^{(k)}}{x_{1}},$$

$$\frac{x_{i}^{(k+1)}}{x_{i}^{(k)}} = \omega (M_{i,i-1} \cdot \frac{x_{i-1}^{(k)}}{x_{i-1}^{(k)}} + \frac{g_{i}}{p}) + (1 - \omega) \frac{x_{i}^{(k)}}{i}, \quad i=2,...,p \quad (6.5.8)$$

The GS method (6.3.5) is defined as

$$\frac{x_{1}^{(k+1)}}{x_{1}^{(k+1)}} = M_{1,p-p} + \frac{g_{1}}{p}$$

$$\frac{x_{1}^{(k+1)}}{x_{1}^{(k+1)}} = M_{1,1-1} + \frac{g_{1}}{p}, \quad i=2,\dots,p.$$
(6.5.9)

Finally the SOR method (6.3.6) is modified as follows

$$\frac{x_{1}^{(k+1)}}{x_{1}} = \omega (M_{1,p} + g_{1}) + (1 - \omega) x_{1}^{(k)}$$

$$\frac{x_{1}^{(k+1)}}{x_{1}} = \omega (M_{1,1} + g_{1}) + (1 - \omega) x_{1}^{(k)}, i = 2, \dots, p \quad (6.5.10)$$

Computational networks for the four methods are given in Fig.6.5.2,3. For the J,JOR methods the subvectors $\frac{x_1^{(k+1)}, \frac{x_2^{(k+1)}}{2}, \ldots, \frac{x_p^{(k+1)}}{2}$ can be computed in parallel, at the expense of additional area and interconnection complexity, in comparison to the computational networks for the same methods for a naturally ordered matrix A (see section 6.3). For the GS,SOR methods the same subvectors are computed in a pipelined fashion, but not in parallel.

The computation performed by each of the nodes in Fig.6.5.2,3 is the same with that performed by a pipeline stage discussed in section 6.4, for the 2-cyclic case. Table 6.5.1 gives a comparison on the order of area and time requirements for the cyclic and the natural case.

An alternative approach is to use the Frobenius' Theorem which states that if M is a weakly cyclic matrix of index p then M^p is completely reducible, i.e. it has the form



(6.5.11)



Fig.6.5.2(a). Systolic network for J method (p-cyclic matrices).



Fig.6.5.2(b). Systolic network for JOR method (p-cyclic matrices).

- 367 -







Fig.6.5.3(b). Systolic network for SOR method (p-cyclic matrices).

Methods		J,JOR	GS,SOR					
No ma	Area ≆	ks						
NOTIMAL	Time ≅	n+k(s+p _A)	2n+2kp _A					
Ovelie	Area ≆)	(ps					
cycric	Time ¥	$\frac{n}{p} + k(s + p_A)$	$\frac{n}{p} + kp(s+p_A)$					

.

Table 6.5.1. Comparison of the area-time requirements of the pipelines for normal and p-cyclic ordering.

- 368 -

- 369 -

where

$$C_{1} = M_{1,p} \qquad M_{p,p-1} \qquad M_{p-1,p-2} \qquad \cdots \qquad M_{2,1}$$

$$C_{2} = M_{2,1} \qquad M_{1,p} \qquad M_{p,p-1} \qquad \cdots \qquad M_{3,2}$$

$$\cdots$$

$$C_{p} = M_{p,p-1} \qquad M_{p-1,p-2} \qquad M_{p-2,p-3} \qquad \cdots \qquad M_{1,p}$$

$$Thus, equation (6.5.6) is equivalent to$$

$$\left[\underline{x}_{1}\right] \qquad \boxed{c}_{1} \qquad \qquad \boxed{x}_{1} \qquad \boxed{h}_{1}$$



Notice that \underline{h} can be computed simultaneously with M^p if the calculation



(6.5.14)

is performed instead of M^P. Thus,

$$\underline{h}_{1} = M_{1,p}(M_{p,p-1}(\dots (M_{32}\underline{g}_{2} + \underline{g}_{3}) + \dots) + \underline{g}_{p}) + \underline{g}_{1}$$

$$\underline{h}_{2} = M_{2,1}(M_{1,p}(\dots (M_{43}\underline{g}_{3} + \underline{g}_{4}) + \dots) + \underline{g}_{1}) + \underline{g}_{2}$$

$$\underline{h}_{p} = M_{p,p-1}(M_{p-1,p-2}(\dots (M_{21}\underline{g}_{1} + \underline{g}_{2}) + \dots) + \underline{g}_{p-1}) + \underline{g}_{p}$$

$$(6.5.15)$$

The linear system can then be solved by using (6.5.11) by means of p uncoupled parallel pipelines similar to those of

Fig.6.4.4. Notice, however that the bandwidth of C_i is now of order ps, leading to increased area requirements and increased delays per pipeline block.

Alternatively, a combination of (6.5.6) and (6.5.11) is possible, i.e. only C_p and \underline{h}_p are calculated and the system

$$\frac{x_{p}}{p} = C_{x} + \frac{h}{p}$$
(6.5.16)

is solved iteratively, using a single pipeline. Then, using (6.5.7) we have

$$\frac{x_{1}}{x_{1}} = \frac{M_{1,p-p}}{m_{i,i-1}} + \frac{g_{1}}{g_{i}}, \quad i=2,\dots,p-1.$$
(6.5.17)

where \underline{x}_p is the final solution obtained by (6.5.16). Thus, a series of (p-1) mvips is enough for the calculation of the whole solution vector; these calculations can be pipelined since \underline{x}_{i-1} is used for the computation of \underline{x}_i , for $i=2,3,\ldots,p-1$.

6.5.2 MULTI-COLORING TECHNIQUES

In [41] it is shown that for any partial differential equation defined on a simple geometry, it is possible to find a r-color ordering of the discrete points on which the GS iteration of the finite difference approximation system $A\underline{x}=\underline{b}$ is solved in r J method steps; a similar technique can be used for the SOR method. For example, the Red-Black ordering, as indicated in Fig.6.5.4 leads to the follwing partitioned matrix iterative form

$$\begin{bmatrix} \mathbf{x}_{R} \\ \mathbf{x}_{B} \\ \vdots \end{bmatrix} = \begin{bmatrix} \mathbf{0} & \mathbf{M}_{RB} \\ \mathbf{M}_{BR} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{x}_{R} \\ \mathbf{x}_{B} \\ \vdots \end{bmatrix} + \begin{bmatrix} \mathbf{g}_{R} \\ \mathbf{g}_{B} \end{bmatrix}$$

(6.5.18)

which has been already examined as the 2-cyclic case in section 6.4. The GS iteration is written as

$$\frac{x_{R}^{(k+1)}}{x_{B}^{k}} = M_{RB} \frac{x_{B}^{(k)}}{x_{B}^{k}} + \frac{g_{R}}{g_{R}}$$

$$\frac{x_{B}^{(k+1)}}{x_{B}^{k}} = M_{BR} \frac{x_{R}^{(k+1)}}{x_{R}^{k}} + \frac{g_{B}}{g_{B}}$$
(6.5.19)

Similarly, suitable coloring patterns for 7-point and 9point stencils are given in Fig.6.5.5,6. The associated linear systems are as follows :

$$\begin{bmatrix} \mathbf{x}_{R} \\ \mathbf{x}_{B} \\ \mathbf{x}_{G} \end{bmatrix} = \begin{bmatrix} \mathbf{0} & \mathbf{M}_{RB} & \mathbf{M}_{RG} \\ \mathbf{M}_{BR} & \mathbf{0} & \mathbf{M}_{BG} \\ \mathbf{M}_{GR} & \mathbf{M}_{GB} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{x}_{R} \\ \mathbf{x}_{R} \\ \mathbf{x}_{B} \\ \mathbf{x}_{G} \end{bmatrix} + \begin{bmatrix} \mathbf{g}_{R} \\ \mathbf{g}_{B} \\ \mathbf{x}_{G} \\ \mathbf{g}_{G} \end{bmatrix}$$
(6.5.20)

and

$$\begin{bmatrix} \underline{x}_{R} \\ \underline{x}_{B} \\ \underline{x}_{G} \\ \underline{x}_{O} \end{bmatrix} = \begin{bmatrix} \overline{0} & M_{RB} & M_{RG} & M_{RO} \\ M_{BR} & 0 & M_{BG} & M_{BO} \\ M_{GR} & M_{GB} & 0 & M_{GO} \\ M_{OR} & M_{OB} & M_{OG} & 0 \end{bmatrix} \begin{bmatrix} \underline{x}_{R} \\ \underline{x}_{B} \\ \underline{x}_{G} \\ \underline{x}_{O} \end{bmatrix} + \begin{bmatrix} \underline{g}_{R} \\ \underline{g}_{B} \\ \underline{x}_{G} \\ \underline{x}_{O} \end{bmatrix} (6.5.21)$$

yielding 3-step and 4-step J iterations for the GS method :

$$\frac{\mathbf{x}_{R}^{(k+1)}}{\mathbf{x}_{R}} = M_{RB} \frac{\mathbf{x}_{B}^{(k)}}{\mathbf{x}_{B}} + M_{RG} \frac{\mathbf{x}_{G}^{(k)}}{\mathbf{x}_{G}} + \frac{g_{R}}{\mathbf{x}_{G}}$$

$$\frac{\mathbf{x}_{B}^{(k+1)}}{\mathbf{x}_{B}} = M_{BR} \frac{\mathbf{x}_{R}^{(k+1)}}{\mathbf{x}_{R}} + M_{BG} \frac{\mathbf{x}_{G}^{(k)}}{\mathbf{x}_{G}} + \frac{g_{B}}{\mathbf{x}_{B}}$$

$$\frac{\mathbf{x}_{G}^{(k+1)}}{\mathbf{x}_{G}} = M_{GR} \frac{\mathbf{x}_{R}^{(k+1)}}{\mathbf{x}_{R}} + M_{GB} \frac{\mathbf{x}_{B}^{(k+1)}}{\mathbf{x}_{B}} + \frac{g_{G}}{\mathbf{x}_{B}}$$
(6.5.22)

and

$$\frac{x_{R}^{(k+1)}}{x_{R}^{(k+1)}} = M_{RB} \frac{x_{B}^{(k)}}{x_{B}^{(k+1)}} + M_{RG} \frac{x_{G}^{(k)}}{x_{G}^{(k)}} + M_{RO} \frac{x_{O}^{(k)}}{x_{O}^{(k)}} + \frac{g_{R}}{g_{R}^{(k+1)}}$$

$$\frac{x_{G}^{(k+1)}}{x_{G}^{(k+1)}} = M_{GR} \frac{x_{R}^{(k+1)}}{x_{R}^{(k+1)}} + M_{GB} \frac{x_{G}^{(k+1)}}{g_{R}^{(k+1)}} + M_{GO} \frac{x_{O}^{(k)}}{x_{O}^{(k)}} + \frac{g_{G}}{g_{G}^{(k+1)}}$$

$$\frac{x_{O}^{(k+1)}}{x_{O}^{(k+1)}} = M_{OR} \frac{x_{R}^{(k+1)}}{g_{R}^{(k+1)}} + M_{OB} \frac{x_{G}^{(k+1)}}{g_{R}^{(k+1)}} + M_{OG} \frac{x_{G}^{(k+1)}}{g_{G}^{(k+1)}} + \frac{g_{O}}{g_{G}^{(k+1)}}$$
(6.5.23)



Fig.6.5.4. 2-color ordering using 5-point stencil.



		R	В	G	В	G	R	G	R	В			R	R	R	В	В	B	G	G	G
		1	2	3	4	5	6	7	8	9			1	6	8	2	4	9	3	5	7
R	1	х	х		х	X					R	1	X]	х	X			х	
В	2	Х	х	х		х	X				R	6		Х		х		Х	х	Х	
G	3		х	X			х				R	8			X		X	X		х	X
В	4	x			х	х		х	х		В	2	X	X		X			X	X	
G	5	х	х		х	Х	х		х	X	В	4	x		X		х	1		х	X
R	6		х	х		х	X			х	В	9		X	X			X		X	
G	7				х			х	х		G	- 3	_	X		X			X		-
R	8				х	х		х	х	X	G	5	X	х	x	х	х	X		X	
B	9					x	x		x	x	G	7			x		x				x

Fig.6.5.5. 3-color ordering using 7-point stencil.



		R 1	В 2	G 3	0 4	G 5	0 6	R 7	B 8	R 9	В 1(G 011	0 112			R 1	R 7	R 9	В 2	B 8	В 1:	G 03	G 5	G 11	0	0 6	0 12
R	1	x	x	•	-	x	x	•	•	-	-			R	1	x	•		x	-			x	1		x	
В	2	x	x	х		x	х	x						R	7		х		x	х	x	х		x	x	x	x
G	3		x	х	x		х	х	x					R	9			x			x		x			X	
0	4			х	х			х	х					B	2	X	X		X			X	X			X	
G	5	x	х			х	х			х	X			В	8		х			х		Х		X	х		x
0	6	х	х	X		Х	х	х		х	х	х		В	10		<u>_X</u>	X			X		X	Х		X	
R	7		х	х	х		х	х	х		х	х	Х	G	3		X		X	X		X			х	X	
В	8			х	х			х	х			Х	X	G	5	X		×	X		X		х			х	
R	9					х	X			х	х			G	11		Х			X	X			X		X	X
В	1()				х	х	х		х	х	х		0	4		Х			Х		х			X		_
G	11	L					х	х	X		х	x	х	0	6	х	х	X	×		X	X	х	X		x	
0	12	2,						х	Х			х	x	0	12	•	X	·		х				X			x

Fig.6.5.6. 4-color ordering using 9-point stencil.

- 373 -

The convergence rates of the multi-coloring techniques are discussed in [3], [41],[220]. The parallelism that can be achieved by using a multi-colored ordering instead of natural ordering techniques is explained with the help of Fig.6.5.7 where simplified computational networks corresponding to equations (6.5.19,22,23) are given. Each block corresponds to a mvm array performing a mvips computation. No synchronisation delays between pipeline stages are shown and the communication channels for each pipeline have been unified, i.e. (r-1) submatrices are carried along each of the r parallel pipelines together with the corresponding subvector.

For r=2, no significant additional parallelism is obtained: $\underline{x}_{R}^{(k+1)}$ must be produced in order for $\underline{x}_{B}^{(k+1)}$ to be computed. The computation is similar to the GS method for the natural ordering in section 6.3, and to the 2-cyclic case discussed in section 6.4. However, for r=3 and 4 more significant additional parallelism is achieved, since the computation of the subvectors can be overlapped. As soon as a subvector is produced it takes part in the calculation of (r-1) subvectors; thus (r-1) subvectors are computed in parallel at any instant of the computation.

In general, each pipeline stage consists of r(r-1) mvm arrays; as soon as one element of a subvector is produced it is fed to the mvm arrays that calculate other subvectors in

- 374 -







(a) r=2



Fig.6.5.7. Systolic networks for r-color ordering.

the order indicated in Fig.6.5.7.At the final pipeline stage the r solution subvectors can be produced in parallel in a skewed fashion and a delay of w'+p'-1 IPS cycles between two successive subvectors, where w'=p'+q'-1 is the bandwidth of time is total computation The Μ.. submatrices the ((n/r)-1)+(r-1)(w'+p'-1) IPS cycles and the area required is r(r-1)w' IPS cells. Table 6.5.2 summarises a comparison of the area and time requirements of a systolic implementation GS method using natural and multi-color ordering, of the for the matrices in Fig.6.5.4-6.

Method	GS	r=2	GS	r=3	GS	r=4
Area	5kIPS+kADD	8kips	7kips+kadd	12KIPS	9kips+kadd	36kIPS
Time	2(n-1)+(k-1)8+5	$(\frac{n}{2}-1)+(k-1)$ 11+10	2(n-1)+(k-1)10+6	$(\frac{n}{3}-1)+(k-1)12+11$	2(n-1)+(k-1)12+7	$(\frac{n}{4}-1)+(k-1)16+15$

Table 6.5.2. Comparison of the area-time requirements of the pipelines for natural and r-colored ordering.

L

6.6 CONCLUSIONS

The close similarity of the mvm algorithm with the convolution and FIR filtering leads to the derivation of similar systolic arrays, as shown, for example, in [160], [184]. Notice also, the close similarity of systolic mvm with the Root-Squaring method described in section 4.3, and the polynomial multiplication algorithm discussed in section 3.1. The systolic mvm array proposed herein can be seen as a generalisation of all these systolic designs. A similar array is proposed in [202] for an irregular wavefront, data-driven mvm algorithm. The same array, as that proposed in section 6.2, is used as an introductory example in [203], but no derivation technique or further applications are presented.

The unidirectional dataflow of the improved mvm array allows for direct efficient mapping of the algorithm onto the CMU Warp Machine (see section 3.4) [162]: a softsystolic simulation program is given in A.3.6. Another possible application is the multiplication of a full matrix with a banded matrix, using a stack of mvm arrays as shown in Fig.6.6.1 [169]. The same stack arrangement can be used for an alternative realisation of the bit-level mvm array in [195], and the same concept can be extended in bit-level convolution and matrix-matrix multiplication [197]. Finally, it would be interesting to investigate the application of the problem partitioning techniques of [209], [215] in the new mvm array.



Fig.6.6.1. Banded-full mmm systolic array.

- 379 -

The pipelines for the J,JOR methods may be improved using the R+F method as applied on the mvm array. Similarly, a step of the GS,SOR methods can be seen as a combination of a mvips followed by a triangular system solution [252]; thus, the R+F method can be used again.

An alternative to the pipeline approach for the implementation of iterative methods is the use of a single pipeline block with feedback mechanism and adequate memory similar to that described in all optical systolic applications and in [209], (see Fig.6.6.2). The area requirements are minimised but no pipelining of several problems can be achieved (see area-time expansion in section 3.4). Instead of pipelining, there exists the possibility of solving a number of problems in parallel as independent iterative systolic designs; an i/o interface should then distribute the problems and collect the results. In [232-233] the time expansion scheme is combined with the partitioning of the coefficient matrix in triangular factors to achieve a more efficient implementation of GS, SOR methods, but for full matrices only.

Another iterative mvm array, that avoids the explicit use of feedback loops, can be derived as a single column version of the re-usable matrix multiplication array proposed in [237], (see Fig.6.6.3). The array performing the computation $A\underline{x}=\underline{y}$, where A is a full (nxn) matrix, consists of n IPS processors and a multiplexer cell. Each one of the

- 380 -



(a) Time-expansion



 $\frac{x}{k} = \frac{Ax}{k-1}$

k=1,2,

• •

(b) Area-expansion

Fig.6.6.2. Time and area expansion for mvm computation.

- 381 -



Fig.6.6.3. Re-usable mvm array.

- 382 -

IPS processors has a memory module of n words, similar to that of the CMU systolic processor in A.3.4. Each processor accumulates the inner products for an element of the resulting vector χ ; as soon as the accumulation is completed the newly formed element is sent towards the multiplexer. Thus, a new mvm can start exactly on time after the completion of the previous computation.

The same array can be used for the mvips operation, $A\underline{x}+\underline{y}$, if the multiplexer is augmented with a memory module holding vector \underline{y} , and an adder. Thus, the mvips computation for a dense (nxn) matrix can be performed in 2n+1 IPS cycles, and the area required is n+1 IPS cells, if we assume that the multiplexing-adding calculation has area and time complexity of 1 IPS. The feedback control complexity within the cells is not taken into account; the iterative design require a total memory of n(n+1) words. A soft-systolic simulation program in OCCAM is given in A.3.7.

The stack arrangement of Fig.6.6.1 can be used for the iterative solution of matrix equations, or for the parallel solution of a set of linear systems using the same coefficient matrix. The same stack organisation can be applied on the iterative design of Fig.6.6.3. Another method for the iterative solution of linear equations, using successive matrix squarings is discussed in the next chapter.

The systolic pipelines described in sections 6.4, 6.5 introduce an additional level of complexity, i.e the

- 383 -

interconnection and concurrent operation of parallel pipelines. These parallel interconnected pipelines, whose blocks consist of systolic arrays are termed herein as 'systolic networks'. A further extension can be the three-dimensional systolic networks to aleviate the interconnection problems between the pipelines.

Similar systems can be produced by using the iterative array of Fig.6.6.3. In this case the systolic networks will have a single block, with the output being fed back into the input, and the coefficient matrices and r.h.s vectors stored into appropriate memory modules. Thus, we can have a set of co-operating time-expanded systolic structures, that perform their computations in parallel, and exchange data using inter-array communication lines. The development of these systems is straightforward, using the designs in Fig.6.4.2-6.4.4, Fig.6.5.2-6.5.3 and Fig.6.5.7.

An extension of the Cyclic Reduction technique is the systolic implementation of the Recursive Doubling algorithm discussed in [210], [214]. A possible realisation can be achieved by using the VLSI arithmetic modules proposed in [137-138].

CHAPTER 7

SYSTOLIC ALGORITHMS USING MATRIX POWERS

7.1 INTRODUCTION

The importance of matrix-matrix multiplication (mmm) in numerical computation is well known, and the systolic implementation of this operation was amongst the first to be considered in [181], [199] (see also section 3.2). This systolic algorithm was mainly designed for banded matrices; another method, for full matrices, is introduced, amongst others, in [173] for the wavefront processor array, and in [273] for the engagement processor array.

Systolic mmm has attracted the attention of numerous researchers, in an effort to improve the performance of the original systolic designs or to produce efficient algorithms for specific applications. For example, in [19], [21] the R+F method is applied to improve the performance of the banded mmm. In [287] two alternative designs are surveyed: one based on a stack of mvm arrays, as discussed in section 6.6, and another having one of the coefficient matrices preloaded into the array. The relationship between wordlevel and bit-level mmm arrays is investigated in [196-197], and bit level arrays for signal processing applications are presented.

Since a mmm can be analysed as a series of mvm's, the mmm algorithm can be mapped onto a linear array, as shown in [158]. A two-level pipelined approach for the same method is given in [161-162]. A general mapping methodology of mmm and mvm algorithms on linear arrays is given in [238]. The block partitioning of the systolic mmm is discussed in [137-138]. A different approach, based on triangular factors partitioning, is proposed in [215]. Algorithm-level fault-tolerance mmm is addressed in [134], [153]. The fault-tolerance on the systolic array level is discussed in [164] using local correctness criteria and the cut theorem, on a unidirectional mmm hex-array.

The unidirectional hex-array, has been derived in [295] as a result of the application of a transformation technique for systolic algorithms. Similar results, extended to full matrix computations are reported in [184], where another mapping methodology is exemplified. In [169] systolic arrays for full-banded mmm are derived, and in [244] the divideand-conquer approach is applied on mmm. Two very efficient systolic arrays are proposed in [204-205], but their data sequence formats and interconnection patterns are rather complex.

Finally, the optical systolic implementation of mmm has

- 386 -

been extensively discussed, using either purely one dimensional architectures, [50]; or two dimensional, as for example in [29]; or a combination of the two approaches in [12], [14].

Many important problems in matrix iterative methods for the solution of linear systems of equations, matrix inversion and eigenvalue-eigenvector computations can be improved if the same methods are applied as A^{2^S} instead of A, where A is the original matrix involved in the calculations and s is the number of successive squarings of matrix A [32], [298]. Therefore, a systolic design performing a certain number of successive matrix squarings can be attached as a preprocessing element in several other systolic systems implementing iterative matrix calculations.

Furthermore, matrix polynomial computations are widely used in digital automatic control and other areas, where the calculation of matrix functions is necessary, [117]. Both successive matrix squarings and matrix polynomial computations are based on a series of mmm's. The systolic implementation of these computations is discussed in this chapter.

In section 7.2 the basic computation of successive matrix squares is investigated and pipeline and iterative systolic designs are proposed. Given the number of different systolic mmm arrays, it is evident that a similar number of matrix squaring designs can be derived. Herein only a limited number of designs is described in detail, i.e. those that seemed to be more suitable for the applications in question.

Some applications of successive matrix squaring are introduced in the following two sections. In section 7.3 the iterative solution of linear systems of equations, using matrix powers is discussed, while in section 7.4 the Power, Matrix Squaring and Raised Power Methods are investigated.

The systolic computation of the exponential of a matrix is addressed in section 7.5, and the systolic evaluation of a polynomial of a matrix is also discussed. Finally, section 7.6, proposes some methods for matrix inversion and evaluation of matrix power series. Furthermore the systolic calculation of several matrix functions is investigated.

7.2. SYSTOLIC DESIGNS FOR SUCCESSIVE MATRIX SQUARING*

The matrix squaring operation can be seen as a simple matrix-matrix multiplication (mmm), i.e. :

$$A^{2} = A * A; A^{4} = A^{2} * A^{2}; A^{8} = A^{4} * A^{4}$$
 (7.2.1)

Consequently one of the available systolic arrays for mmm suffices for its realisation. Following the definitions introduced in sections 3.3 and 6.6 (see also [151]), an iterative process can be expanded either in time, yielding an iterative array, or in area, producing a pipeline. Fig.7.2.1 outlines these concepts for the case of the matrix squaring computation.

7.2.1 SYSTOLIC PIPELINE DESIGNS

Firstly we concentrate on banded matrices. In order to achieve high pipelineability the dataflow of the mmm array should be unidirectional; in this way the output of a matrix-squaring array can immediately form the input of the next matrix-squaring stage without any re-routing of the data sequence. A systolic array satisfying the above mentioned requirements is the hex-connected array with unidirectional dataflow described in [184], [295]. The array and the data sequence are illustrated in Fig.7.2.2(a), for the case of the multiplication of two tridiagonal matrices

* This section is part of a paper to be published in Parallel Computing.



 $A_{k}=A_{k-1}^{2}, k=1,2,...$





.



Fig.7.2.2(a). Banded matrix multiplication on unidirectional hex-array.

ŀ



Fig.7.2.2(b). Banded matrix squaring on a unidirectional hex-array $w_A^{=3}$, $p_A^{=}q_A^{=2}$.

A

(w=3) of size n=5. The array consists of w^2 IPS cells and the computation time is n+w.

It is observed that in the case of matrix squaring, also shown in Fig.7.2.2, the input sequence of the array consists of two copies of matrix A that differ only in the position of the dummy elements. Notice also that, for $n \gg_A$, and w_A, w_A^2 the bandwidths of matrices A, A^2 we have $w_A^2 = 2w_A^{-1}$, and in general

$$W_{A^{2i}} = 2^{i}W_{A^{-}}(2^{i}-1), i=0,1,...,s$$
 (7.2.2)

This also means that in order to calculate $A^{2^{S}}$ an array of $w_{A^{2}}^{2}(s-1)$ IPS cells is required. It is obvious then that successive squarings broaden the profile of a banded matrix and therefore only a limited number of squaring steps should be performed if the matrix profile is to be kept within reasonable size. Since the matrix-squaring design is to be used as a preprocessor, the alteration of the matrix profile affects the size of the main systolic processor, if we assume that it is bandwidth-dependent. On the other hand, however, a considerable speed-up of the computation will be achieved: consequently, an area-time tradeoff analysis can indicate the desirable number of successive squarings. In general, if the maximum available area for a single mmm is w_{max}^{2} , then the maximum number of squaring steps s is given by the relation

$$2^{s-1}w_{A}^{-}(2^{s-1}-1) \leq w_{max}$$
 (7.2.3)

The squaring of a matrix with unequal semibands is illustrated in Fig.7.2.3, for $w_A=4$, $p_A=3$ and $q_A=2$. If we add $|p_A-q_A|$ dummy diagonals in the side with the min (p_A,q_A) then the matrix is transformed to one with equal semibands. For the systolic realisation of the dummy diagonals we can add $|p_A-q_A|$ delays in the appropriate input stream; the same amount of delay is added in the output sequence so that the coefficients of the output matrix are received in the same format as in the symmetrical case.Relation (7.2.2) is written as

$$\begin{array}{l} {}^{w}_{A}^{2i} = {}^{p}_{A}^{2i} + {}^{q}_{A}^{2i} - 1, i=0,1,\dots,s \\ \text{with} \\ {}^{p}_{A}^{2i} = {}^{2i}_{P}_{A}^{-}(2^{i}-1) \\ {}^{q}_{A}^{2i} = {}^{2i}_{Q}_{A}^{-}(2^{i}-1) \end{array} \right\} i=0,1,\dots,s$$

$$\begin{array}{l} (7.2.4) \\ i=0,1,\dots,s \\ (7.2.4) \\ i=0$$

A Thus, for the computation of $A^{2^{S}}$ the additional delays would be $2^{(s-1)}|p_{A}-q_{A}|$. The computation time for the array of Fig.7.2.3 is $n+w'_{A}$, where $w'_{A} = 2\max(p_{A},q_{A})-1$, and the array consists of w_{A}^{2} IPS cells. The total computation time for the calculation of $A^{2^{S}}$ in both Fig.7.2.2,3 is

$$n + \sum_{i=0}^{s-1} \left(\left(2^{i} w'_{A}^{-} \left(2^{i} - 1 \right) \right) \right)$$
(7.2.5)

IPS cycles; in the symmetric case $w'_A = w_A$. From now on the symmetric case is only considered since the non-symmetric can be transformed easily to a symmetric case.



Fig.7.2.3. Banded matrix squaring $w_A^{=4}$, $p_A^{=3}$, $q_A^{=2}$.

The input for a squaring stage can be reduced to only one matrix, if the inputs of the hex-arrays in Fig.7.2.2,3 are connected to a preprocessing element, producing two copies of a given matrix. The additional complexity introduced by the branching module is insignificant in comparison to the complexity saved by the reduction of the input requirements. A straightforward way to implement the branching operation is by means of a crossbar switch with fixed interconnections of input to output paths; alternatively optical interconnection is also possible.

Fig.7.2.4 illustrates a block of a pipeline design for successive matrix squaring. Notice that for a matrix with w=p+q-1, in the one copy of the matrix the upper diagonal k, $k=1,2,\ldots,p-1$ should be delayed for k cycles. For the other copy, the input of the q-1 lower diagonals is delayed accordingly. The total area required by a pipeline in order to produce $A^{2^{S}}$ is

$$\sum_{i=0}^{s-1} (2^{i} w_{A}^{-} (2^{i} - 1))^{2}$$
(7.2.6)

IPS cells, while the total time is given by (7.2.5). It is assumed that the area occupied by the branching elements and the reformatting delays is negligible. The major disadvantage of this pipeline design is the rapid increase in area requirements that is observed. An upper limit for the total area requirements can be obtained, similar to that in (7.2.3), using (7.2.6).


Fig.7.2.4. Matrix squaring for a banded matrix A with bandwidth w=5, p=q=3.

397

I.

The same hex-array can be used for dense matrix computations, as is indicated in Fig.7.2.5, for the multiplication of two square matrices with n=3. The area required for the full mmm is $(2n-1)^2$ -2n IPS cells, and the computation time is 3n-1 IPS cycles. In Fig. 7.2.6 a pipeline block for successive squaring of full matrices is given, for n=5. To produce $A^{2^{S}}$ a total area of

$$s((2n-1)^2-2n)$$
 (7.2.7)

IPS cells is required, and the total time is

$$s(3n-1)$$
 (7.2.8)

IPS cycles. Notice again the symmetric organisation of the reformatting delay elements in the input of the array. In this case no bandwidth increase is observed, i.e. all pipeline blocks are identical. Soft-systolic simulation programs for the unidirectional hex-array and the matrix squaring pipeline are given in A.4.1, A.4.2.

7.2.2 SYSTOLIC ITERATIVE DESIGNS

Instead of a pipeline design, an iterative scheme might be adopted, where the output of a pipeline block is fed back into its input; this is possible only in the case of full matrix computations, i.e. for Fig.7.2.6, since no bandwidth alteration occurs. An alternative systolic design with greater area and time efficiency for successive full mmm's is introduced in [237], and it is shown in Fig.7.2.7. This



Fig.7.2.5. Dense matrix multiplication on unidirectional hex-array, n=3.



.

Fig.7.2.6. Matrix squaring pipeline block for a full (nxn) matrix A, with n=3.

- 400 -





- 401 -

array can be used for the time expansion scheme, since its output is immediately fed back to the input. The area requirements of this array are approximately n^2 IPS cells, if the feedback control is assumed to occupy insignificant area. The array can be expanded by means of multiplexing elements that allow for matrix computations of the form AB+X to be performed. In this case, the area is approximately $(n+1)^2-1$ IPS cells. The computation time is 5n-1 IPS cycles, if we assume that the feedback control requires a full IPS cycle; however, a new mmm can start in 2n+1 cycles.

The matrix squaring operation for this 're-usable' mmm array is given in Fig.7.2.8, where the i/o sequences are given together with the feedback control (1=on, 0=off) and the number of mmm's required. Thus, the computation of $A^{2^{S}}$ in the re-usable mmm array takes a time of 2n(s+2)+n-1IPScycles.



Fig.7.2.8. Iterative array configuration for successive matrix squaring.

7.3 <u>SYSTOLIC ITERATIVE SOLUTIONS OF LINEAR SYSTEMS</u> <u>USING</u> <u>MATRIX POWERS</u>*

Consider the system of n linear equations $A\underline{x}=\underline{b}$, and the iterative methods for the solution of the system described in section 6.3. Hotelling has expressed these iterative methods in a matrix format where the linear system is written in the form [131], [148] :

$$\begin{bmatrix} 1 & \underline{o}^{\mathrm{T}} \\ \underline{b} & -A \end{bmatrix} \cdot \begin{bmatrix} 1 \\ \underline{x} \\ \underline{x} \end{bmatrix} = \underline{0}$$
 (7.3.1)

Thus, the J, JOR methods (6.3.3,4) can be rewritten as,

$$\underline{y}^{(k+1)} = \underline{T}\underline{y}^{(k)}$$
(7.3.2)

where

$$\underline{y}^{\mathrm{T}} = [1] \underline{x}^{\mathrm{T}}$$
 (7.3.3)

and

$$\mathbf{T} = \begin{bmatrix} 1 & \underline{\mathbf{o}^{\mathrm{T}}} \\ \underline{\mathbf{g}} & (1-\omega)\mathbf{I}+\mathbf{M} \end{bmatrix}$$
(7.3.4)

with \underline{y} and \underline{T} possessing identical column partitionings. Similarly the GS,SOR methods (6.3.5,6) are modified as follows :

$$\underline{y}^{(k+1)} = R\underline{y}^{(k)}$$
(7.3.5)

* Part of this section has been presented in the Workshop on VLSI Computation, Univ. of Leeds, Feb. 16 1987, and in Conf. NUMETA 87, Swansea, Wales. where

$$R = T_n T_{n-1} \cdots T_1$$
 (7.3.6)

with

$$\mathbf{T}_{i} = \begin{bmatrix} 1 & \underline{0}^{1} \\ \underline{g}_{i} & \mathbf{I} + ((1-\omega)\mathbf{I}_{i} + \mathbf{M}_{i}) \\ \mathbf{I} + ((1-\omega)\mathbf{I}_{i} + \mathbf{M}_{i}) \end{bmatrix}$$
(7.3.7)

where $\underline{g_i}$, I_i , M_i have non-zero elements only in row i. The advantage of writing the iterative methods in matrix form is that the sequence of matrices T, T^2 , T^4 , ..., T^{2^k} or R, R^2 , R^4 , ..., R^{2^k} can readily be obtained through successive matrix squaring. Thus, k successive squarings will be equivalent to 2^k complete iterations of the methods in their original form. A further advantage is that the problem of solving a system of linear equations has been reduced to the problem of performing a sequence of mmm operations. If the iterative process is convergent the first column of the matrix T^{2^k} or R^{2^k} will converge to the solution vector \underline{x} .

An important feature of matrix T is that its squaring operation can be partitioned as follows :

 $\begin{bmatrix} 1 & \underline{o}^{\mathrm{T}} \\ \underline{y} & c \end{bmatrix}^{2} = \begin{bmatrix} 1 & \underline{o}^{\mathrm{T}} \\ \underline{y^{+}cy} & c^{2} \end{bmatrix}$ (7.3.8)

where $C = (1-\omega)I+M$ and matrix C retains all the sparsity characteristics of the original matrix A. Therefore, the squaring of matrix T can be analysed into a mmm and a mvips if this partitioning yields better area results, which is true if A is a banded matrix. However, it should be noted that the successive matrix squarings will destroy the matrix profile as it is indicated in section 7.2. In the case of matrix R the partitioning of the mmm computation has limited effect since the sparsity characteristics of matrix A are only partially retained on matrix R.

The first column vector that is produced from (7.3.8) after a number of successive squarings can be expressed as $\underline{x} = \underline{y} + C\underline{y} + C^2 (\underline{y} + C\underline{y}) + C^4 (\underline{y} + C\underline{y} + C^2 (\underline{y} + C\underline{y})) + \dots$

 $= \underline{y} + C \underline{y} + C^{2} \underline{y} + \dots$ (7.3.9) = $\underline{y} + C (\underline{y} + C (\underline{y} + \dots + C\underline{y}) \dots)$.

indicating the equivalence of this process with the one proposed in chapter 6.

7.3.1 SYSTOLIC DESIGNS

From the description of the Hotelling iterative methods it is evident that the main computational effort is the calculation of successive mmm's. Systolic designs for this computation has been described in section 7.2, for both area and time expansion schemes.

Firstly, the iterative array implementation (time expansion) is discussed, using the systolic array of Fig.7.2.7. The system configuration in the case of the J,JOR-Hotelling methods is given in Fig.7.3.1: k iterations of the method require 2(n+1)(k+2)+1 cycles, where n is the size of matrix A. A similar configuration for the GS,SOR-Hotelling methods is illustrated in Fig. 7.3.2. The computa-



Fig.7.3.1. Iterative array configuration for the J,JOR-Hotelling methods.



Fig.7.3.2. Iterative array configuration for the GS,SOR-Hotelling methods.

tion of k iterations takes 2(n+1)(n+k+1)+1 cycles. The only difference in the two methods is the input sequence for matrix T, and the corresponding feedback sequence. Softsystolic simulation programs are given in A.4.3, A.4.4.

A pipeline design similar to that of Fig.7.2.6 can be implement the same iterative methods, if full used to matrices are involved. Especially for the J, JOR-Hotelling method a partitioned matrix squaring pipeline can be used, when matrix A is banded, implementing relation (7.3.8). A pipeline block is given in Fig.7.3.3 for a matrix C with bandwidth w=p+q-1 (here w=5). The partitioned matrix squaring block comprises a mmm and a mvm array that perform in parallel a matrix squaring and a mvips computation. Since the delay introduced by the computation of the mvm array is w+p-1 cycles, whereas the delay of the computation in the mmm array is only w cycles, the output of the mmm array must be delayed by p-1 IPS cycles. In general, for k iterations the pipeline requires an area of

$$\sum_{j=1}^{k-1} (2^{j-1} w_{A}^{-} (2^{j-1} - 1))^{2} + \sum_{j=1}^{k} (2^{j-1} w_{A}^{-} (2^{j-1} - 1))$$
(7.3.10)

IPS cells and the computation time is

$$n + \sum_{j=1}^{k} (2^{j-1}(w_A + p_A) - (2^{j-1} - 1)) - k$$
 (7.3.11)

A soft-systolic simulation program is given in A.4.5. Notice that the area of the last matrix squaring stage is not taken into account since its computation is redundant.



Fig.7.3.3. Matrix squaring and matrix-vector inner product step for a banded matrix C, w=5, p=q=3.

- 410 -

The input to both the iterative and the pipeline designs can be provided by a preprocessing element similar to that described in chapter 6 for the production of M, g for given A, <u>b</u>, (see Fig.6.3.11).

7.4 <u>SYSTOLIC DESIGNS FOR EIGENVALUE-EIGENVECTOR COMPUTATION</u> USING MATRIX POWERS

Let the eigenvalues of a (nxn) matrix A be real and ordered according to absolute value so that,

$$|\lambda_1| > |\lambda_2| \ge |\lambda_3| \ge \dots \ge |\lambda_n|$$
(7.4.1)

Now let A operate repeatedly on a vector \underline{u} , which is a linear combination of the eigenvectors of A, (section 2.4):

$$\underline{\mathbf{u}} = \mathbf{c}_1 \underline{\mathbf{x}}_1 + \mathbf{c}_2 \underline{\mathbf{x}}_2 + \dots + \mathbf{c}_n \underline{\mathbf{x}}_n$$
(7.4.2)

Then, through a series of successive mvm's, we have

$$A^{k}\underline{u} = \lambda_{1}^{k} \{c_{1}\underline{x}_{1} + c_{2}(\frac{\lambda_{2}}{\lambda_{1}})^{k}\underline{x}_{2} + \dots + c_{n}(\frac{\lambda_{n}}{\lambda_{1}})^{k}\underline{x}_{n}\}$$
(7.4.3)

For k sufficiently large, we have,

$$A^{k}\underline{u} \cong \lambda_{1}^{k}c_{1}\underline{x}_{1}, \quad c_{1} \neq 0.$$
(7.4.4)

Thus, the dominant eigenvalue λ_1 is calculated as

$$\lambda_{1} \approx (A^{k}\underline{u})_{i} / (A^{k-1}\underline{u})_{i}, \quad i=1,2,...,n$$
 (7.4.5)

while $A^k \underline{u}$ is proportional to the corresponding eigenvector \underline{x}_1 , providing no overflow occurs. The convergence rate is determined by

$$|\lambda_2/\lambda_1|^k < \varepsilon \tag{7.4.6}$$

where ε is a specified tolerance. The Power Method algorithm with a normalising strategy included can be formulated as follows [298] : Given χ_0 , an arbitrary vector, and let the sequences \underline{z}_k , \underline{y}_k be defined by the equations,

$$\frac{z_{j+1}}{y_{j+1}} = \frac{z_{j+1}}{\mu(z_{j+1})}, \quad j=0,1,\dots,k-1$$
with
(7.4.7)

$$\mu(\underline{\mathbf{x}}) = ||\underline{\mathbf{x}}||_{\infty}$$
(7.4.8)

For k sufficiently large,

$$\underline{\mathbf{y}}_{\mathbf{k}} \cong \underline{\mathbf{x}}_{\underline{\mathbf{l}}} / \mu(\underline{\mathbf{x}}_{\underline{\mathbf{l}}})$$

$$\mu(\underline{\mathbf{z}}_{\mathbf{k}}) \cong \lambda_{\underline{\mathbf{l}}}$$
(7.4.9)

A modified version of the method is proposed in [148], where the normalisation factor $\mu(\underline{x})$ is defined as:

$$\mu(\underline{x}) = \text{the first non-zero element of } \underline{x} . \qquad (7.4.10)$$

Thus, the dominant eigenvalue λ_1 is given by the first nonzero component of \underline{z}_k , and \underline{y}_k corresponds to \underline{x}_1 divided by its first non-zero component. The scaling operation by $\mu(\underline{z}_j)$ aims to prevent overflow due to repeated mvm's. However, if (7.4.10) is used, the scaling may be cancelled, if the first non-zero element of \underline{z}_j is either too big or too small. On the other hand, (7.4.8) imposes a considerable delay in the computations, since \underline{z}_{j+1} cannot be computed before all elements of \underline{z}_j are known.

The convergence rate in (7.4.6) can be improved, if, instead of forming the sequence $A^k \underline{y}$, a sequence of matrix powers is constructed directly [298]. The Matrix Squaring method has the advantage that when A^j is computed, A^{2j} is

obtained in one mmm. Hence the sequence $A, A^2, A^4, \ldots, A^{2^{S}}$ is easily built up. For sufficiently large s [56]:

$$A^{2^{s}} \cong \lambda_{1}^{2^{s}} c_{1} \underline{x}_{1} \underline{x}_{1}^{T}$$
 (7.4.11)

Each column of $A^{2^{S}}$ is proportional to \underline{x}_{1} , each row is proportional to \underline{x}_{1}^{T} . To obtain the eigenvalue λ_{1} any column of $A^{2^{S}}$ can be multiplied by A. Then (7.4.5) can be applied between the original column of $A^{2^{S}}$ and that of $AA^{2^{S}}$.

Generally the successive squares will either increase or decrease rapidly in size and overflow may occur. This can be avoided by scaling each matrix by a power of 2. This introduces no additional rounding errors. The convergence rate is determined by

$$\left|\lambda_{2}/\lambda_{1}\right|^{2^{s}} < \varepsilon , \qquad (7.4.12)$$

If A is not sparse, there is about n times as much work in one step of the Matrix Squaring method as in one step of the Power method. Thus, Matrix Squaring is more efficient for $|\lambda_2/\lambda_1|$ close to 1; if A is banded its sparseness is destroyed by Matrix Squaring.

A combination of the two methods can be considered as the application of a limited number of matrix squaring steps, followed by simple power method iterations. This combination, i.e. the Raised Matrix Power (RMP) method, seems to be the most economical strategy for finding the dominant eigenvalue and the corresponding eigenvector. The number r of matrix squaring steps depends on the ratio λ_2/λ_1 , but, a simple estimation can be derived [183]

$$\mathbf{r} = \lfloor \log_2(\sqrt{n}) \rfloor . \tag{7.4.13}$$

Therefore, the first phase of the RMP method involves the calculation of $B=A^{2^{r}}$. Then, after k simple iterations:

$$B^{k}\underline{u} = (\lambda_{1}^{*})^{k}c_{1}\underline{x}_{1}, \qquad (7.4.14)$$

where

$$\lambda_1 = \lambda_1^{2^r} \tag{7.4.15}$$

as is shown in [148],[298]. Thus, the dominant eigenvalue is obtained by a series of r square root extractions; \underline{x}_1 is collected as before. The rate of convergence is now determined by

$$(|\lambda_2/\lambda_1|^{2^r})^k < \varepsilon \quad . \tag{7.4.16}$$

For the first phase of the method the same limitations apply as in Matrix Squaring; however the overflow effect is now alleviated and similarly for r small the sparseness is not totally destroyed.

7.4.1 SYSTOLIC DESIGN CONSIDERATIONS

The main computational effort in the Power Method is the calculation of a series of mvm's; the Matrix Squaring method is based on successive mmm's, and the RMP method combines both the computations. Both computations are iterative, i.e. the output of step k forms the input of step k+1. The systolic implementation of iterative mvm and mmm operations has been already considered, using area and time expansion approach schemes, in chapter 6 and section 7.2.

An additional complexity for the iterative algorithms discussed is the normalisation or scaling that is necessary between two successive iteration steps. For the Power method, the technique in (7.4.8) requires a full search of the vector to be normalised; this imposes a delay of approximately n cycles and therefore cancels the potential of the pipelined implementation of the algorithm. As an alternative, the technique in (7.4.10) requires approximately 1 cycle delay but may cause adverse effects than those expected. A third alternative would be an externally controlled normalisation factor based on a priori knowledge of the specific matrix characteristics.

In the case of the area expansion approach (pipeline), the technique in (7.4.10) or the externally controlled factor, a combination of the two methods can be used; for the time expansion iterative design an additional possibility is to use the technique (7.4.8), but for the normalisation of the next iterative vector [56]:

 $\underline{Y}_{j+1} = \underline{z}_{j+1} / \mu(\underline{z}_j) t , j=0,1,...,k-1;$ (7.4.17)

for the first iteration the normalisation factor is assumed

- 416 -

to be externally given; t is a given adjustment constant.

Similar observations can be made for the Matrix Squaring method: for the pipeline approach the scaling should be based on prior knowledge of the matrix characteristics, possibly with a combination of a limited sample of the output of the certain pipeline stage. For the feedback approach there is the additional possibility to search the matrix and apply the corresponding scaling in the next iteration, with some predefined adjustment. However, the searching process should be adapted to the mmm calculation, which is performed in parallel.

The RMP method, as a combination of the two preceding methods, shares the same problems and possible solutions. Especially for the matrix squaring phase, the scaling can be avoided or fixed since only a limited number of squaring steps is involved.

7.4.2 SYSTOLIC DESIGNS

A systolic pipeline block for the Power method is shown in Fig.7.4.1, based on the interconnection of the unidirectional mvm array. The normalisation cell that is placed between two successive pipeline stages can operate using (7.4.10), a systolically propagated normalisation factor, or a combination of the two techniques, so that the dynamic range of the system is efficiently used. The computation performed by the normalisation cell can be

- 417 -

```
if
  (not first.found) and (yin <> 0)
    par
        first := first * yin
        first.found := true
yout := yin / first
if
    reset
    par
        first.found := false
        first := norm.factor
```

The area and time complexity of the computation of the normalisation cell can be reduced if the multiplication and division are replaced by shift operations. Therefore, the Power method for an (nxn) banded matrix can be performed on in Fig.7.4.1, in n+k(w+p) cycles, a pipeline as where w=p+q-1 is the bandwidth of the matrix and k is the number of iterations. The area required is kw IPS cells + k normalisation cells. The length of a time cycle is determined by the time complexity of the normalisation cell. A softsystolic simulation program for the Power method is given in A.4.6.

A systolic pipeline block for the Matrix Squaring method for a banded matrix A is shown in Fig.7.4.2, based on the unidirectional mmm array. The major characteristic of this pipeline is the sharp increase of the area requirements as k increases, as is explained in section 7.2. This fact imposes an upper limit on the number r of the pipeline stages, i.e. the number of matrix squaring steps allowed, as well as on the bandwidth of matrix A.

The scaling postprocessor array is an extension of the

- 418 -



Fig.7.4.1. Power method pipeline block for a banded matrix A with bandwidth w=5, p=q=3.

.



Fig.7.4.2. Matrix Squaring method pipeline block for a banded matrix A with bandwidth w=5, p=q=3.

- 420 -

normalisation cell used for the Power method pipeline. The simplest post-processor can be one where a scaling factor is preloaded to all scaling cells, performing a simple division (or shift-right). In order to achieve greater flexibility the scaling should be adapted to the matrix being processed; however this is very time consuming, if the whole matrix is to be searched. Instead the first row and column can be searched and the scaling factor determined using this data; for small bandwidths the search can be performed serially, otherwise a systolic search tree might be used. In the case of the simple scaling post-processor the area and time complexity is comparable to an IPS cell.

Therefore the Matrix Squaring method for a banded matrix A can be performed on a pipeline as in Fig.7.4.2 in

$$n + \sum_{i=0}^{s-1} \{ (2^{i}w - (2^{i} - 1)) + 1 \}$$
(7.4.18)

cycles and with area requirements of

$$\sum_{i=0}^{s-1} \{ (2^{i}w - (2^{i}-1))^{2} + (2^{i}w - (2^{i}-1)) \}$$
(7.4.19)

IPS and scaling cells. Again the time unit is determined by the scaling cell complexity. A soft-systolic simulation program for the Matrix Squaring pipeline is given in A.4.7. A similar pipeline block for full matrix computation can be readily derived from Fig.7.2.6.

The RMP method can be implemented by simply interconnecting the Matrix Squaring and the Power method pipelines; the only interface required is a set of reformatting registers, imposing no additional delay. Since only a limited number of matrix squaring steps will be used the scaling process can be relaxed. Another reason that dictates a small number of matrix squaring steps is the fact that, in the case of banded matrices, the area of the Power method pipeline blocks depends on the bandwidth of the output matrix of the Matrix Squaring pipeline.

The iterative systolic array for the Power method is shown in Fig.6.6.3, with the difference that the multiplexer cell now incorporates the normalisation process. The normalisation is similar to that used in the area expansion cases; as an alternative, the normalisation cell can be modified to operate as suggested in (7.4.17). Therefore, the Power method for a full (nxn) matrix can be performed on an iterative array as in Fig.6.6.3 in 2n(k+1)+1 cycles, and the area required is n IPS cells + 1 boundary cell. Each of the IPS cells has a memory module of n words. A soft-systolic simulation program for the Power method iterative array is given in A.4.8.

The iterative systolic array for the Matrix Squaring method is shown in Fig.7.2.7, with the difference that the multiplexers include the scaling generation. The i/o feedback sequence is given in Fig.7.4.3. The scaling process has complexities similar to those of the area expansion case of Matrix Squaring. Firstly, an externally determined scaling





- 423 -

factor can be used; in order to improve the efficiency of the scaling process a sample of the matrix must be taken. For example, for a given class of matrices, it is possible to search a specific row or column of the iterates. More generally, the row or column to be searched may change dynamically from iteration to iteration.

However, the searching of a column and a row of the matrix imposes a delay of more than n cycles. Extending the idea in (7.4.17), the scaling factor calculated during iteration k can be used to determine the matrix for iteration k+1.

The final mmm, i.e. $AA^{2^{S}}$ can also be computed in the same array, provided that A is stored by the systolic array interface and produced in the final mmm. Therefore the Matrix Squaring method for a full (nxn) matrix can be performed on an iterative array as in Fig.7.2.7 in 2n(s+3)+1 cycles; the area required is n² IPS cells and 2n boundary cells. A soft-systolic simulation program for the Matrix Squaring iterative array is given in A.4.9.

The RMP can be performed on either the Matrix Squaring array or a combination of the Power method and the Matrix Squaring arrays. In the first case, r squaring steps are followed by k degenerate mmm's between $B=A^{2^r}$ and $[\underline{x_i}|0]$, $i=0,1,\ldots,k$. Thus only n+1 processors are active; the remaining processors act as simple memory registers. In the second case, B is passed to the Power method array so that the Matrix Squaring array is free for another computation.

- 424 -

7.5 SYSTOLIC COMPUTATION OF THE EXPONENTIAL OF A MATRIX

Mathematical models of many scientific and engineering problems involve systems of linear, constant coefficient ordinary differential equations, of the form

$$\frac{dx}{dt} = A(t)x_0$$
(7.5.1)

where A is a given (nxn) matrix. In principle, the solution is given by,

$$\underline{x} = e^{A} \underline{x}_{0}, \qquad (7.5.2)$$

where \underline{x}_0 is a given initial condition. Now, e^A can be formally defined by the convergent power series, (section 2.2):

$$e^{A} = I + A + \frac{A^{2}}{2!} + \frac{A^{3}}{3!} + \dots$$
 (7.5.3)

The systolic computation of this matrix function is discussed herein. The definition of e^A in (7.5.3) can be the basis for a simple algorithm calculating the exponential of a matrix using Taylor series approximation techniques. Thus,

$$e^{A} \approx T_{k}(A) = \sum_{i=0}^{k} (\frac{1}{i!} A^{i})$$
 (7.5.4)

However, such an algorithm is known to be unsatisfactory, since k is usually very large, for a sufficiently small error tolerance. Furthermore, the round-off errors and the computing costs of the Taylor approximation increase as ||A|| increases. These difficulties can be controlled by exploiting a fundamental property of the exponential function:

$$e^{A} = (e^{A/m})^{m}$$
 (7.5.5)

In the scaling and squaring method, m is chosen so that :

$$m = 2^{j}$$
 and $\frac{||A||}{2^{j}} \leq 1$. (7.5.6)

With this restriction, the Taylor approximation in (7.5.4) can be satisfactorily used, and then e^{A} is formed by j successive squarings [117]. For a given error tolerance ε , and magnitude of $||A||_{1}$, Table 7.5.1 summarizes the optimum (k,j) associated with $[T_{k}(A/2^{j})]^{2^{j}}$, [210].

7.5.1 SYSTOLIC DESIGNS

The systolic implementation of successive matrix squarings has been considered in section 7.2. The calculation of $T_k(A/2^j)$ as in (7.5.4) is a matrix polynomial computation in which Horner's scheme of nested multiplication can be used [237],[288]:

$$s_0 = x_k$$

 $s_i = As_{i-1} + x_{k-i}$, for i=1,2,...,k, (7.5.7)

where

$$x_{i} = \frac{1}{i!} I$$
, $S_{k} = T_{k}(A)$. (7.5.8)

Thus, the evaluation of the Taylor series approximation has been reduced to a series of matrix-matrix inner product steps (mmips) as in (7.5.7). Following the discussion in section 7.2 and chapter 6, the process of repeated mmips can be expanded either in time, yielding an iterative network,

E A A A	10 ⁻³	10 ⁻⁶	10 ⁻⁹	10 ⁻¹²	10 ⁻¹⁵
10 ⁻²	1,0	2,1	3,1	4,1	5,1
10 ⁻¹	3,0	4,0	4,2	4,4	5,4
10 ⁰	5,1	7,1	6,3	8,3	7,5
10 ¹	4,5	6,5	8,5	7,7	9,7
10 ²	4,8	5,9	7,9	9,9	10,10
10 ³	5,11	7,11	6,13	8,13	8,14

Table 7.5.1. Optimum (k,j) for given ε and $||A||_1$.

or in area, producing a pipeline (see Fig.7.5.1). Both these approaches are pursued herein.

The re-usable array shown in Fig.7.2.7, can perform both matrix polynomial and matrix squaring computations. The i/o/feedback sequences for the computation of e^A are shown in Fig.7.5.2. The area required is not more than n^2+2n IPS cells, and the computation time is 2n(k+j+2)+n-11PS cycles. A soft-systolic simulation program in OCCAM is given in A.4.10.

A pipeline design calculating the exponential of а banded matrix is shown in Fig.7.5.3. It consists of two parts: the first part performs the matrix polynomial computation, while the second part is the matrix squaring pipeline, already discussed in section 7.2. The matrix polynomial pipeline exhibits similar characteristics to the matrix squaring pipeline, as regards the bandwidth alteration of the output matrix. However, the bandwidth increase is not so rapid, since in stage i the bandwidth of the output matrix is iw-(i-1) while in matrix squaring the same bandwidth is $2^{i}w-(2^{i}-1)$, where w is the bandwidth of matrix A. This is because the bandwidth of $A/2^{j}$ is fixed in all mmips stages, and X_i , i=0,1,... have bandwidth equal to 1.

On the other hand, the matrix polynomial pipeline introduces two other problems. Firstly it is necessary to propagate matrix $A/2^{j}$ along the pipeline systolically, in a way similar to that described in the mvm pipelines of chapter 6. Then, the development in the bandwidth of the two



 $S_{k}=S_{k-1}+AX_{k-1}$ k=1,2,... i=0,1,...,k



Fig.7.5.1. Time and area expansion for mmips operation.

			•						cyc]	e	feedb	ack	input	output
i .									k+j+	2	1		0	s ²⁾
									•		•		•	•
									•		•		•	•
									k+ 3	3	1		0	s ² _k
					•				k+2	<u>.</u>	1		0	s, k
									k+1		0		I	R
									k		0		A/2 ^j	R .k-1
									•		•		•	•
									• 2		0		A/2 ^j	R
								•	1		0		a/2 ¹	~o
									-		0			Ť
													<u> </u>	
													Ϋ́	
														 L
output	² ئ		ء2	c	Ð	Ð		в	~			ļ		
output	k	•••	٦k	k	ጉአ	°k-	1	•• ••	•	4	1	1		
									-		·			
											Ţ			
input	0	•••	0	0	×o	×1	• • •	× _{k-1}	x k		(+)→	1		
6	•			•			•	•	-			L		
reedbaCK	T	•••	T	T	T	T.	•••	1	T					
cycle k	:+j+2	•••	k+3 :	k+2	k+1	.k.	• • •	2	1					

 $x_{i} = \frac{1}{i!} I$, $R_{i} = T_{i}(A/2^{j}) - X_{k-i}$, $S_{k} = T_{k}(A/2^{j})$, $S_{k}^{2^{j}} = e^{A}$

Fig.7.5.2. Iterative array configuration.

.

- 430 -

. .



Fig.7.5.3. Pipeline configuration for k=2, j=1, w=3.

431

matrices to be multiplied is uneven, thus leading to hexarrays as shown in Fig.7.5.4. General banded matrices can be transformed to matrices with equal bandwidths and semi-bands if appropriate 'dummy diagonals' are added. These diagonals are realised as delay elements.

As observed in Fig.7.5.4 (see also Fig.7.2.2), the mmm hex-array is designed for the multiplication of two (nxn) matrices with bandwidths $w_A = p_A + q_A - 1$, $w_B = p_B + q_B - 1$, and $w_A, w_B < < n, p_A = q_B$ (or $q_A = p_B$). Therefore, $|p_A - q_B|$ delays should be added in the input of the matrix with $\max(p_A,q_B)$. Furthermore, if the output sequence is to be kept uniform in all cases, another $|p_B-q_A|$ delays should be added in the output of the side with $\max(p_B,q_A)$. Thus, in general, the systolic mmm of two matrices introduces a delay of $\min(w_A, w_B) + |p_A - q_B| + |p_B - q_A|$ IPS cycles, and requires an area of $w_A w_B$ IPS cells and $max(w_A, w_B)^2 - w_A w_B$ delay registers.

Applying these results on the matrix polynomial pipeline for the computation of the matrix exponential, we conclude that the two pipelines require area of

$$1 + \sum_{i=0}^{k-1} (w(iw-(i-1)) + \sum_{i=0}^{j-1} (2^{j}w'-(2^{i}-1))^{2}$$
(7.5.9)

IPS cells and requires computation time of

$$n+w+\sum_{i=1}^{k-1}(iw-(i-1)) + \sum_{i=0}^{j-1}((2^{j}w'-(2^{i}-1)))$$
(7.5.10)

IPS cycles where w'=kw-(k-1). The area occupied by the delay registers is assumed to be negligible. Upper limits for the


Fig.7.5.4(a). Banded matrix multiplication, $w_A = 4$, $p_A = 3$, $q_A = 2$, $w_B = 3$, $p_B = 2$, $q_B = 2$.

- 433 -



Fig.7.5.4(b). Banded matrix multiplication, $w_A=3$, $p_A=2$, $q_A=2$, $w_B=4$, $p_B=3$, $q_B=2$.

values of k,j are imposed due to the bandwidth increase for the output matrix. Thus, if the maximum area available for a single systolic mmm computation is w_{max}^2 then, we have

$$2^{j-1}(kw-(k-1))-(2^{j-1}-1) \le w_{max}$$
 (7.5.11)

A soft-systolic simulation program for the matrix polynomial pipeline is given in A.4.11. A pipeline for the computation of the exponential of a full matrix can be obtained if the systolic design of Fig.7.2.6 is used. In this case, no bandwidth alterations are observed, and therefore the pipeline will consist of k+j identical mmm arrays.

7.6 CONCLUSIONS

The key requirement for the systolic calculation of successive matrix powers is the fact that the output of a given powering stage should form the input of the next stage. This leads to the need for all matrices taking part in the computation to move through the array at some stage during the computation. Thus, stationary systolic mmm arrays with preloaded coefficient matrices cannot be used; the special case of successive matrix multiplications, where one of the matrices is constant is considered in [211-212].

The unidirectional data flow restriction for the pipeline case, can be removed by means of a more complicated, reflected data flow for the original hex-array as proposed in [248-249]. However this leads to lower efficiency and unnecessary complexity for the boundary cells.

The transposition of a matrix can be effected by means of a simple systolic design, as shown in [221]; similar designs are proposed to reorder a matrix data sequence. These designs can be used in conjunction with the mmm systolic array described in [197] and elsewhere for successive matrix squaring, since this array accepts input by rows and by columns and produces output by diagonals. The same array is used in [211-212] for triple-matrix products, but it requires intermediate sbrage.

In sections 6.4, 6.5 it was shown that a linear system

of the form $A\underline{x}=\underline{b}$, with A a p-cyclic matrix, can be reduced to a set of p uncoupled linear systems, using the Frobeni us theorem. These computations can be readily implemented using the results of sections 7.3, 7.5, either by means of pipeline designs or iterative arrays.

Finally, the concept of iterative systolic structures, performing complex matrix computations, is further extended in [74], for the iterative solution of matrix equations; and in [88], [216], for the VLSI implementation of the Faddeev algorithm for matrix inversion.

In the remainder of this section the systolic matrix inversion and the systolic computation of several matrix functions are briefly discussed, using the recurrence relations and matrix polynomial approximation methods introduced in section 2.2.

7.6.1 SYSTOLIC INVERSION USING MATRIX POWERS

Let A be a (nxn) real matrix and we want to compute A^{-1} ; we assume that an approximate inverse B is known. Then, for

M = I-AB (7.6.1) the inverse is calculated as

 $A^{-1} = B(I-M)^{-1} = B(I+M+M^2+M^3+...)$ (7.6.2) with ||M|| < 1 [111]. The efficient calculation of the approximate inverse B is investigated in [223], so that the convergence condition is satisfied.

Three alternative ways of evaluating the sum

$$S_k = I + M + M^2 + ... + M^{k-1}$$
 (7.6.3)
are proposed in [32]:

(i) Nested multiplication (Horner's scheme): k-1 mmips
yield S_k:

$$S_k = I + M + (I + M(I + ... M(I + MI)...))$$
 (7.6.4)

(ii) Especially for $k=2^{(1+1)}$, successive squares of M are computed, and S_k is calculated. Starting from $S_0=I$, $S_2=I+M$ one computes S_4 , S_8 , ..., S_k in 1+1 steps, using the recurrence:

$$s_2i+1 = s_2i + s_2iM^{2i}$$
 i=0,1,2,...,1 (7.6.5)

$$S_k = (I+M)(I+M^2)(I+M^4) \dots (I+M^{2^{\perp}})$$
 (7.6.6)

Notice, that each step of (7.6.5,6) consists of 1 mmips and a mmm for matrix squaring and has approximately twice more work than that of (7.6.4). On the other hand, only 1+1 steps are required instead of 2^{1+1} . Further, if we truncate the power series in (7.6.2), after the first two terms. we obtain the Newton method for matrix inversion:

$$B_{k+1} = B_k(2I-AB_k).$$
(7.6.7)

Each step of the Newton method can be analysed into a mmips, i.e. $T_k=2I-AB_k$, and a mmm , i.e. $B_{k+1}=B_kT_k$.

All these recurrences can be readily realised using the systolic pipeline or iterative designs proposed in the previous sections. Especially for the Newton method, the iterative structure seems more appropriate, since B_k can be stored in order to be used in both steps of the iteration. Another method for parallel matrix inversion, based on matrix powers is discussed in [69], [293]. This method requires the calculation of $tr(A^i)$, $i=1,2,\ldots,k$, and of the coefficients of the characteristic polynomial of A, as discussed in section 4.5.

7.6.2 SYSTOLIC COMPUTATION OF MATRIX FUNCTIONS

The calculation of $A^{1/2}$ has been studied in [128], [129], [117]. The Newton's method in the form

$$P_{0} = A, Q_{0} = I$$

$$P_{k+1} = \frac{1}{2}(P_{k} + Q_{k}^{-1})$$

$$Q_{k+1} = \frac{1}{2}(Q_{k} + P_{k}^{-1}), k=0,1,2,...$$
(7.6.8)

is recommended as the most stable, but its main disadvantage for a systolic implementation is the computation of matrix inverses in each iteration. It is interesting to note that the computation of $A^{-1/2}$ involves no matrix inversion during the iterations [111], [229]:

 $Z_k = I - T_k A T_k$

 $T_{k+1} = T_k (I + \frac{1}{2} t_k), \ k=0,1,2,\dots$ where T_0 is an initial approximation of $A^{-1/2}$. Thus, apart from the importance of $A^{-1/2}$ itself in numerical applications [229], it can also provide $A^{1/2}$, since $A^{1/2} = AA^{-1/2}$. The systolic implementation of (7.6.9) is based on iterative mmips and mmm operations, and therefore computational structures similar to that of section 7.5 can be used. The iterative array seems especially more attractive since T_k must be temporarily stored, so that it can be used in both steps of an iteration.

In [117], [265] the calculation of 'cos(A), sin(A) is discussed, based on the double-angle method

 $\cos(2A) = 2\cos(A)^2 - I$

sin(2A) = 2sin(A)cos(A) (7.6.10) Thus, in a manner analogous to the matrix exponential computation, we can choose a scaling factor j, so that $||A||/2^{j} \leq 1$. Then, the Taylor series approximations for cos(B), sin(B) are calculated, where $B=A/2^{j}$, for a given number k of steps.

$$\sin(B) = S_0 = B - \frac{B^3}{3!} + \frac{B^5}{5!} - \dots$$

$$\cos(B) = C_0 = I - \frac{B^2}{2!} + \frac{B^4}{4!} - \dots$$
(7.6.12)

Finally, the actual values of cos(A), sin (A) are given by the recurrence

$$s_k = 2s_{k-1}c_{k-1}$$

 $c_k = 2c_{k-1}^2 - 1, k=1, 2, ..., j$ (7.6.13)

Again the similarity of the calculations with that of section 7.5 is evident. Two pipelines or two iterative arrays working in parallel can produce systolically cos(A), sin(A).

Finally, a similar computation is proposed in [117], [268], [127] for log(A), based on the Taylor series approximation

$$\log(A) = (A-I) - \frac{(A-I)^2}{2} + \frac{(A-I)^3}{3} - \dots$$
 (7.6.14)

However, no scaling method such as those given for the other matrix functions is generally available. Thus, the computation of (7.6.13) can be efficient only for ||A|| < 1.

CHAPTER 8

OPTICAL SYSTOLIC ALGORITHMS

8.1 INTRODUCTION

The optical implementation of systolic algorithms was originally introduced in [58], where a systolic mvm algorithm is implemented using LED's and Acousto-optic cells as input devices (transducers, see section 3.5). Since then a significant number of algorithms and architectures for optical systolic processing has been proposed, using acoustooptic, electro-optic, magneto-optic and other techniques for the realisation of the transducers.

Herein we concentrate on the optical implementation of basic matrix computations, such as mmm, LU decomposition and Gauss Elimination algorithms. In general, the optical realisation of these algorithms require 2-dimensional (2-d) i/o devices, i.e. transducers and detector arrays. Three types of optical processors using 2-d arrays have been proposed, according to the classification introduced in [49], [241].

Firstly, a direct mapping of a 2-d VLSI systolic archi-

tecture onto an optical computing structure leads to an optical processor with 2-d transducer and detector arrays. For example, in [58], a direct mapping of a systolic mmm algorithm is outlined. The same technique is utilized in [30] for the optical implementation of the engagement array processor. The reduction of many matrix computations in a series of mmm operations, is addressed in [28], [84]. Thus, the LU decomposition, direct solution of linear systems using Gauss Elimination, matrix inversion, QR factorization, etc., can be decomposed in a series of mmm's.

An interesting property of light transmission and detection is the summation of the intensities of light beams that are being collected simultaneously by the same detector (space integration, see for example [49-50]). This property can be used to reduce the dimensionality of either the transducer or detector arrays. Notice, however that more complex imaging systems are introduced, than that required for the direct mapping of the VLSI arrays.

Thus, in [12-14], [234] the outer product optical processor is described, using 1-d input devices and 2-d output array. A number of numerical and signal processing algorithms for the outer product processor are also discussed, such as mmm, LU decomposition, convolution, correlation, triple-matrix-product, etc. An integrated optical implementation of the outer product processor, using electro-optic processor elements, is addressed in [291].

- 443 -

An alternative approach is discussed in [54], where the detector array becomes 1-d at the expense of a 2-d transducer array. This architecture is used as a basis for the implementation of a wide range of numerical algorithms. Direct and iterative methods for the solution of linear and non-linear systems of equations are more specifically discussed in [51], [53], where these algorithms are implemented as a series of mvm's.

Some further examples of the optical implementation of systolic algorithms are given in [56] for the Power and Matrix squaring methods for eigenvalue computation; in [290], [292] for polynomial computation. In [2] some simulation and experimental results of the combination of digital electronic and analog optical components in a hybrid processor are given, for the case of iterative solution of linear systems of equations. Possible error sources and faulttolerance techniques are investigated in [52], [57].

In this chapter the basic techniques for the optical implementation of systolic algorithms are exemplified, and some new algorithms for banded matrix computation are presented.

In section 8.2 the R+F and the unidirectional systolic mmm algorithms are directly mapped onto an optical processor. In the following section the LU decomposition and the R+F improvement of the method are also mapped on a similar processor. Furthermore the solution of triangular linear systems is discussed.

In section 8.4 the outer product processor for banded matrix computations is introduced, and the mmm and LU decomposition algorithms are applied. The same basic processor is used in the following section for the optical implementation of Gauss Elimination algorithms.

In the last section of this chapter, the improvement of the accuracy of the optical computations is discussed, as well as the combination of digital and optical computations. Finally, some further optical systolic algorithms are proposed, mainly based on the iterative systolic algorithms described in the previous chapters.

8.2 OPTICAL SYSTOLIC BANDED MATRIX MULTIPLICATION*

8.2.1 MAPPING OF A R+F ALGORITHM ON AN OPTICAL PROCESSOR

The R+F method improves the efficiency of some basic systolic algorithms without causing any changes in the structure of the cells or in the communication geometry of the VLSI system and only requires some additional prepostprocessing of the matrix elements (see section 3.3). The fact that no changes are made on the underlying computational structure makes the R+F method suitable for direct optical implementation: the case of a linear array of acoustooptic cells is examined in [93]. Herein the concept is expanded to 2-d geometry, using matrix multiplication as the target algorithm.

The R+F method for matrix multiplication is discussed in [21] ; Fig.8.2.1 illustrates the method with an example, for two tridiagonal matrices $(w_A = w_B = 3)$ of order n=5; this example is used for the illustration of the optical implementation of the method. The bands are rotated and folded in the directions shown by the arrows and the elements in the boxes of dotted lines are repeated once, so that the bands are extended by one row/column. The input and output data sequences are shown in Fig.8.2.2, for the VLSI hex-connected systolic array.

* This section is a revised version of a paper to be published in Optics Communications.



Fig.8.2.1. R+F matrix multiplication.

447



Fig.8.2.2. R+F matrix multiplication array.

The basic concepts of an optical system implementing R+F matrix multiplication algorithm are explained with the the help of Fig.8.2.3,4. This system is based on an optical processor originally outlined in [58], where some technical details and possible extensions are also given. The processor shown in Fig.8.2.3,4 consists of two 2-d transducer arrays, each with $w_{h}w_{B}$ pixels. Between the transducer planes there is a system of lenses that focuses the light beams emitted by the first plane onto the appropriate pixels of the second plane. Notice that the distribution of the pixels on the arrays allows for the direct mapping of the dataflow the of mmm algorithm as illustrated in Fig.8.2.2; i.e. the optical implementation follows the same dataflow principles the hex-connected VLSI systolic array, although the data as now travel in three parallel planes instead of one.

The light beams are modified in intensity as they pass through the transducer arrays, and then they enter an imaging system that drives the light beams onto an array of light detectors which are connected to a set of shift registers. The detectors transform the incident light beams to an electrical charge, according to their intensity, and this charge is accumulated to the associated shift register. The system is controlled by a clock; thus, the input of the elements of matrices A and B into the transducer drivers, to be transformed into modulating segments, and the shift of the charge packets, which represent the elements of matrix C, are synchronised.



Fig.8.2.3. Details of optical processor for R+F mmm.

- 450



Fig.8.2.4. Matrix multiplication optical processor.

- 451

At the moment illustrated in Fig.8.2.3, modulating segments proportional to b_{11} , b_{12} , b_{55} , b_{21} and a_{11} , a_{21} , a_{55} , a_{12} have been input to the transducer drivers producing modulating segments. The light beams transmitted through the imaging system to the detector array are proportional to $a_{11}b_{12}$, $a_{21}b_{11}$, $a_{55}b_{55}$ and $a_{12}b_{21}$. These beams are imaged onto the CCD detectors, and the appropriate shift registers, which are c $_{12'}$ c $_{21'}$ c $_{55}$ and c_{11} respectively. The presence of dummy elements in the transducer planes means that no useful light beam is formed and therefore no useful result is collected by the shift detectors.

Thus, by projecting the modulated light onto the corresponding register, we achieve the accumulation of the necessary inner products onto the appropriate coefficients of matrix C, i.e. we achieve the same result as in the VLSI array of Fig.8.2.2. In the next clock tick the modulating segments travel in the direction shown by the arrows in Fig.8.2.3 and the transducer drivers produce new modulating segments from the elements of matrices A, B. Similarly the charge packets on the shift registers travel upwards accumulating in this way the inner products that form the coefficients of matrix C.

In general, the optical implementation of the R+F systolic mmm algorithm, for two banded matrices with bandwidths w_A , w_B respectively, requires an optical system of two 2-d transducer arrays, and one 2-d detector-shift register array, each one with w_Aw_B pixels.

8.2.2 MAPPING OF THE UNIDIRECTIONAL MMM ARRAY ON AN OPTICAL PROCESSOR

The unidirectional banded matrix multiplication systolic algorithm discussed in chapter 7 is now mapped onto the optical processor shown in Fig.8.2.4. If the direction of the dataflow in the detector-shift register plane is reversed and the input data sequence format takes the compact form shown in Fig.7.2.2 then the optical processor is modified as shown in Fig.8.2.5. Notice that there are no dummy elements in the input sequences and furthermore the input-output format is in natural order, i.e. no additional processing is necessary. The hardware requirements remain the same as for Fig.8.2.3.

In both the optical systolic mmm algorithms described, the computation time is the same as in the corresponding VLSI implementation. Furthermore, all the issues concerning matrices with unequal semibands, as well as dense matrices, discussed in chapter 7 for the unidirectional mmm VLSI systolic array, can be readily applied on the optical processor of Fig.8.2.5.

In general the direct mapping of a VLSI systolic array onto an optical processor can be effected as follows. Each of the operands of the IPS computation moves in a separate plane: for example, matrices A, B and C move in different



Fig.8.2.5. Details of optical processor for unidirectional mmm.

planes. More specifically, the operands involved in the multiplication move on the transducer planes, whereas the operand that accumulates the results (i.e it is involved in the addition) moves on the detector-shift register plane. The dataflow and the cell interconnection pattern of the VLSI array is preserved in the optical implementation. Each data stream retains the data sequence format and relative directions of movement through the optical processor. Thus, all timing, efficiency and complexity results concerning the VLSI arrays can be directly applied to their optical counterparts.

8.3 OPTICAL SYSTOLIC LU DECOMPOSITION AND SOLUTION OF TRI-ANGULAR SYSTEMS*

The R+F method improves the efficiency of the systolic LU decomposition and triangular system solution algorithms as explained in sections 3.3, 5.2. Herein, initially the optical implementation of the systolic LU decomposition is described. Then the R+F method is applied and the optical solution of the resulting upper and lower triangular systems is discussed. The attention is concentrated on the case of tridiagonal matrices, for simplicity, but the method can be extended in general matrices as discussed in [19].

The example used and the corresponding VLSI systolic implementation are given in section 3.3. Only the case of n odd (n=5) is considered, since the even case can be easily derived from it. Notice that for the LU decomposition algorithm to be optically implemented the optical processors involved must be capable of operating in the full range of real numbers.

8.3.1 OPTICAL LU DECOMPOSITION

The basic concepts of an optical implementation of the decomposition of a matrix A in lower and upper triangular

^{*} This section is a revised version of a paper presented in the 4th Int. Symposium on Optical and Optoelectronic Applied Science and Engineering, The Hague, The Netherlands, April 1987.

factors, A=LU, are illustrated with the help of Fig.8.3.1,2. The system shown in Fig.8.3.1 consists of two 2-d transducer arrays whose drivers accept their input from the output of the detector-shift register plane, after some processing which is later explained. Between each two planes there is an imaging system, i.e. a system of lenses that allows the proper mapping of the light beams onto the transducer pixels, the corresponding detectors, and then the appropriate shift register. Thus, we can say that the transducer arrays and the detector-shift register array have identical pixel arrangement although not the same interconnections between pixels.

The elements of matrix A enter the shift register array in the same order as in the hex-connected VLSI systolic design and move upwards every clock tick. In their route they are modified in accordance with the light exposure of the detector plane: for an element a ij in a certain pixel of shift register array in a certain time unit the modifithe cation is $a_{ij} = a_{ij} + c$, where c is a quantity proportional to intensity of the incident light beam collected by the the corresponding detector. Then the modified element moves to the next pixel of the shift register array; the movement of all information in the optical processor is synchronised by a clock.

Finally, after a number of modifications, the elements of matrix A are produced in the output as the entries of



Fig.8.3.1. Optical processor for LU decomposition.



TRANSDUCERS

DETECTORS SHIFT REGISTERS

Fig.8.3.2. Details of optical processor.

- 459 -

matrix U; the same output, augmented with the modified elements of matrix A that are also involved in the calculation of the coefficients of matix L, are sent to the transducer drivers (see Fig.8.3.1). The division (reciprocal) and subtraction (negation) operations that are required are not readily realisable with optical techniques. These operations are performed by conventional preprocessing elements attached the transducer planes. Since the entries of L to are produced after these calculations the output for matrix L is collected from the transducer planes.

The derivation of the optical system from the corresponding systolic array is illustrated in Fig.8.3.2 for the case of tridiagonal systems. The relative directions of the data streams must be maintained in the optical implementation, although all data cannot move in the same plane any The operations performed by each part of the optical more. system are also indicated in the same figure: note that only pixel operates as a pure IPS processor. In the general one case of a banded matrix with bandwidth w = p+q-1 we have (p-1)x(q-1)pixels performing IPS computations; p-1 cells perfoming negations, i.e. producing -u_{ij}; q-1 cells performing simple multiplications, i.e. producing the elements of matrix L, and finally one cell producing u_{ij}^{-1} . Therefore, the main optical processing with regard to the IPS computation through the transducers and the accumulation of the results onto the shift registers is concentrated on (p-1)x(q-1) pixels.

- 460 -

A schematic overview of the optical processor for the LU decomposition of a tridiagonal matrix is shown in Fig.8.3.3. The transducers are assumed coplanar for clarity - similarly the detectors and the shift registers; no imaging system is shown. Notice that, in the shift register array, the off-diagonal elements of matrix A are not involved in an IPS computation and therefore they pass the plane with no delay; however, in order to keep the output data stream exactly the same as in the VLSI array, a delay is introduced for the off-diagonal elements of matrix U.

Thus, the optical system of Fig.8.3.1-3 can perform the systolic LU decomposition algorithm in the same number of steps as the corresponding VLSI array. The time unit of the optical processor is equal to the time necessary for the completion of the longest operation, i.e the reciprocal computation, plus the data transfer time.

The same optical processor, with only a small modification, can be used for the implementation of the R+F LU decomposition method, as shown in Fig.8.3.4. In the first step, a_{11} enters the detector array and in the second step it passes through unchanged as u_{11} ; simultaneously u_{11}^{-1} is calculated and a_{55} enters the detector array. In the third step, a_{21} and a_{12} are available and $l_{21}=a_{21}u_{11}^{-1}$, $-u_{12}=-a_{12}$ are calculated. The output of u_{12} is delayed for the next cycle and $u_{55}=a_{55}$ is produced. In the fourth cycle a_{22} , a_{45} and a_{54} enter the detector array and simultaneously l_{21} and u_{12}

- 461 -



Fig.8.3.3. Optical processor for LU decomposition of a tridiagonal matrix.



Fig.8.3.4. Operation of optical processor for R+F LU decomposition.



Fig.8.3.4. (continued)

ł

are produced; l_{45} and $-u_{54}$ are calculated and the transducer plane produce light beam with intensity $l_{21}(-u_{12})$ which is added to a_{22} to produce $a_{22} = a_{22} + l_{21}(-u_{12})$.

The modification that is necessary so that the R+F method can be accommodated regards the final steps of the computation. Then, the confrontation of the two LU decomposition streams is resolved by a double modification of the central element of matrix A, i.e. for our example a_{33} . For this reason, the contents of the appropriate register or pixel must be delayed and kept for two time units.

8.3.2 OPTICAL SOLUTION OF TRIANGULAR SYSTEMS

The optical processor implementing the systolic algorithm for the solution of the lower triangular systems of the general form $A\underline{x} = \underline{b}$ is shown in Fig.8.3.5, based on a processor proposed in [50]. Similar results can be derived for upper triangular systems.

The processor in Fig.8.3.5 consists of two 1-d transducers and an 1-d detector-shift register array. The inputs to the driver of the first transducer vector are the offdiagonal elements of matrix A, while the elements of the main diagonal provide the input to the driver of the second transducer, after the necessary preprocessing, which consist of subtraction and division operations. The processor performing these operations also accepts input from <u>b</u> and the output of the shift register array (indicated as <u>y</u>).



Fig.8.3.5. Optical processor for triangular system solution.

465 -

Т

The shift register array accumulates the charges that are produced by the detectors in proportion to their light exposure. The light beams reaching the detector array have been modulated by the two transducers so that an optical multiplication is achieved. The optical IPS is concluded by the accumulation of the charges on the shift registers, exactly as already described for the LU decomposition optical processor. The time unit is determined by the longest operation, i.e the sequence of subtraction and division. In general, the 1-d transducer and detector vectors have length q-1 pixels, where q is the bandwidth of the system.

For the bidiagonal linear systems produced by the LU decomposition of a tridiagonal matrix, the optical processor is given schematically in Fig.8.3.6. Since q=2, single-pixel devices are sufficient.

The same processor, with only minor modifications can be used for the R+F method: the operation of the system is illustrated in Fig.8.3.7. In the first step, x_1 is produced as a_1 , b_1 enter the subtraction and division processing element; in the second step x_5 is produced in a way similar to that of x_1 while the product $a_{21}x_1$ is accumulated onto y_2 ; in the third step x_1 is the output, the product $a_{54}x_5$ is accumulated on x_4 and x_2 is calculated from b_2 , a_{22} and y_2 .

The modification required for the optical processor to accommodate the R+F method is similar with that of the LU decomposition processor. When the central element, in our



Fig.8.3.6. Optical processor for bidiagonal system solution.







Fig.8.3.7. Optical processor for R+F triangular system solution.
example x_3 , enters the optical processor, it has to be kept for two cycles in the subtraction-division processing element so that a double modification occurs.

8.4 OPTICAL SYSTOLIC ALGORITHMS USING OUTER PRODUCTS*

8.4.1 BANDED MATRIX MULTIPLICATION

The common approach to the mmm operation, of the form AB=C, is to define each element of the output matrix C as an inner product between a row of A and a column of в. An alternative way is to see matrix C as a summation of matrices formed by outer products between columns of A and rows of В [13]. This second approach is exemplified in Fig.8.4.1 for two (3x3) matrices A, B: the multiplication is realised in n (here n=3) steps. In the first step the first column of A and the first row of B produce matrix F_1 ; in the second step, the second column of A and the second row of B produce matrix F₂ that is added upon F₁. Thus, after n steps, $C = F_1 + F_2 + ... + F_n$.

In Fig.8.4.2 the first two steps of a mmm for two (nxn) banded matrices A, B are shown. Applying the outer product concept, it is observed that the lengths of the row-column vectors involved in the computations are now fixed to w_A and w_B , where w_A and w_B are the bandwidths of matrices A, B respectively (here $w_A = w_B = 3$). Similarly, the entries of the output matrix C that are affected in any particular step of

^{*} Sections 8.4 and 8.5 form the basis of two papers presented in the 4th Int. Symposium on Optical and Optoelectronic Applied Science and Engineering, The Hague, The Netherlands, April 1987, and in 7th World Congress of Cybernetics and Systems, London, Sept. 1987.



Fig.8.4.1. Matrix multiplication using outer products.

471



Fig.8.4.2. Banded matrix multiplication using outer products.

- 472

1

the computation are located within an 'active window' of size $w_A w_B$. The computation is again completed in n steps. Notice that now the 'active window' is not static, as in the case of full matrix computations, but it moves along the diagonal, at the rate of one matrix element per mmm step. Notice also that the superscripts in Fig.8.4.2. indicate the number of outer products that are accumulated in an entry of matrix C.

The outer product optical processor is illustrated in Fig.8.4.3, and its operation will be described in the context of mmm. The optical processor consists of two 1-d transducers and a 2-d detector [13]. The imaging system is not shown in Fig.8.4.3 for clarity: spherical and cylindical lenses are placed in front of each 1-d transducer for beam shaping and special optic systems are placed between the two transducer vectors and between the second transducer vector and the 2-d detector array to achieve the beam directions shown in Fig.8.4.3. The length of the 1-d transducers is n pixels, and the detector array has area n^2 pixels, where n is the size of the matrix (here n=3).

The outer product calculation between the first column of matrix A and the first row of matrix B (see Fig.8.4.1) is illustrated in Fig.8.4.3. The outer product multiplications are performed by means of modulating the intensity of the light beams that travel in the directions shown by the arrows. For example, the beam with intensity proportional to





£

 a_{31} passes through the transducer pixel which modulates its intensity in proportion to b_{11} , yielding a beam with intensity proportional to $a_{31}b_{11}$. Thus, the detector array collects and stores in its pixels charges that correspond to the values of the entries of matrix F_1 . In the next step, the 1-d transducers are loaded with the second column of matrix A, and the second row of matrix B. Thus, the outer product processor will form matrix F_2 which will be added in the detector array onto matrix F_1 . Finally, after n outer products, matrix C is stored in th detector array.

The optical processor for mmm of banded matrices using outer product is shown in Fig.8.4.4. The following modifications can be observed in comparison with the optical processor of Fig.8.4.3. Since only the 'active window' elements, for all matrices, need to be present at any step of the computation, the 1-d transducers have now lengths of w_A and w_B pixels respectively, instead of n (see Fig.8.4.2). Similarly, the detector array is of area $w_A^{}w_B^{}$ pixels instead of n^2 pixels. The detector array is now augmented with a plane shift registers so that the resulting 'active window' of of matrix C can be moved along the detector array as indicated Fig.8.4.2. Thus, the elements of matrix C are not stored in in the detector array but are produced systolically as an output of the shift registers.

The operation of the optical processor, based on the matrix of Fig.8.4.2 is illustrated in Fig.8.4.5. Each full

- 475 -







		•	
	^b 32	^b 33	^b 43
		_	
a ₄₃	c ⁽³⁾ 22	c ⁽²⁾ c ²³	c ⁽¹⁾ 24
^a 33	c ⁽²⁾ 32	c ⁽²⁾ 33	c ⁽¹⁾ 34
a ₂₃	$c_{42}^{(1)}$	$c_{43}^{(1)}$	$c_{44}^{(1)}$

1: shift in new row-column vectors shift out matrix C entries 2: perform outer product and addition.

Fig.8.4.5. Phases of a matrix multiplication.

477 -

mmm step can be divided into two phases. In the first phase, the 'active window' elements of a column of matrix A, and the 'active window' elements of a row of matrix B enter the 1-d transducers. Simultaneously, the contents of the shift registers move in the direction shown by the arrow, so that $w_C = w_A + w_B - 1$ entries of matrix C are produced. In the second phase, the outer product computation takes place and the result is accumulated on the detector-shift register plane.

Now the optical processor is ready to start a new mmm step, i.e to shift out another 'active window' row and column of matrix C, load the entries of matrices A, В and restart the calculations. The computation is completed in $n+min(w_{A},w_{B})$ time units, where a time unit is defined as the time necessary for a full mmm step to be performed. Notice the analogy of the calculations performed by the outer product processor with the optical processors described in section 8.2, and subsequently, the unidirectional VLSI mmm array in chapter 7. Thus, all the results concerning the multiplication of matrices with unequal semibands, as well full mmm computations, can be readily applied on the as outer product processor.

In comparison with the outer product processor discussed in [13], the proposed system reduces significantly the hardware requirements especially when $n >> w_A, w_B$; furthermore the system is size-independent, i.e it works for any n given a maximum bandwidth; finally, no unloading cycles are

- 478 -

necessary at the end of the computation. On the other hand, some additional hardware complexity is introduced since the detector array must be coupled with a shift register plane to accommodate the movement of the elements of matrix C.

8.4.2 BANDED MATRIX LU DECOMPOSITION

The LU decomposition of a matrix A can be expressed as a series of elementary row operations in which multiples of a row are subtracted from all the rows beneath it to generate zeros below the main diagonal of A. In general, a (nxn) matrix requires (n-1) complete steps to be transformed to an upper triangular form U; the row multipliers involved in each step with opposite sign also form the corresponding lower triangular matrix L.

This procedure can be carried out by using the matrix multiplication operation as illustrated in Fig.8.4.6 for a (3x3) matrix [28]. Matrix E_1 of the multipliers is multiplied with $A^{(1)}=A$ to give $A^{(2)}$ where the elements of the first column below the main diagonal are eliminated. By repeating the same procedure: $E_2A^{(2)} = A^{(3)}$ and the elements of the second column are now eliminated.

It is noticed that E_j can be written as an identity matrix I, plus a matrix E_j^* which has only one non-zero column. Therefore

$$A^{(j+1)} = E_{j}A^{(j)} = (I+E_{j}^{*})A^{(j)} = A^{(j)}+E_{j}^{*}A^{(j)}$$
(8.4.1)

 $\begin{bmatrix} 1 & 0 & 0 \\ e_1 & 1 & 0 \\ e_2 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} & a_{13}^{(1)} \\ a_{21}^{(1)} & a_{22}^{(1)} & a_{23}^{(1)} \\ a_{31}^{(1)} & a_{32}^{(1)} & a_{33}^{(1)} \end{bmatrix} = \begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} & a_{13}^{(1)} \\ a_{11}^{(1)} & a_{12}^{(1)} & a_{13}^{(1)} \\ 0 & a_{22}^{(2)} & a_{23}^{(2)} \\ 0 & a_{32}^{(2)} & a_{33}^{(2)} \end{bmatrix}$ A⁽¹⁾ _م(2) Е, $e_1 = -\frac{a_{21}^{(1)}}{a_{11}^{(1)}}$, $e_2 = -\frac{a_{31}^{(1)}}{a_{11}^{(1)}}$. $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & e_3 & 1 \end{bmatrix} \times \begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} & a_{13}^{(1)} \\ 0 & a_{22}^{(2)} & a_{23}^{(2)} \\ 0 & a_{32}^{(2)} & a_{33}^{(2)} \end{bmatrix} = \begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} & a_{13}^{(1)} \\ a_{11}^{(1)} & a_{12}^{(1)} & a_{13}^{(1)} \\ 0 & a_{22}^{(2)} & a_{23}^{(2)} \\ 0 & 0 & a_{33}^{(2)} \end{bmatrix}$ (2) م A⁽³⁾ E, where $e_3 = -\frac{a_{32}^{(2)}}{a_{32}^{(2)}}$ $\begin{vmatrix} 1 & 0 & 0 \\ k_{21} & 1 & 0 \\ k_{21} & k_{22} & 1 \end{vmatrix} = \begin{vmatrix} 1 & 0 & 0 \\ -e_{1} & 1 & 0 \\ -e_{2} & -e_{2} & 1 \end{vmatrix}$ L $\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix} = \begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} & a_{13}^{(1)} \\ 0 & a_{22}^{(2)} & a_{23}^{(2)} \\ 0 & 0 & a_{33}^{(2)} \end{bmatrix}$ U

Fig.8.4.6. LU decomposition using outer products.

- 480 -

Since E_1^* has non-zero entries only in the first column the matrix product $E_1^*A^{(1)}$ will consist only of one outer product between the first column of E_1^* and the first row of $A^{(1)}$. The second step will consist of calculating the outer product between the second column of E_2^* and the second row of $A^{(2)}$ and adding the resulting matrix to $A^{(2)}$ to generate $A^{(3)}=U$. Matrix L contains a record of the elementary row operations, i.e. the jth column of L is essentially the jth column of E_1^* , (see also section 2.3).

The n-1 outer products required for the LU decomposition of a full matrix are calculated between vectors of length n, n-1, ..., 2. Thus, the entries of the matrix that take part in a step of the LU decomposition are located in an 'active window' of size nxn, (n-1)x(n-1), ..., 2x2 respectively. Furthermore, the first row and the first column of the 'active window' are known in advance since the first row is not altered and the remaining entries of the first column are reduced to zero. Therefore, the lengths of the vectors that participate in the modification of A are reduced to n-1, n-2, ..., 1 and similarly the size of the 'active window' is also reduced.

In Fig.8.4.7 the first two steps of the LU decomposition procedure for banded matrices is illustrated for a quindiagonal matrix (p=q=3). Applying the ideas discussed previously it is observed that the lengths of the vectors involved in the outer product calculation are now fixed to

'11 | [∽]12 `-l₂₁₁ (1)^a21 0 -¢ 31 1 ^a 31 ^a32 ^a33 ^a35 a (1) 32 о = 1 a₄₂ a₄₃ a₄₅ 1 ^a11 (1) a22 11 0 = ^a34 ^a35 0 0 a 0 1 a₅₅

Fig.8.4.7. Banded LU decomposition using outer products.

482

p-1, q-1 and the size of the 'active window' is also fixed to (q-1)x(p-1). This means that only a fixed number of entries need to be present at a given step of the computation. Notice that in Fig.8.4.7 the superscripts indicate the number of modifications each element undergoes.

The outer product calculation between the first column E_1^{\star} and the first row of $A^{(1)}$ (see Fig.8.4.6) is illustrated in Fig.8.4.8. The 2-d detector array contains the components of matrix $A^{(1)}$. The outer product multiplications are performed by means of modulating the intensity of the that travel in the directions shown by the light beams arrows. For example, the beam with intensity proportional to e, passes through the transducer pixel which modulates its intensity in proportion to $a_{11}^{(1)}$, yielding a beam with intensity proportional to $e_2 a_{11}^{(1)}$. The detectors accumulate a charge that corresponds to $e_2a_{11}^{(1)}$ onto the already existing charge $a_{31}^{(1)}$ to produce the new charge that corresponds to element $a_{31}^{(2)}$ of matrix $A^{(2)}$, i.e. zero. In this way we have: $E_1^{*}A^{(1)} + A^{(1)} = A^{(2)}.$

The schematic diagram for the outer product optical processor for implementing the LU decomposition algorithm is shown in Fig.8.4.9 [13]. The coefficient matrix A is first loaded into the detector-shift register array. The detector array then shifts its contents along the row and the column direction as indicated by the arrows. The top row is the first row of matrix U, whereas the first column is used in





1-d input transducers

2-d detector array

Fig.8.4.8. LU decomposition using outer product.

- 484 -





Fig.8.4.9. Optical processor for full matrix LU decomposition.

calculating the first column of matrices E_j^* and L (see Fig.8.4.6).* An outer product is then calculated between the first column of E_j^* and the first row of U and added to the shifted matrix A. This process is repeated n-1 times to generate matrix U one row at a time and matrix L one column at a time.

An optical processor for the LU decomposition of banded matrices is shown in Fig.8.4.10. The following modifications can be observed in comparison with the optical processor of Fig.8.4.8. No preloading of the whole matrix A is necessary; only the 'active window' elements need to be present plus one element from the uppermost and lowermost diagonals. This means that there is no preloading delay and the number of pixels is reduced to (q-1)x(p-1)+2. The shift register plane now moves its contents in one direction only, yielding a simpler detector array design. The 1-d transducers are now of length (q-1) and (p-1) pixels respectively.

The operation of the optical processor, based on the matrix of Fig.8.4.7 is illustrated in Fig.8.4.11. Each full step of the LU decomposition algorithm has three phases. In phase 1 the elements of matrix A are shifted as indicated and the outgoing entries form a row of U and a column of L: e.g $[a_{11} \ a_{12} \ a_{13}]$ form the first row of U and $[a_{11} \ a_{21} \ a_{31}]^T$ are involved in the calculation of the first column for L, E_1^* . The second phase calculates the coefficients for L, E_1^* .

^{*}The computation of the columns of E^{*} and L is performed using conventional electronic components for subtraction and division.









for L,E

1: shift elements of A

2: calculate L,E columns



3: perform outer product and addition

Fig.8.4.11. Phases of a full step of LU decomposition.

- 488 -

to the existing matrix $A^{(1)}$: e.g. $a_{22}^{(1)} = a_{22}^{-1}a_{12}^{-1$

Now the processor is ready to shift out another row of and another column for L and to restart the calculations. U The computation is completed in $n+\min(p-1,q-1)$ time units, where a time unit is defined as the time necessary for a full step of the LU decomposition to be performed. This includes one optical IPS computation, one subtractiontransfers. Notice division and the data that the subtraction-division processing element must be capable of performing q calculations in parallel, while in the case of the LU decomposition optical processor in section 8.3 no such facility is required.

The LU decomposition outer product processor in Fig.8.4.10 can be seen as the mmm outer product processor of Fig.8.4.3 augmented with the feedback mechanism, the processing element performing subtractions-divisions and two additional pixels for the detector-shift register array to act as delays for the uppermost and lowermost diagonal entries of matrix A.

In comparison with the outer product processor discussed in [13], the proposed system reduces significantly the hardware requirements especially when n>>p,q; furthermore the system is size-independent, i.e. it works for any n and for given maximum bandwidth; the proposed system is also faster as no preloading cycles are necessary.

- 489 -

8.5 OPTICAL GAUSS ELIMINATION USING OUTER PRODUCTS

The outer product processor for banded matrix LU decomposition proposed in section 8.4 is now utilised for the implementation of several Gauss elimination (GE) algorithms, i.e matrix triangularization; direct solution of linear sytems; matrix inversion and solution of matrix equations. The quindiagonal matrix is again used as an example (p=q=3, n=5).

The triangularization of a matrix is illustrated in Fig.8.5.1; with the help of this figure the Gauss elimination procedure performed with outer products is explained. is shown in Fig.8.5.1 the GE algorithm can be performed As in n-1 major steps, each step involving an 'outer product plus addition' computation. For example, in the first step [a₁₁ a₂₁ a₃₁]^T determines the multipliers of GΕ [1 e₂₁ e₃₁]^T; then an outer product is calculated between $\begin{bmatrix} 1 & e_{21} & e_{31} \end{bmatrix}^T$ and $\begin{bmatrix} a_{11} & a_{12} & a_{13} \end{bmatrix}$ and the resulting matrix is added to matrix A to produce the modified matrix for the second step. The actual modification takes place in the of size (q-1)x(p-1)'active window' as indicated in Fig.8.5.1. The GE moves diagonally from the top of the matrix towards its bottom and produces a triangular matrix U; the elements of the matrix that take part at any step of the computation are shown in Fig.8.5.1 and the locality of the transformations of GE is illustrated.

A further final step of the triangularization algorithm

can be the division of the elements of each row with the diagonal element of the row, as shown in Fig.8.5.1. Thus the main diagonal entries are all set to 1. Although this step can again be analysed in n outer products, it is more convenient to have a different implementation as will be explained later on. The triangularization method is the same with the LU decomposition method with the difference that no record of the multipliers is kept and therefore no matrix L is produced.

The algorithm described can be readily modified to solve a system of simultaneous linear equations based on GE and the use of augmented matrices. For the system Ax=b, where A is as defined in Fig.8.5.1 the augmented matrix is [A|b]; it is possible using elementary row operations to transform [A|b] to [U|b'] where U is upper triangular and solve the system with a back substitution. The method is illustrated in Fig.8.5.2; for the sake of simplicity all non-zero entries are indicated with 'x'; again the 'active window' and the elements involved in each computation step are indicated in the figure. A final division step can be included, where each row of b is also divided by the diagonal element of the corresponding row of matrix A.

The GE algorithm for the direct solution of a linear system is extended to the Gauss-Jordan (GJ) method at the expense of n more cycles as shown in Fig.8.5.3. The superdiagonal elements are now eliminated using the 'outer product

- 491 -

- 492 -



Fig.8.5.1. Triangularization of A.

plus addition' pattern. The computation continues beyond the end of the GE algorithm and starts from the bottom and moves towards the top of the matrix.Since the lower diagonal entries are all eliminated by GE, there are no modifications in matrix A only eliminations, while there are still modifications for <u>b</u>.

In both the algorithms described for the solution of 'active windows': one for linear systems there are two matrix A and one for b. Note that there is some redundancy in the calculations for the GJ method. After the first division step (Fig.8.5.2) the main diagonal entries are all 1 and therefore the 'outer product plus addition' is used only for b; furthermore the final division step (in Fig.8.5.3) is not necessary. Alternatively the first division step can be omitted and the remainder of the computation is then carried out exactly as shown in Fig.8.5.3.

The GJ algorithm can be used for the inversion of a matrix A if the method is applied on the augmented matrix [A|I], where I is the identity matrix. At the end of the computation the augmented matrix is transformed to $[I|A^{-1}]$. The matrix inversion algorithm is shown in Fig.8.5.4. The operations in the left-hand side matrix A are identical to those of Fig.8.5.2,3. The calculations in the right-hand side matrix are slightly different from the operations discussed up to now. It is obvious from Fig.8.5.4 that the profile of matrix A is not preserved in A^{-1} and therefore the

x	x	x			
x	×	X	x		
x		ÿ	x	x	
	x	x	x	×	
		x	x	×	

x	x	x			
	x	x	×		
	x	X,	×	x	
	x	×	X	x	
		x	x	x	



x x

x

×

x

x

x	x	x		
	x	x	×	
		x	x	x
		x		×
		×	×/	

xx

x



x

x



х

n-1 modification steps

x	x	x			
	x	×	×		
		x	x	×	
			×	×	
				×	



Fig.8.5.2. Gauss Elimination for A, b.

x

x

x

 x
 x
 x

 x
 x
 x

 x
 x
 x

 x
 x
 x

 x
 x
 x

 x
 x
 x

 x
 x
 x

 x
 x
 x

 x
 x
 x

 x
 x
 x

 x
 x
 x

 x
 x
 x

 x
 x
 x

x x

x



x



x	x	x		
	x	×		
		x		
			x	
				×





n-1 modification steps

 x
 x

 x
 x

 x
 x

 x
 x

 x
 x

 x
 x

 x
 x

 x
 x

 x
 x

 x
 x

 x
 x

 x
 x

 x
 x

 x
 x

 x
 x

 x
 x

 x
 x

division

Fig.8.5.3. Back substitution for A, b.

- 496 --

x x x / } x x / Ĺ x х х х x х x x x x

1				
\square	1			
		1		
			1	
				1

x	x	x		
	x	×	×	
	x	× /	\backslash	×
	x	X	\x>	×
		x	x	x

1				
x	1			
X	\mathbb{N}	1		
	\mathbf{N}		1	
				1

x	x	x		
	x	x	x	
		x	° X	x
		x		X
	_	x	7	

1				
x	1			
x	×	1		
X	X	\mathbb{N}	1	
\mathbb{N}	\setminus	\mathbb{N}		1

 x
 x
 x

 x
 x
 x

 x
 x
 x

 x
 x
 x

 x
 x
 x

 x
 x
 x

 x
 x
 x

 x
 x
 x

1				
x	1			
x	x	1		
x	x	x	1	
×	×	X	\mathbb{N}	1

n-1 modifications





division

Fig.8.5.4(a). Gauss-Jordan method for matrix inversion.

- 497 -







1				
×	\checkmark	\square	\square	\square
×	×	X	X	x
x	x	x	x	x
x	x	x	x	1

x	x	x		
	x	×		
		x		
			x	
				x

\checkmark				\square
×	×	×	X	1
x	x	x	x	×
x	x	x	x	×
x	x	x	x	1



	×	×	×	X
x	x	x	x	×
x	x	x	×	x
x	x	x	x	x
x	×	x	×	1

n-1 modifications



x	x	x	x	×
x	x	x	x	x
x	x	x	x	×
x	×	x	x	×
x	×	x	x	1

division

Fig.8.5.4(b). Gauss-Jordan method for matrix inversion.

concept of the 'active window' cannot be applied in the same way on A^{-1} . This is more clear in Fig.8.5.4(b) where in each computational step the nx(q-1) elements of matrix A^{-1} are modified. Therefore, the concept of the 'active window' can be applied only in one dimension of A^{-1} ; in other words matrix A^{-1} can be viewed as n columns, each one representing a 'vector <u>b</u>' of the GJ algorithm in Fig.8.5.3. It should be noted that since the 'active window' for A^{-1} has now size nx(p-1) or nx(q-1), the matrix inversion algorithm is not size-independent in the sense of the algorithms previously discussed.

The division steps in Fig.8.4.4 introduce the same redundancy as in the GJ algorithm for the solution of the linear systems of equations; again one of the two division steps can be omitted. The same algorithm can be used for the solution of a matrix equation AX=Y, if the augmented matrix takes the form [A|Y], so that after the computation we have $[I|A^{-1}Y]$.

8.5.1 OPTICAL IMPLEMENTATION

The basic component for an optical implementation for all the algorithms described, is the optical processor shown in Fig.8.4.10, configured for the case of our example matrix. The calculations performed by the optical processor are detailed in the first four steps of Fig.8.5.1. If the final division step is necessary as well, then the optical processor can be expanded as shown in Fig.8.5.5: each row of



Fig.8.5.5. Optical processor for matrix triangularization.



Fig.8.5.6. Optical processor for Gauss Elimination.

•

.

- 499 -

the triangular matrix produced as described before is now divided by its first element, thus giving a triangular matrix with all main diagonal elements equal to 1. No significant additional delay is introduced since the division can be performed in parallel with the calculation of the multipliers; thus only the last step creates a delay. The hardware for this additional processing element is a simpler version of that needed for the subtraction-division processing element.

Therefore, the triangularization of a banded matrix with bandwidth w=p+q-1 using GE without pivoting, can be performed in n+min(p-1,q-1) major steps, on an optical processor as that shown in Fig.8.5.5. The detector array has size (q-1)x(p-1)+2 and the transducers (q-1) and (p-1) pixels. One (or two) processing elements are necessary performing subtraction and division (or division only); these processing elements must be capable of performing p (and q) calculations in parallel.

The same optical processor, with some extensions, can be used for the direct solution of a system of linear equations based on GE and the use of augmented matrices. The system is shown in Fig.8.5.6. The 'outer product plus addition' operation is now extended to include <u>b</u>, as explained with the help of Fig.8.5.2. This is achieved by means of a second outer product processor working in parallel with the one already discussed; the column of the multipliers is the same for both outer products. The computation follows the same pattern, as described for the matrix triangularization. Notice that q-1 initialisation steps are required, so that the elements of <u>b</u> are loaded in the detector array; the input sequence of matrix A must be modified accordingly.

Therefore the direct solution of a banded linear system of equations with semibandwidths p,q can be performed in n+(q-1) major steps on an optical processor as that of Fig.8.5.6. The detector arrays have sizes (q-1)x(p-1)+2 and q-1 pixels; the transducers have lengths q-1, p-1, and 1,q-1pixels. The subtraction-division processing element performs q calculations in parallel while the additional division element performs p+1 calculations in parallel.

The GJ algorithm for the solution of a linear system can be optically implemented by means of the same basic component of Fig.8.5.6. The two phases of the algorithm, as shown in Fig.8.5.2,3 can be performed by the same optical processor provided that in the second phase the triangular matrix U is transposed; both U and b' are processed from the bottom towards the top during the second phase of the algorithm. A system based on the optical processor of Fig.8.5.6 is shown in Fig.8.5.7: A and b enter the system and they are routed to the optical processor; the intermediate results, i.e. U and b' are stored into a LIFO memory module; for the second phase of the algorithm the transposed matrix U and b' enter the optical processor in the reverse order and the



Fig.8.5.7. Optical processor system for Gauss-Jordan method.

resulting vector \underline{x} is produced as output of the system.

Therefore the GJ algorithm can be performed in 2(n+max(p-1,q-1)) major steps on an optical processor as in Fig.8.5.7. The maximum semibandwidth is used as the basis of the design, so that both phases can be performed on the same optical processor. The redundancy of the computations can be avoided if a special optical processor for the second phase is used, since no division is necessary and no output for U is produced.

The GJ method can be extended for matrix inversion if <u>b</u> is replaced by the identity matrix. All the transformations on the original matrix A are recorded on the identity matrix to produce A^{-1} , as shown in Fig.8.5.4. As it is explained, the concept of the 'active window' has only partial application on A^{-1} ; thus, for the optical implementation of the matrix inversion, the optical processor is extended as shown in Fig.8.5.8. The outer product processor for the modification of <u>b</u> is expanded so that it can modify nxmax(p-1,q-1)elements of the matrix A^{-1} simultaneously. Thus the optical implementation of the matrix inversion is size-dependent.

The overall configuration of the optical system for matrix inversion is exactly the same as that of Fig.8.5.7, if $\underline{b}, \underline{b}', \underline{x}$ are replaced by I, $(A^{-1})', A^{-1}$. Similarly the redundancy of the calculations can be removed by a special optical processor for the second phase of the algorithm.



.

Fig.8.5.8. Optical processor for matrix inversion.
The solution of the matrix equation AX=Y can be effected in the same optical processor as for matrix inversion, if I is replaced by Y.

8.6 CONCLUSIONS

Although the optical operations described in the previous sections can be initially applied only on non-negative real numbers, there are well-known methods to accommodate the full range of real numbers as well as complex arithmetic: see for example [58], [222], [292].

The limited accuracy of the basically analog optical computations that are discussed in this chapter can be increased by means of the Digital Multiplication via Analog This algorithm is outlined, Convolution (DMAC) method. amongst others, in [241], and is based on the convolution of two binary encoded input signals, to produce an output signal in mixed binary form that can then be transformed into pure binary arithmetic. The output signal is equivalent to the product of the input signals. This process is equivalent to the polynomial multiplication algorithm, discussed in section 3.1, where the a's and b's are all 0 or 1, that **SO** each polynomial represents a binary coded number, in integer or fixed-point real arithmetic. The product of the two polynomials can be seen as the result of the multiplication of these two numbers, in mixed binary form. It is possible to convert this result in purely binary notation with some simple postprocessing.

A survey of some of the methods proposed for the implementation of the DMAC method is given in [49]. The algorithm has been applied on the outer product processor in [12] and experimental results are given in [14]. The DMAC algorithm has been extended to fixed-point arithmetic in [29], [31]. A similar implementation has been reported for integratedoptical processors in [11], [291].

Thus, each of the optical systolic algorithms can be extended to include digital accuracy, if the entries of the matrices or vectors involved are binary encoded and the DMAC algorithm is incorporated on the optical processor.

However, it is argued that this straightforward extension leads to excessive requirements, as regards the number of pixels of the transducer and detector arrays. Some other difficulties that are introduced include the loss of the speed of the optical computation and the accuracy required in the Analog-to-Digital conversion [10], [235]. Nevertheless, the DMAC method has received considerable attention and some encouraging experimental results have been reported [31], [54], [67], [75].

Two other interesting problems arising from the optical systolic algorithms discussed herein are the feedback mechanisms as well as the combination of electronic and optical processing elements. Given the speed of the optical IPS computations, the feedback mechanisms should be equally fast, if they are not to introduce any delay in the computations. Thus, optical feedback might be considered, using some of the optical signal transmission methods discussed in section 3.5, although it requires additional signal conversions.

The combination of conventional digital electronic processing elements with optical components has been advocated so that computations not readily realisable in optical computing can be performed. This operations include all non-IPS calculations, as shown in sections 8.3-8.5. This hybrid processor, however, forces a pipelined computation to be performed at the speed of the slowest component, i.e. the digielectronic processing elements. Thus, an interesting tal field for further investigation, is the development of parallel algorithms for hybrid procesors, where the arrangement of the computations is such that to minimize the delay introduced by the non-linear (conventional) components.

Some alternative, all-optical implementations of the basic unary operations have been proposed in [290], [292], based on the concept of an optical polynomial evaluation processor. However, for the current state of development of optical computing, the realisation of all-optical digital (or analog) computers is only experimental [10], [35], [85], [206]. Therefore, the combination of optical and electronic components seems to be the next step forward for optical computing, either in the form of optical interconnections, or in the form of hybrid processors.

One way to reduce the need for complex non-IPS calculations that may cause long delays, is the expression of the algorithms to be implemented as a series of simple mvm or

^{*} This technique may be the cause of potential problem, in power budgets, if the bandwidth of signals is great.

mmm computations. This technique has been used in the case of direct methods, as is shown in the previous sections, but it mainly favours iterative algorithms as very good candidates for optical implementation.

Optical iterative systolic algorithms using a series of mym computations have been extensively discussed for the solution of linear systems of equations [50-54]; and for eigenvalue-eigenvector computation [56]. Following this line of thinking, it would be very interesting to investigate the design of optical systolic pipelines, as well as optical systolic iterative arrays, implementing the algorithms proposed in chapter 6 for VLSI systolic networks. One obvious advantage of the optical systolic networks would be the elimination of the interconnection difficulties. Thus, an intermediate approach can be the design of 3-d systolic networks with VLSI mvm arrays and optical interconnection components.

Optical systolic iterative algorithms based on mmm operations are discussed in [56], where each mmm is decomposed as a series of mvm's. The optical realisation of the systolic array designs for successive mmm's, proposed in chapter 7, will allow for the direct implementation of all the algorithms discussed in this chapter.

Finally, an important aspect for the development of optical systolic algorithms, and optical computing in general, is the actual testing and verification of the

- 509 -

algorithms and architectures, using simulation techniques: see, for example, in [2], [52], [84] for the use of simulation methods in the development of optical systolic concepts. The soft-systolic simulation technique offers a suitable framework for this process, as it is shown in the programs listed in the Appendix (sections A.5 and A.6).

In A.6.2 there are some pixel definitions of the basic components of an optical systolic processor, such as: light emitter pixel; light modulator pixel; detector and shift register pixel. These basic building blocks are used in the programs of A.5 to develop soft-systolic simulation programs some simple optical systolic algorithms. Two types of for banded mvm are presented (A.5.1 and A.5.2), one using outer product concepts and the other using simple space integraproduct). These architectures are tion (inner further extended to accommodate banded mmm, in A.5.3 and A.5.4. Finally, a possible implementation of the DMAC algorithm is given, where two binary-coded sequences are multiplied to produce a sequence in mixed binary form.

It would be interesting to further develop the softsystolic simulation method to fully accommodate optical computing concepts.

CHAPTER 9

CONCLUSIONS

This thesis has introduced some new systolic algorithms and architectures for numerical computation, under the framework of being suitable for implementation on to VLSI processor arrays, or optical processors.

This final chapter is structured as follows. Initially, a summary of the main topics of discussion is given, together with an overview of the related results, thus complementing the thesis organisation presented in section 1.3. Then, some general conlusions are given, partially in relation with the major areas of current systolic systems and computing research, discussed in section 1.2.

9.1 THESIS SUMMARY

The preceding chapters can be divided in three parts: the first part (chapters 1-2) consists of survey and background material for the ready comprehension of this work; the second part (chapter 3) combines an overview of basic concepts with the introduction of some more recent developments in the area of systolic computing; the third part (chapters 4-8) is devoted to the presentation of new systolic algorithms.

Chapter 1 gives a brief overview of the environment for the development of the systolic approach, by discussing: the applications of systolic systems; the advances of VLSI technology; and the relation of the systolic approach with other parallel processing methods and the theory of cellular automata. Further, an outline of the major areas of systolic systems research is presented, such as: implementation issues and optics technology; overall system design and programming; and algorithm design and applications. Finally, the organisation of the thesis is discussed.

Chapter 2 contains basic mathematical definitions in the areas of polynomial equations; matrix computations; linear programming; and differential equations. Then, there introduction to specific topics: solution of polynois an mial equations; solution of systems of linear equations; matrix eigenproblem solutions; matrix functions; and discrete approximation of differential equations. Finally, there is a brief discussion of the numerical algorithms used in subsequent chapters.

Chapter 3 starts with the description of a simple systolic algorithm, through which the basic definitions and terminology are introduced; then a framework for systolic algorithm development on VLSI is presented. This is followed

- 512 -

by the discussion of a set of fundamental systolic designs for: matrix-vector multiplication; matrix-matrix multiplication; and the solution of linear systems of equations. These designs are used as building blocks and benchmarks in the Next there is an overview of some subsequnent chapters. transformational techniques, for derivation and improvement of systolic designs: the retiming method; cut theorem; area-time expansion; and R+F method. Further, the importance of systolic programming and simulation is discussed and some special-purpose (Warp, WAP) and general-purpose (Transputer) 'systolic computers' are presented. Then, the soft-systolic paradigm is introduced; simulation, systolic programming and soft-sytolic approach form the basis for the development of soft-systolic simulation method, using OCCAM. Finally, the the optical implementation of systolic algorithms is investigated, as well as the use of optical signal transmission techniques. The chapter is concluded with a modification of framework initially proposed so that it can accommodate the the different classes of systolic algorithms: hard (tradidirect VLSI implementation); hybrid (programmable, tional, special-purpose VLSI processor arrays); soft (network of processes on a general-purpose parallel processing computer); and optical (analogue optical or hybrid electronicoptical processor implementation)

Chapter 4 investigates the systolic implementation of a new group of numerical algorithms, i.e. the solution of polynomial equations. Two traditional methods are discussed

- 513 -

detail: Graeffe and Bernoulli. In the study of these in methods transformational techniques are exemplified: retiming technique; bidirectional array transformed to ring architecture; area-time expansion. Further, a general svstolic ring architecture is proposed that can accommodate the majority of iterative methods for the solution of polynomial equations, with modifications in the number of parallel polynomial evaluations and in the calculation of the new approximation. Finally, the Sturm sequence property is used for the systolic computation of the roots of the characteristic polynomial of a tridiagonal matrix. The method can be extended to unsymmetric tridiagonal or guindiagonal matrices; moreover, the calculation of the characteristic equation of Hessenberg matrices is also possible, using a linear array, similar to that of a triangular system solution. Noticing that, the latter can also be transformed in a systolic ring, we can conclude that, it is possible to build a more general-purpose ring architecture for polynomial evaluation, solution of polynomial equations, as well as other polynomial computations.

Chapter 5 presents some systolic algorithms for the efficient solution of linear systems of equations, using LU decomposition. Initially, a mathematical transformation is introduced (block (2x2) R+F method) for improving the efficiency of the systolic algorithms for banded LU/LDU decomposition. Then, the problem of updating LU factors is discussed, in the context of the Simplex method; the same

- 514 -

approach can be used in other cases where updating is necessary, such as real-time signal processing and algorithmic extended to The same technique can be fault-tolerance. include pivoting, as well as other matrix factorization methods (e.g. QR decomposition). Further, it would be interesting to investigate the combination of the R+F method updating technique, and other factorization with the methods. Finally, the LU decomposition with partial pivoting is used in the systolic implementation of the Inverse Iteration method, for the determination of the eigenvectors of symmetric tridiagonal matrices. This array may be combined with the design determining the eigenvalues of the same type of matrices, discussed in chapter 4; thus, it is possible to produce a systolic system solving the eigenproblem for symmetric tridiagonal matrices which form the end-product of the eigenproblem solution of general symmetric matrices. As a conclusion, notice that the similarity of LU and QR decomposition methods (with or without pivoting) makes possible design of a programmable systolic array performing all the these types of algorithms, according to the problem at hand.

Chapter 6 develops the concept of combining systolic arrays for matrix-vector multiplication; this technique is applied to the iterative solution of linear systems of equations, usually occurring in the discrete approximation of ordinary and partial differential equations. Initially, retiming techniques are applied for the derivation of a unidirectional banded matrix-vector multiplication array. Further, the area-time expansion method is employed for the derivation of an iterative systolic array, suitable for full matrix computations. Thus area and/or time efficient pipeline and iterative designs are produced for different solution methods and techniques: Jacobi; Gauss-Seidel; JOR; SOR; cyclic reduction; multi-coloring technique. Finally, the concept of systolic networks is introduced, defined as parallel, co-operating pipelines of systolic arrays (area expansion) or co-operating systolic iterative structures (time expansion). An interesting development would be the integration of optical data transmission lines, to formulate 3-d 'prismatic' systolic networks, in order to overcome interconnection and communication problems. The importance of matrix-vector multiplication in scientific and signal processing applications, suggests a wide range of applications for the unidirectional, as well as the iterative, arrays discussed in this chapter; thus, a further development would be the comparison of their performance with that of existing arrays, in other specific applications.

Chapter 7 develops further the idea of combining systolic arrays to produce pipelines or iterative structures, following the reasoning of area-time expansion schemes. A new group of systolic algorithms is introduced, based on successive matrix-matrix multiplications. Previously, successive matrix multiplication algorithms had been considered as too expensive to implement, in terms of both storage and processing; further, they introduced scaling and

- 516 -

normalisation problems. However, the advances in technology possible the consideration of these algorithms that make introduce a high degree of parallelism, provided that scaling and normalisation problems are solved satisfactorily. Thus, matrix-matrix multiplication arrays are combined to form systolic pipelines or iterative structures, and used for the solution of a variety of problems: iterative solution of linear systems; eigenproblem solution, with Power, Matrix Squaring and Raised Power methods; matrix inversion; matrix polynomial evaluation; approximation of matrix functions, such as: exponential; square root; inverse square root; cosine and sine; and logarithm of a matrix. As a conclusion we should notice that the general significance of this group of algorithms has not been fully investigated, not only in the area of systolic computing, but also in the general field of large-scale scientific computation, where the 'matricialization' of the basic functional units and algorithms is also under consideration.

Chapter 8 presents some optical systolic algorithms, for fundamental matrix computations, such as matrix-matrix multiplication and LU decomposition. Initially, the direct mapping of VLSI algorithms on to optical processors is investigated, and a framework for generalising this technique is given. The implementation of the R+F method is also introduced to improve the efficiency of some optical systolic algorithms. Then, the Outer Product processor is presented and the potential of 3-d parallelism is exploited to produce optical systolic algorithms with higher performance and reduced hardware requirements. An Outer Product processor for banded matrix computations is defined, and used for the optical implementation of a wide range of sysalgorithms, mainly based on Gauss Elimination. tolic Finally, some of the problems of the optical implementation systolic algorithms are briefly discussed: first the of utilization of the Digital Multiplication through Analog Convolution (DMAC) algorithm, for acquiring digital accuracy in the computations; second the combination of electronic and optical components into hybrid processors. As a conclusion, we should point out that optical computing is an area that is by no means 'settled', either in terms of the underlying technology, or in terms of theory and applications, i.e. the potential of optical computing has not been fully understood. Herein we only give some indications of the possible applications of optical computing concepts on systolic algorithms, and parallel processing.

Summarizing, we can say that the main subject of this work is the investigation of new systolic algorithms for numerical computation, where 'new' may mean:

New groups of algorithms: solution of polynomial equations; evaluation and solution of characteristic equations; algorithms involving successive matrix powers.

New algorithms in traditional areas of research: updating of LU factors; inverse iteration; algorithms for iterative solution of linear equations using cyclic reduction, multicoloring techniques.

Improvements of existing algorithms: LU/LDU decomposition; matrix-vector multiplication; iterative solution of linear systems; power method.

The OCCAM programming language for parallel processing is extensively used for the soft-systolic simulation, and the partial verification, of the algorithms presented. By soft-systolic simulation we mean the simulation of systolic designs on a conventional uniprocessor, using the method of soft-systolic algorithm development. By verification we simply mean the production of expected results for qiven Thus, systolic architectures are considered as netinputs. works of processes, rather than processors, and the computation is data-driven. Further, no attempt is made to optimise the performance of the programs for a specific parallel processing computing structure. However, a methodology is developed that considerably simplifies the development and manipulation of systolic algorithms, using OCCAM. Finally, all programs (and algorithms) retain the possibility of direct implementation on VLSI processor arrays (transputer networks) with only minor modifications.

Further, an important recurring theme is the area-time tradeoffs, especially in the form of area-time expansion schemes for the systolic implementation of iterative algorithms, i.e: solution of polynomial equations; iterative

solution of linear systems; algorithms involving successive matrix powers. Moreover, the possible interconnection of systolic arrays, in the form of systolic pipelines and iterative structures, is addressed, in an attempt to develop systolic systems for the solution of complex problems, based on simple building blocks. Thus, a new level of pipelining is introduced, in addition to bit-level, arithmetic operation level and 'block of computations' level. It can be termed pipelining at an 'algorithmic level': complex alqorithms can be decomposed into simpler ones, which form a structure that can be expanded either in area, to produce systolic pipelines or networks, or in time, to produce iterative systolic arrays, with limited reconfigurability.

Finally, the impact of the advances in optics technology and optical computing is discussed, and the optical implementation of systolic algorithms is further investigated. Basic systolic algorithms for VLSI are modified in order to be implemented on to optical processors. The softsystolic simulation technique is used to simulate optical algorithms in OCCAM. Therefore, it can be argued that the simulation of algorithms and architectures for optical computing consists a new application field for the OCCAM programming language. The programs presented herein offer some initial indications in this direction.

Using the classification of systolic algorithms of section 3.5, it is possible to categorize the new algorithms

- 520 -

presented in this thesis, in the four groups: hard, soft, hybrid and optical systolic algorithms. Initially, it is clear that all algorithms can be classified as softsystolic, in the sense that they are conceived and simulated as networks of processes; thus it is relatively straightforward to map them on to an appropriate computing structure (transputer network). Notice that the optical-systolic algorithms, especially in sections 8.4 and 8.5 are the most difficult to map due to global interconnections; however a mapping is still possible, and the classification legal.

In the optical-systolic group, apart from the algorithms in chapter 8, we may add some of the systolic pipelines and networks described in section 6.5 and 6.6, as well as chapter 7, because of the optical data interconnections that may be used.

Hybrid-systolic algorithms allow for programmable components with significant amounts of local memory and control. In this category we could include the iterative structures for successive matrix-matrix or matrix-vector multiplications, in chapters 6 and 7. Further, the block (2x2) R+F decomposition array of section 5.2, the rectangular array for updating LU factors in section 5.3, and the programmable linear array for inverse iteration. Finally, all hard-systolic algorithms can be seen as hybrid ones.

In the class of hard-systolic algorithms, we can identify the polynomial equation solvers, in chapter 4, since they involve minimum programmability and memory; however, a general ring architecture would be hybrid-systolic. Further, in the same class we can include: the linear array for updating LU factors in section 5.3; the inverse iteration array in section 5.4; the matrix-vector multiplicaton arrays and the related simple pipelines in chapter 6. Notice that, all linear arrays that can be seen as degenerate 2-d arrays may well be classified as hybrid-systolic designs, in the sense that they can be transformed into pure 1-d structures, if adequate local memory is available.

In this thesis we have investigated a wide range of numerical algorithms and, to some extent, have achieved some unification by applying similar structures and methodology to this diversity. Thus, some generic systolic architectures can be identified; for each generic structure, we can outclass of corresponding algorithms. The algorithms line а belonging to a specific group can be implemented on the corresponding architecture, provided that the systolic system is appropriately programmed and/or reconfigured. These systolic systems can be envisaged either as special-purpose systolic computers, implementing hybrid-systolic algorithms, soft-systolic structures performable on a generalas purpose parallel processing computer. Therefore, the following structures, and related groups of algorithms can be identified: [°]

- Systolic Ring architecture, for polynomial computations,
 e.g. solution of polynomial equations, (see chapter 4).
- Systolic Networks, based on mvm mmm arrays, for algorithms involving successive mvips mmips computations,
 e.g. iterative solution of linear systems of equations,
 (see chapter 6); or matrix squaring algorithms, (see chapter 7).
- Optical Systolic Outer Product Processor, for optical systolic algorithms decomposable in simple outer product steps, e.g. Gauss elimination, (see chapter 8).

The Systolic Ring archiecture, for the solution of polynomial equations can be viewed as a three-part system. The main component is a pipeline structure capable of performing one or more polynomial evaluations at given points, which can provide the approximations to the roots sought. Then, another processor can check for the convergence of the method; finally a processor can close the ring by producing either the results or new approximations, if necessary. It may be possible for several methods to be implemented, depending on the computations performed in the pipeline and the configuration of the convergence and approximation pro-The Systolic Ring may process many polynomials in cessors. parallel, or may produce a number of roots of the same polynomial in parallel. Further, the pipeline can be used for other computations, such as polynomial multiplication, synthetic division, polynomial division, etc.

Systolic Networks have been defined as parallel, cooperating systolic designs, each design having the form of either a pipeline of systolic arrays, or an iterative systolic structure. The basic component of the system is a number of mvm (and/or mmm) systolic arrays for full (or banded) matrix computations. These arrays can be combined through a reconfigurable interconnection system, which allows the outputs of an array to be routed to other arrays and, possibly, back to the input of the same array. This feature is especially useful for cyclic reduction and multi-coloring methods, as well as in the computation of

matrix functions. A number of processors, attached to the interconnection system, can be used for further simple processing of the intermediate results, as they are routed through the network for the next iteration. This is. for example, the case of additional IPS operations required in some iterative methods for solving linear systems of equations, e.g GS method, or JOR method for 2-cyclic matrices. Finally, the Systolic Network should be accompanied with adequate memory for storing matrix operands and/or intermediate results. The storage of matrix operands is necessary the case of iterative structures based on mvm or mmm in arrays, while the storage of intermediate results is useful the two-stage recurrences for the computation of matrix in functions.

The Outer Product Processor can be used as the basic component for the construction of hybrid optical-electronic processors, implementing optical systolic algorithms based on banded mmm computations decomposable in a series of outer products. The processor is compared favourably with the optprocessor discussed in [13], since it requires less ical area and is problem size-independent. A number of Outer Pro-Processors can operate in parallel to perform multiple duct outer products using the same coefficient marix, as is the Gauss elimination algorithms. Further, a feedback case in mechanism may allow the use of the processor for algorithms based on successive mmm's.

9.2 SOME FURTHER SUGGESTIONS

In less than ten years since the first publication on systolic algorithms and architectures, research on systolic systems has undergone an impressive expansion, covering a large number of very divergent areas, as shown in sections 1.1 and 1.2. This thesis is just a small example of this multi-disciplinary research effort, as it tries to combine numerical algorithms, parallel processing languages and programming techniques, VLSI processor arrays and optical computing, using the systolic approach as the connecting theme.

However, as systolic computing and systems research reaches its first decade, there awaits the imperative task of actually implementing a significant proportion of the ideas set forth - mainly algorithms and architectures. This implementation side is the one that requires further development, for the systolic approach to be established as one of the major contenders in parallel processing.

In this direction, we can distinguish three types of implementation strategies, roughly corresponding to the proposed classification of systolic algorithms. Firstly, algorithmically specialized systems: they can perform a limited number of algorithms; they correspond to the groups of hard, and optical systolic arrays, with limited programmability and memory, and are of major interest for critical algorithms in real-time signal and image processing. Then, special-purpose systolic computers: they can perform a large number of algorithms in some specific areas of computing, e.g. computer vision, specific matrix computations, solution certain differential equations; they correspond to of hybrid-systolic algorithms and are of main interest as attached units in large-scale scientific computers and signal-image processing research. Finally, general-purpose parallel computers: they can perform algorithms from a large addition to the ones number of areas of computing, in already referred to, e.g. general numerical applications, data structures and data bases, systems programming, artificial intelligence; they correspond to soft-systolic algorithms and are of general interest for the development of parallel processing.

The problems that have to be solved in all these three implementation strategies can be summarized under the headings: hardware implementation of computing structures; overall system design and programming; algorithm design and development (see also section 1.2). However, the relative weighting of the problems for each strategy may be different.

Thus, for algorithmically specialized systems, it is important to minimize the system design costs and to achieve very high performances. Therefore, the main research effort is concentrated on the topics: automated design of systolic algorithms and architectures; use of specialized Computer Aided Engineering (CAE) tools; hardware and/or algorithmic fault-tolerance; efficient problem partitioning; use of Very High Speed Integrated Circuits (VHSIC), and/or hybrid optical-electronic processors; area and time optimization at all levels, [171], [217], [264], [302].

For special-purpose systolic computers, it is important to achieve a combination of relative flexibility and high performance. Therefore, the main research effort is concentrated on the overall system design and architecture, so that it will be suitable for some 'generic' algorithms, representing the areas of specialization. Problems such as processor and interconnection complexity, host-system communication and environment for program development are very significant. Notice that the algorithms now have to match the architecture, in a way similar to that of a low-level sequential machine environment. A good example for this implementation strategy is the recently commercially available, Warp machine.

Further, for general-purpose parallel computers, the problems start from the definition itself: for the 'generic' algorithms are too general (or too many) with a large number of tradeoffs to be balanced, and thus a large number of alternatives to be followed. Therefore, the main research effort is to achieve near-optimal balance between computation and communication for algorithms that vary considerably in their pattern of parallelism. Problems arise at the level of processor and interconnection complexity: 'clusters' of

- 525 -

processors and multiple interconnections are being considered. Further, at the algorithm design level, the algorithm itself may be detached from the computer structure geometry, and therefore the problem of the optimal mapping of the algorithm on to the structure is very important. For the moment, the soft-systolic approach produces useful results in the areas of systems programming, as well as logic programming; further, the concept of Instruction Systolic Machines is investigated, that distribute systolically both instructions and data, [156], [256], [286].

Finally, we think that the combination of the softsystolic concept with other, competing and/or closely related approaches in parallel processing may be fruitful; such examples are the Hypercube computer, the Connection machine [264] and the Reconfigurable Array Processor (RPA) [150]. Also, recent developments in parallel reduction systems [208], [281]; multi-valued logic [213]; neural systems (using optical computing techniques) [279]; VLSI and optical cellular computers [55], [70-71]; and molecular computers (in relation to cellular automata) [66], constitute an interesting challenge for further cross-fertilization of the systolic concept with other radical approaches in parallel computing.

What is imperative, in all the above strategies, is the formulation of a simple, but powerful, notation (language), for the development of parallel processing algorithms in general and systolic algorithms specifically. In this direction, OCCAM proves to be a very useful tool, although not fully developed yet. Furthermore, the ability to build, using transputers, many types of experimental parallel machines, is very important for the development of systolic (and parallel) algorithms and architectures.

This thesis is completed with a comprehensive list of references, followed by an Appendix on OCCAM, the Loughbourough implmentation of the language, and a selection of programs, simulating some of the systolic algorithms proposed herein.

REFERENCES

- Abdel Kader, A.A., "OCSAMO: A Systolic Array for matrix operations", in Proc. CONPAR 86, Springer-Verlag, 1986, pp. 319-328.
- Abushagur, M.A.G., Caulfield, H.J., "Speed and convergence of bimodal optical computers", <u>Optical Engineer-</u> ing, 1987(26), pp. 22-27.
- 3. Adams, L., "Iterative algorithms for large sparse linear system on parallel computers", Ph.D. thesis, Virginia University, 1982.
- Alves Marques, I., Cunha, A., "Clocking of VLSI circuits", in "VLSI Architecture", Randell, B., et al. (eds.), Prentice-Hall, 1983, pp. 165-178.
- 5. Ahmed, H.M., Delosme, J., Morf, M., "Highly Concurrent Computing Structures for Matrix Arithmetic and Signal Processing", IEEE Computer, 1982(15), pp. 65-82.
- Ang, P.H., Morf, M., "Concurrent Array Processor for Fast Eigenvalue Computations", Proc. IEEE ICASSP 1984, pp. 34A.2.1-34A.2.4.
- Annaratone, M., et al., "Extending the CMU Warp Machine with a Boundary Processor", Proc. SPIE, RTSP VIII, 1985(564), pp. 56-65.
- Annaratone, M., et al., "Warp Architecture and Implementation", Proc. 13th Annual Intern. Symposium on Computer Architecture, Tokyo, 1986, pp. 346-356.
- 9. Arnould, E. et al., "A Systolic Array Computer", Proc. IEEE ICAASP, 1985, pp. 6.11.1-6.11.4.
- Arrathoon, R., "Digital Optical Computing: possibilities and pitfalls", Proc. SPIE, RTSP VIII, 1985(564), pp. 108-118.
- 11. Arrathoon, R. et al., "Digital Convolution and Correlation with Electrooptic Bragg Processors", Proc. SPIE,

RSIP VII, 1984(495), pp. 150-158.

- Athale, R.A., Collins, W.C., Stiluell, P.D., "High accuracy matrix multiplication with outer product optical processor", Applied Optics, 1983(22), pp. 368-370.
- Athale, R.A., Lee, J.N., "Optical Processing using Outer-Product Concepts", <u>Proc. IEEE</u>, 1984(72), pp. 931-941.
- 14. Athale, R.A., Lee, J.N., Hoang, H.Q., "High accuracy matrix-multiplication with a magnetooptic spatial light modulator", Proc. SPIE, RTSP VI, 1983(441), pp. 187-193.
- 15. Atkinson, K.E., "An Introduction to Numerical Analysis", John Wiley, 1978.
- 16. Avila, J.H., Kuekes, P.J., "A one gigaflop VLSI systolic processor", Proc. SPIE, RTSP VI, 1983(441), pp. 159-165.
- 17. Bartels, R.H., "A Stabilization of the Simplex Method", Numer. Math., 1971(16), pp. 414-434.
- Barth, B., Martin, R.S., Wilkinson, J.H., "Calculation of the Eigenvalues of a Symmetric Tridiagonal Matrix by the Method of Bisection", <u>Numer</u>. <u>Math.</u>, 1967(9), pp. 386-393.
- 19. Bekakos, M.P, "A Study of Algorithms for Parallel Computers and VLSI Systolic Processor Arrays", Ph.D Thesis, Dept. of Computer Studies, LUT, 1986.
- Bekakos, M.P., Evans, D.J., "The Exposure and Exploitation of Parallelism in Fifth Generation Computer Systems", Proc. Parallel Computing '85, North-Holland, 1986, pp. 425-442.
- 21. Bekakos, M.P., Evans, D.J., "A Rotating and 'Folding' Algorithm using a two-dimensional 'Systolic' Communication Geometry", <u>Parallel</u> <u>Computing</u>, 1988, in press.
- 22. Bellman, R., "Introduction to Matrix Analysis", McGraw-Hill, 1960.
- 23. Bennet, J.M., "Triangular Factors of Modified Matrices", <u>Numer</u>. <u>Math</u>., 1965(7), pp. 217-221.
- 24. Bertossi, A.A., Bonuccelli, M.A., "A VLSI Implementations of the Simplex Algorithm", <u>IEEE</u>, <u>Trans. on Com-</u> puters, 1987(C-36), pp. 241-247.
- 25. Berzins, M., Buckley, T.F., Dew, P.M., "Systolic Matrix

Iterative Algorithms", Proc. Parallel Computing '83, North-Holland, 1984, pp. 483-488.

- 26. Blackmer, J., Kuekes, P., Frank, G., "A 200 MOPS systolic processor", Proc. SPIE, RTSP IV, 1981(298), pp. 10-18.
- 27. Blum, E.K., "Numerical Analysis and Computation: Theory and Practice", Addison-Wesley, 1972.
- 28. Bocker, R.P., "Algebraic operations performable with electrooptical engagement array processors", Proc. SPIE, Optical Information Processing, 1983(388), pp. 212-220.
- 29. Bocker, R.P., Bromley, K., Clayton, S.R., "A digital Optical Architecture for performing matrix algebra", Proc. SPIE, RTSP VI, 1983(441), pp. 194-200.
- Bocker, R.P., Caulfield, H.J., Bromley, K., "Rapid Unbiased Bipolar Incoherent Calculator Cube (RUBIC)", Proc. SPIE, Optical Information Processing, 1983(388), pp. 205-211.
- 31. Bocker, R.P., et al., "Optical Fixed-Point Arithmetic", Proc. SPIE, RTSP VIII, 1985(564), pp. 150-156.
- 32. Bodewig, E., "Matrix Calculus", North-Holland, 1959.
- 33. Bojanczyk, A., Brent, R.P., "Tridiagonalisation of a symmetric matrix on a square array of mesh-connected processors", Technical Report CMA-R45-83, Centre for Mathematical Analysis, Australian National University, 1983.
- 34. Bojanczyk, A., Brent, R.P., Kung, H.T., "Numerically stable solution of Dense System of Linear Equations using Mesh-Connected Processors", <u>SIAM J. on Scientific</u> and <u>Statistical Computing</u>, 1984(5), pp. 95-104.
- 35. Brenner, K.H., Lohmann, A.W., "The Digital Optical Computing Program at Erlangen", Proc. CONPAR '86, Springer-Verlag, 1986, pp. 69-75.
- 36. Brent, R.P., Kung, H.T., "Systolic Arrays for Linear Time GCD Computation", <u>IEEE</u>, <u>Trans. on Computers</u>, 1984(C-33), pp. 731-736.
- 37. Brent, R.P., Kung, H.T., Luk, F.T., "Some Linear-time Algorithms for Systolic Arrays", Proc. 9th World Computer Congress, Paris, 1983, pp. 19-23.
- 38. Brent, R.P., Luk, F.T., "A Systolic Architecture for almost Linear-time solution of the Symmetric Eigenvalue

Problem", Tech. Report TR-CS-82-10, Depart. of Comp. Sci., Australian National Univ., Canberra, 1982.

- 39. Brent, R.P., Luk, F.T., "The solution of Singular Value Problems using Systolic Arrays", Proc. SPIE, RTSP VII, 1984(495), pp. 7-12.
- 40. Brent, R.P., Luk, F.T., Van Loan, C., "Computation of the Singular Value Decomposition using Mesh-Connected Processors", Proc. SPIE, RTSP VI, 1983(441), pp. 66-71.
- 41. Brochard, L., "Domain Decomposition and Relaxation Methods", in "Parallel Algorithms and Architectures", Cosnard, M. et. al. (eds.), North-Holland, 1986, pp. 61-72.
- 42. Brodetsky, S., Smeal, G., "On Graeffe's Method for Complex Roots of Algebraic Equations", <u>Proc. Cambridge</u> <u>Philosophical Society</u>, 1924(22), pp. 83-87.
- 43. Bromley, K., et al., "Systolic Array Processor Developments", in "VLSI Systems and Computations", Kung, H.T., et al. (eds.), Computer Science Press, 1981, pp. 273-284.
- Broomhead, D.S., et al., "A practical comparison of the Systolic and Wavefront Array Processing Architectures", Proc. IEEE ICASSP, 1985, pp. 8.7.1-8.7.4.
- 45. Brudaru, O., "Systolic Algorithms to solve Linear Systems by Iteration Methods", Computer Centre, Polytechnical Institute, Iasi, Romania, 1985.
- 46. Capello, P.R., Steiglitz, K., "Digital Signal Processing Applications of Systolic Algorithms", in "VLSI Systems and Computations", Kung, H.T., et al. (eds.), 1981, pp. 245-254.
- 47. Capello, P.R., Steiglitz, K., "Selecting Systolic Designs using Linear Transformations of Space-Time", Proc. SPIE, RTSP VII, 1984(495), pp. 75-85.
- 48. Carnahan, B., Luther, H.A., Wilkes, J.O., "Applied Numerical Methods", John Wiley, 1969.
- 49. Cartwright, S., Gustafson, S.C., "Convolver-based optical systolic processing architectures", <u>Optical</u> Engineering, 1985(24), pp. 59-64.
- 50. Casasent, D., "Acoustooptic Linear Algebra Processors: Architectures, Algorithms and Applications", <u>Proc</u>. IEEE, 1984(72), pp. 831-849.
- 51. Casasent, D., Ghosh, A., "Optical Linear Algebra",

- 531 -

Proc. SPIE, Optical Information Processing, 1983(388), pp. 182-189.

- 52. Casasent, D., Ghosh, A., Neuman, C.P., "Direct and Indirect Optical Solutions to Linear Algebraic Equations: Error Source Modeling", Proc. SPIE, RTSP VI, 1983(441), pp. 201-208.
- 53. Casasent, D., Ghosh, A., Neuman, C.P., "Iterative Solutions to Nonlinear Matrix Equations using a Fixed Number of steps", Proc. SPIE, RTSP VII, 1984(495), pp. 102-108.
- 54. Casasent, D., Taylor, B.K., "Optical Finite Element Processor", Proc. SPIE, RTSP VIII, 1985(564), pp. 139-149.
- 55. Caulfield, H.J., "Optical Cellular Array Processors", Proc. SPIE, RTSP VIII, 1985(564), pp. 45-48.
- 56. Caulfield, H.J., Gruninger, J.H., Cheng, W.K., "Using Optical Processors for Linear Algebra", Proc. SPIE, Optical Information Processing, 1983(388), pp. 190-196.
- 57. Caulfield, H.J., Putman R.S., "Fault tolerance and self-healing in Optical Systolic Array Processors", Optical Engineering, 1985(24), pp. 65-67.
- 58. Caulfield, H.J., et al., "Optical Implementation of Systolic Array Processing", <u>Optics Communications</u>, 1981(40), pp. 86-90.
- 59. Chandy, K.M., Misra, J., "Systolic algorithms as programs", <u>Distributed Computing</u>, 1986(1), pp. 177-183.
- 60. Chapman, R., Durrani, T.S., Willey, T., "Design Strategies for Implementing Systolic and Wavefront Arrays using OCCAM", Proc. IEEE ICASSP, 1985, pp. 8.6.1-8.6.4.
- 61. Chen, M.C., Mead, C.A., "Concurrent Algorithms as Space-Time Recursion Equations", in "VLSI and Modern Signal Processing", Kailath, T., et al. (eds.), Prentice-Hall, 1985, pp. 224-240.
- 62. Chen, M.J., Yao, K., "On Realizations of Least-Squares Estimation and Kalman Filtering by Systolic Arrays", Proc. Int. Workshop on Systolic Arrays, Univ. of Oxford, 1986, pp. 161-170.
- 63. Codenotti, B., "The Matrix Equation MX+XN=B in the VLSI Model", <u>International</u> J. of <u>Computer</u> <u>Mathematics</u>, 1986(19), pp. 93-98.
- 64. Cohen, A.M., "Numerical Analysis", McGraw-Hill, 1973.

- 65. Collar, A.R., "Some notes on Jahn's method for the improvement of approximate latent roots and vectors of a square matrix", <u>Quart. J. of Mechanical</u> and <u>Applied</u> Maths, 1948(1), pp. 145-148.
- 66. Conrad, M., "On design principles for a Molecular Computer", <u>Communications</u> of the <u>ACM</u>, 1985(28), pp. 464-480.
- 67. Cooley, E.S., Israel, S.C., "Sideways Summer", Proc. SPIE, RTSP VIII, 1985(564), pp. 98-100.
- 68. Cosnard, M., Robert, Y., Trystram, D., "Parallel Solution of Dense Linear Systems using Diagonalization Methods", <u>Intern. J. Computer Math</u>, 1988, in press.
- 69. Csanky, L., "Fast Parallel Matrix Inversion Algorithms", <u>SIAM J. of Computing</u>, 1976(5), pp. 6i8-623.
- 70. Culik, II,K., Gruska, J., Salomaa, A., "Systolic Trellis Automata, Part I and II", <u>Intern. J. Computer</u> <u>Math.</u>, 1984(15), pp. 195-212; 1984(16), pp. 3-22.
- 71. Culik, II,K., Yu, S., "Fault-tolerant schemes for some systolic systems", <u>Intern. J. Computer Math.</u>, 1987(22), pp. 13-42.
- 72. Dahlquist, G., Bjork, A., "Numerical Methods", Prentice-Hall, 1974.
- 73. Danielsson, P.E., "Serial/Parallel Convolvers", <u>IEEE</u>, <u>Trans. on Computers</u>, 1984(C-33), pp. 652-667.
- 74. Davis, G.J., "Numerical Solution of a Quadratic Matrix Equation", <u>SIAM</u>, <u>J. Sci. Stat. Comput.</u>, 1981(2), pp. 164-175.
- 75. Davis, J.A., Jones, K.D., Lilly, R.A., "Improved system for binary multiplication by optical convolution", <u>Opt-</u> ical Engineering, 1986(25), pp. 572-574.
- 76. Delosme, J.M., Ipsen, I.C.F., "Systolic Array Synthesis: Computability and Time Cones", in "Parallel Algorithms and Architectures", Cosnard, M., et al. (eds.), North-Holland, 1986, pp. 295-316.
- 77. Delosme, J.M., Ipsen, I.C.F., "Efficient Systolic Arrays for the solution of Toeplitz Systems: an illustration of a methodology for the construction of Systolic Architectures in VLSI", Proc. Int. Workshop on Systolic Arrays, Univ. of Oxford, 1986, pp. 37-46.
- 78. Denyer, P.B., "An Introduction to bit-serial architectures for VLSI signal processing", in "VLSI

Architecture", Randell, B., et al. (eds.), Prentice-Hall, 1983, pp. 225-241.

- 79. Denyer, P.B., Smith, S.G., "Bit Serial Architectures for Parallel Arrays", Proc. SPIE, Highly Parallel Signal Processing Architectures, 1986(614), pp. 66-73.
- Dew, P.M., "VLSI Architectures for problems in Numerical Computation", in "Supercomputer and Parallel Computation", Paddon, D.J. (ed.), Oxford Univ. Press, 1984, pp. 1-24.
- 81. Dew, P.M., Manning, L.J., "Comparison of Systolic and SIMD Architectures for Computer Vision Computations", Proc. Int. Workshop on Systolic Arrays, Univ. of Oxford, 1986, pp. 273-282.
- Dew, P.M., Manning, L.J., McEvoy, K., "A tutorial on Systolic Array Architectures for High Performance Processors", Univ. of Leeds, Technical Report 205, 1986.
- 83. Distante, F., Sami, M.G., "A Protocol for Asynchronous Wavefront Computation Arrays", Proc. Int. Workshop on Systolic Arrays, Univ. of Oxford, 1986, pp. 249-258.
- 84. Drake, B.L., Bocker, R.P., "A Highly Parallel Algorithm for Computing the Singular Value Decomposition using Optical Processing Techniques", Proc. SPIE, RTSP VII, 1984(495), pp. 166-174.
- 85. Drake, B.L., et al., "Photonic computing using the modified signed-digit number representation", Optical Engineering, 1986(25), pp. 38-43.
- 86. Dunway, D.K., "Calculation of the zeros of a real polynomial through factorization using Euclid's algorithm", <u>SIAM J. Numer. Anal.</u>, 1974(11), pp. 1087-1104.
- 87. Eichman, G., "Systolic Arrays for Eigenvalue Computation", Proc. SPIE, RTSP VIII, 1985(564), pp. 39-44.
- 88. El-Amawy, A., Porter, W.A., Aravena, J.L., "Array architectures for iterative matrix calculations", <u>Proc</u>. IEEE, 1987(134-E), pp. 149-154.
- 89. Ersoy, O., "Semisystolic Array Implementation of Circular, Skew Circular and Linear Convolutions", <u>IEEE</u>, Trans. on Computers, 1985(C-34), pp. 190-196.
- 90. Evans, D.J., "Computation of eigenvalues and eigenvectors of a symmetric quindiagonal matrix", J. of Computational and Applied Mathematics, 1977(3), pp. 131-141.
- 91. Evans, D.J., "Parallel Processing Systems", Cambridge

- 534 -

Univ. Press, 1982.

- 92. Evans, D.J., "Systolic Arrays" and "Design of Systolic Arrays", Lecture Notes, Dept. of Computer Studies, LUT, 1986.
- 93. Evans, D.J., Bekakos, M.P., "On the Implementation of Acousto-optic Cells for a 'Rotating' and 'Folding' Algorithm Systolization", <u>Intern. J. Computer Math.</u>, 1986(20), pp. 123-129.
- 94. Evans, D.J., Bekakos, M.P., Margaritis, K.G., "Optical 'Dequeues' for a 'R and F' Systolic LU-factorization of Tridiagonal Systems", Proc. 4th Int. Symposium on Optical and Optoelectronic Applied Science and Engineering, Hague, 1987.
- 95. Evans, D.J., Margaritis, K.G., "Optical Implementation of Banded Matrix Algorithms using outer products", Proc. 4th Int. Symposium on Optical and Optoelectronic Applied Science and Engineering, Hague, 1987.
- 96. Evans, D.J., Margaritis, K.G., "Systolic Designs for the Root-Squaring Method", <u>Intern. J. Computer Math.</u>, 1987(22), pp. 43-62.
- 97. Evans, D.J., Margaritis, K.G., "Improved Systolic designs for Iterative solution of Linear systems", in Proc. NUMETA '87, Swansea, 1987, pp. S18.1-S18.17.
- 98. Evans, D.J., Margaritis, K.G., Bekakos, M.P., "A systolic and holographic pyramidical soft-systolic designs for successive matrix powers", <u>Parallel</u> <u>Computing</u>, 1988, in press.
- 99. Evans, D.J., Margaritis, K.G., Bekakos, M.P., "On Acousto-optic cell planes to map a R+F algorithm using a 2-D Systolic geometry", to appear in <u>Optics</u> <u>Communi-</u> cations, 1988.
- 100. Evans, D.J., Megson, G.M., "Romberg integration using Systolic Arrays", <u>Parallel</u> <u>Computing</u>, 1986(3), pp. 289-304.
- 101. Evans, D.J., Shanehchi, J., Rick, C.C., "A Modified Bisection Algorithm for the Determination of the Eigenvalues of Symmetric Tridiagonal Matrix", <u>Numer</u>. <u>Math</u>., 1982(38), pp. 417-419.
- 102. Fiat, A., Shamir, A., Shapiro, E., "Polymorphic Arrays: an architecture for a programmable systolic machine", Technical Report, CS84-20, Weisman Institute of Science, Rehovot, Israel, 1984.

- 103. Fisher, A.L., Kung, H.T., "Synchronizing Large VLSI Processor Arrays", <u>IEEE</u>, <u>Trans. on Computers</u>, 1985(C-34), pp. 734-740.
- 104. Fisher, A.L., Kung, H.T., "Special purpose VLSI Architectures: general discussions and a case study", in "VLSI and Modern Signal Processing", Kailath, T., et al. (eds.), Prentice-Hall, 1985, pp. 153-169.
- 105. Fletcher, R., Mathews, S.P.J., "Stable modification on explicit LU factors for simplex updates", Technical Report, NA/64, Dept. of Mathematical Sciences, Univ. of Dundee, 1983.
- 106. Forsythe, G.E., Malcolm, M.A., Moler, C.B., "Computer methods for Mathematical Computations", Prentice-Hall, 1977.
- 107. Fortes, J.A.B., "Algorithm Reconfiguration Techniques for Gracefully Degradable Processor Arrays", Proc. Int. Workshop on Systolic Arrays, Univ. of Oxford, 1986, pp. 259-268.
- 108. Fortes, J.A.B., Fu, K.S., Wah, B.W., "Systematic Approaches to the Design of Algorithmically Specified Systolic Arrays", Proc. IEEE ICASSP, 1985, pp. 8.9.1-8.9.4.
- 109. Foster, M.J., Kung, H.T., "The design of special purpose VLSI chips", IEEE Computer, 1980(13), pp. 26-40.
- 110. Frison, P., Quinton, P., "An integrated systolic machine for speech recognition" in "VLSI: Algorithms and Architectures", Bertolazzi, P., et al. (eds.), North-Holland, 1985, pp. 175-186.
- 111. Froberg, C.E., "Introduction to Numerical Analysis", Addison-Wesley, 1965.
- 112. Gachet, P., Joinnault, B., Quinton, P., "Synthesizing Systolic Arrays using DIASTOL", Proc. Int. Workshop on Systolic Arrays, Univ. of Oxford, 1986, pp. 25-36.
- 113. Gentleman, W.M., Kung, H.T., "Matrix triangularization by systolic arrays", Proc. SPIE, RTSP IV, 1981(298), pp. 19-26.
- 114. Gerald, C.F., "Applied Numerical Analysis", Addison-Wesley, 1970.
- 115. Gill, P.E., et al., "Methods for modifying Matrix Factorizations", <u>Math. Comput.</u>, 1974(28), pp. 505-535.
- 116. Goodman, J.W., et al., "Optical Interconnections for

VLSI Systems", Proc. IEEE, 1984(72), pp. 850-866.

- 117. Golub, G.H., Van Loan, C.F., "Matrix Computations", North Oxford, 1983.
- 118. Guerra, C., "A unifying framework for systolic designs", Proc. AWOC '86, Springer-Verlag, 1986, pp.140-149.
- 119. Guerra, C., Kanade, T., "A systolic algorithm for stereo matching", in "VLSI: Algorithms and Architectures", Bertolazzi, P., et al. (eds.), North-Holland, 1985, pp. 103-112.
- 120. Guibas, L.J., Kung, H.T., Thompson, C.D., "Direct VLSI implementation of combinatorial algorithms", Proc. Cal-Tech Conf. on VLSI, 1979, pp. 509-525.
- 121. Guibas, L.J., Liang, F.M., "Systolic stacks, queues and counters", Proc. of Conf. on Advanced Research in VLSI, M.I.T., 1982, pp. 155-164.
- 122. Hasegawa, M., Shigei, Y., "AT² =O(Nlog⁴ N), T=O(logN) FFT in a Light connected 3-Dimensional VLSI", Proc. 13th Ann. Int. Symp. on Computer Architecture, Tokyo, 1986, pp. 252-260.
- 123. Hauck, C.E., Bamji, C.S., Allen J., "The Systematic Exploration of Pipelined Array Multiplier Performance", Proc. IEEE ICASSP, 1985, pp. 38.3.1-38.3.4.
- 124. Haynes, L.S., et al., "A survey of Highly Parallel Computing", <u>IEEE Computer</u>, 1982(15), pp. 9-24.
- 125. Heller, D., "Partitioning Big Matrices for Small Systolic Arrays", in "VLSI and Modern Signal Processing", Kailath, T., et al. (eds.), Prentice-Hall, 1985, pp. 185-199.
- 126. Heller, D.E., Ipsen, I.C.F., "Systolic Networks for Orthogonal Equivalence Transformations and their Applications", Proc. of Conf. on Advanced Research in VLSI, M.I.T., 1982, pp. 113-122.
- 127. Helton, B.W., "Logarithms of Matrices", Proc. American Mathematical Society, 1968(19), pp. 733-736.
- 128. Higham, N.J., "Newton's Method for the Matrix Square Root", Math. Comput., 1986(46), pp. 537-549.
- 129. Hoskins, W.D., Walton, D.J., "A faster method of computing the square root of a matrix", <u>IEEE</u>, <u>Trans. on</u> Automatic Control, 1978(AC-23), pp. 494-495.
- 130. Hossfeld, F., "Strategies for Parallelism in Algorithms", Lecture notes in IBM summer school in Parallel Computing, 1986.
- 131. Hotelling, H., "Some new methods in matrix calculation", <u>Ann. Math. Stat.</u>, 1943(4), pp. 1-33.
- 132. Hu, Y.H., "VLSI Architecture for solving Covariance Eigen System", Proc. IEEE ICASSP, 1985, pp. 8.4.1-8.4.4.
- 133. Hu, Y.H., Kung, S.Y., "Computation of minimum eigenvalue of Toeplitz matrix by Levinson Algorithm", Proc. SPIE, RTSP IV, 1981(298), pp. 40-45.
- 134. Huang, K.H., Abraham, J.A., "Algorithm based fault tolerance for matrix operations", <u>IEEE</u>, <u>Trans.</u> on <u>Com-</u> puters, 1984(C-33), pp. 518-528.
- 135. Hsu, F.H., et al., "LINC: the link and interconnection chip", Technical Report, Dept. of Computer Science, CMU, 1984.
- 136. Hsu, I.N., et al., "The VLSI Implementation of a Reed-Solomon Encoder using Berlekamp's Bit-Serial Multiplier Algorithm", <u>IEEE</u>, <u>Trans. on Computers</u>, 1984(C-33), pp. 906-911.
- 137. Hwang, K., Briggs, F.A., "Computer Architecture and Parallel Processing", McGraw-Hill, 1984.
- 138. Hwang, K., Cheng, Y.H., "Partitioned Matrix Algorithms for VLSI Arithmetic Systems", <u>IEEE</u>, <u>Trans</u>. <u>on</u> <u>Comput-</u> ers, 1982(C-31), pp. 1215-1224.
- 139. Ibarra, O.H., Kim, S.M., Palis, M.A., "Designing Systolic Algorithms using Sequential Machines", <u>IEEE</u>, <u>Trans. on Computers</u>, 1986(C-35), pp. 531-542.
- 140. Inmos Ltd., "OCCAM: user's manual", Prentice-Hall, 1985.
- 141. Inmos Ltd., "Transputer: reference manual", 1985, "The Transputer Family: product information", 1986.
- 142. Ipsen, I.C.F., "Stable Matrix Computations in VLSI", Ph.D. thesis, Pennsylvania State University, 1983.
- 143. Ipsen, I.C.F., "Singular Value Decomposition with Systolic Arrays", Proc. SPIE, RTSP VII, 1984(495), pp. 13-21.
- 144. Jagadish, H.V., et al., "A study of Pipelining in Computing Arrays", <u>IEEE</u>, <u>Trans.</u> on <u>Computers</u>, 1986(C-35),

pp. 431-440.

- 145. Jahn, H.A., "Improvement of an approximate set of latent roots and modal columns of a matrix by methods akin to those of classical perturbation theory", Quart. J. of Mech. and Appl. Math., 1948(1), pp. 131-144.
- 146. Jenkins, B.K., Giles, C.L., "Parallel Processing Paradigms and Optical Computing", Proc. SPIE, Optical Computing, 1986(625), pp. 22-29.
- 147. Jenkins, M., Traub, J., "A three-stage algorithm for real polynomials using quadratic iteration", <u>SIAM</u> J. <u>Numer. Anal.</u>, 1970(7), pp. 545-566.
- 148. Jennings, W., "First Course in Numerical Methods", McMillan, 1962.
- 149. Jesshope, C.R., "Support for the rapid processing of large data structures in OCCAM", Dept. of Electronics and Comp. Sci., Univ. of Southampton, 1986.
- 150. Jesshope, C.R., "The RPA: an Intelligent Transputer Memory System in an OCCAM Programming Model", Proc. Int. Workshop on Systolic Arrays, Univ. of Oxford, 1986, pp. 283-293.
- 151. Johnsson, L., Cohen, D., "A Mathematical Approach to Modelling the Flow of Data and Control in Computational Networks", "VLSI Systems and Computations", Kung, H.T., et al. (eds.), Computer Science Press, 1981, pp. 213-225.
- 152. Jones, G., "Programming in OCCAM", Oxford Univ. Computing Laboratory, Programming Research Group, 1985.
- 153. Jou, J.Y., Abraham, J.A., "Fault-tolerant Matrix Arithmetic and Signal Processing on Highly Concurrent Computing Structures", Proc. IEEE, 1986(74), pp. 732-741.
- 154. Koren, I., Pradhan, D.K., "Yield and Performance Enhancement through redundancy in VLSI and WSI Multiprocessor Systems", <u>Proc. IEEE</u>, 1986(74), pp. 699-711.
- 155. Kunde, M., "A General Approach to sorting on 3dimensionally Mesh-connected Array", Proc. CONPAR '86, Springer-Verlag, 1986, pp. 329-337.
- 156. Kunde, M., et al., "The Instruction Systolic Array and its relation to other models of Parallel Computers", Proc. Parallel Computing '85, North-Holland, 1986, pp. 491-498.

- 157. Krishnakumar, A.S., Morf, M., "A tree architecture for the symmetric eigenproblem", Proc. SPIE, RTSP VI, 1983(441), pp. 77-83.
- 158. Kulkarni, A.V., Yen, D.W.L., "Systolic Processing and an implementation for Signal and Image Processing", IEEE, Trans. on Computers, 1982(C-31), pp. 1000-1009.
- 159. Kung, H.T., "The structure of parallel algorithms", Advances in Computers, 1980(19), pp. 65-112.
- 160. Kung, H.T., "Why Systolic Architectures?", <u>IEEE Com-</u> puter, 1982(15), pp. 37-46.
- 161. Kung, H.T., "Two-level Pipelined Systolic Arrays for Matrix Multiplication, Polynomial Evaluation and Discrete Fourier Transform", Proc. Int. Workshop on Dynamical Behaviour of Automata: theory and applications, Academic Press, 1983.
- 162. Kung, H.T., "Systolic Algorithms for the CMU Warp Processors", Dept. of Computer Science, CMU, 1984.
- 163. Kung, H.T., "A listing of Systolic Papers", Dept. of Computer Science, CMU, 1985.
- 164. Kung, H.T., Lam, S.M., "Wafer-scale integration and two-level pipelined implementations of Systolic Arrays", J. of Parallel and Distributed Computing, 1984(1), pp. 32-63.
- 165. Kung, H.T., Lin, W.T., "An Algebra for VLSI Algorithm design", Proc. of Conf. on Elliptic Problem Solvers, 1983, pp. 141-160.
- 166. Kung, H.T., Picard, R.L., "One-dimensional Systolic Arrays for Multidimensional Convolution and Resampling", in "VLSI for Pattern Recognition and Image Processing", Fu, K.S. (ed.), Springer-Verlag, 1984, pp. 9-24.
- 167. Kung, H.T., Ruane, L.M., Yen, D.W.L., "A two-level pipelined Systolic Array for Convolutions", in "VLSI Systems and Computations", Kung, H.T., et al. (eds.), Computer Science Press, 1981, pp. 255-264.
- 168. Kung, H.T., Yu, S.Q., "Integrating high-performance special purpose devices into a system", in "VLSI Architecture", Randell, B., et al. (eds.), Prentice-Hall, 1983, pp. 205-211.
- 169. Kung, S.Y., "On Supercomputing with Systolic/Wavefront Array Processors", <u>Proc. IEEE</u>, 1984(72), pp. 867-884.

- 170. Kung, S.Y., "On Programming Languages for VLSI Array Processors", Proc. SPIE, Highly Parallel Signal Processing Architectures, 1986(614), pp. 118-133.
- 171. Kung, S.Y., "VLSI Array Processors", Proc. Int. Workshop on Systolic Arrays, Univ. of Oxford, 1986, pp. 7-24.
- 172. Kung, S.Y., et al., "Hierarchical Flowgraph Integration for VLSI Array Processors", Proc. IEEE ICASSP, 1985, pp. 8.5.1-8.5.4.
- 173. Kung, S.Y., et al., "Wavefront Array Processor: Language, Architecture and Applications", <u>IEEE</u>, <u>Trans</u>. on Computers, 1982(C-31), pp. 1054-1066.
- 174. Kung, S.Y., Hu, Y.H., "A highly concurrent algorithm and pipelined architecture for solving Toeplitz systems", IEEE, Trans. on Acoustics, Speech and Signal Processing, 1983(ASSP-31), pp. 66-75.
- 175. Kung, S.Y., Johl, J.T., "VLSI Wavefront Arrays for Image Processing", in "VLSI for Pattern Recognition and Image Processing", Fu, K.S. (ed.), Springer-Verlag, 1984, pp. 133-155.
- 176. Kung, S.Y., Lo, S.C., Annevelink, J., "Temporal localization on systolization of Signal Flow Graph (SFG) Computing Networks", Proc. SPIE, RTSP VII, 1984(495), pp. 58-66.
- 177. Lam, M.S., Mostow, J., "A transformational model of VLSI Systolic Design", <u>IEEE</u>, <u>Computer</u>, 1985(18), pp. 42-52.
- 178. Lang, H.W., et al., "Systolic Sorting on a Meshconnected Network", <u>IEEE</u>, <u>Trans.</u> on <u>Computers</u>, 1985(C-34), pp. 652-658.
- 179. Lang, H.W., Schimmler, M., Schroder, H., "Pattern matching in binary trees on a mesh-connected processor array", in "VLSI: Algorithms and Architectures", Bertolazzi, P., et al. (eds.), North-Holland, 1985, pp. 113-124.
- 180. Leighton, F.T., Leiserson, C.E., "Wafer-scale integration of systolic arrays", <u>IEEE</u>, <u>Trans. on Computers</u>, 1985(C-34), pp. 448-461.
- 181. Leiserson, C.E., "Area efficient VLSI computation", Ph.D. thesis, Dept. of Computer Science, CMU, 1981.
- 182. Leiserson, C.E., Saxe, J.B., "Optimising Synchronous Systems", J. of VLSI and Computer Systems, 1983(1), pp.

41-68.

- 183. Levin, M., "Parallel Algorithms for SIMD and MIMD computers", Ph.D. thesis, Dept. of Computer Studies, LUT, 1987.
- 184. Li, G.J., Wah, B.W., "The design of optimal systolic arrays", <u>IEEE</u>, <u>Trans.</u> on <u>Computers</u>, 1985(C-34), pp. 66-77.
- 185. Lin, Y.C., Lin, F.C., "A family of systolic arrays for relational database operations", Proc. Int. Workshop on Systolic Arrays, Univ. of Oxford, 1986, pp. 191-200
- 186. Lin, W.T., Chin, C.Y., "A reconfigurable processor array using LINC chip", Proc. Int. Workshop on Systolic Arrays, Univ. of Oxford, 1986, pp. 313-320.
- 187. Lipton, R., Lopresti, D., "Comparing Long Strings on a Short Systolic Array", Proc. Int. Workshop on Systolic Arrays, Univ. of Oxford, 1986, pp. 181-190.
- 188. Luk, F.T., "Algorithm-based fault-tolerance for parallel matrix equation solvers", Proc. SPIE, RTSP VIII, 1985(564), pp. 49-53.
- 189. Luk, F.T., Park, H., "An analysis of algorithm-based fault-tolerance techniques", Technical Report, EE-CEG-86-11, School of Electrical Engineering, Cornell Univ., 1986.
- 190. Margaritis, K.G., Evans, D.J., "Optical Gauss Elimination Algorithms using Matrix outer products", Proc. 7th Int. Congress of Cybernetics and Systems, London, 1987, pp. 622-630.
- 191. Margaritis, K.G., Evans, D.J., "Parallel Systolic LU Factorization for Simplex Updates", Proc. ICS '87, Athens, 1987.
- 192. Martin, A.R., Tucker, J.V., "The Concurrent Assignment Representation of Synchronous Systems", Technical Report, 887, Centre for Theoretical Computer Science, Dept. of Computer Sci., Univ. of Leeds, 1987.
- 193. May, D., Shepherd, R., Keane, C., "Communicating Process Architecture: Transputers and OCCAM", INMOS Ltd., 1986.
- 194. McAulay, A.D., "Optical Crossbar Signal Processor", Proc. SPIE, RTSP VIII, 1985(564), pp. 131-138.
- 195. McCanny, J.V., McWhirter, J.G., "Bit Level Systolic Array Circuit for Matrix Vector Multiplication", Proc.

IEE, 1983(130-G), pp. 125-130.

- 196. McCanny, J., McWhirter, J., "The derivation and utilization of bit level systolic array architectures", Proc. Int. Workshop on Systolic Arrays, Univ. of Oxford, 1986, pp. 47-59.
- 197. McCanny, J.V., et al., "The relationship between word and bit level systolic arrays as applied to matrixmatrix multiplication", Proc. SPIE, RTSP VI, 1983(441), pp. 114-120.
- 198. McWhirther, J.G., "Recursive Least-Squares Minimisation using a Systolic array", Proc. SPIE, RTSP VI, 1983(441), pp. 105-112.
- 199. Mead, C.A., Conway, L., "Introduction to VLSI Systems", Addison Wesley, 1980.
- 200. Megson, G.M., "Novel algorithms for the soft-systolic paradigm", Ph.D Thesis, Dept. of Computer Studies, LUT, 1987.
- 201. Megson, G.M., Evans, D.J., "Soft-Systolic Pipelined Matrix Algorithms", Proc. Parallel Computing '85, North-Holland, 1986, pp. 171-180.
- 202. Melhem, R., "Irregular Wavefronts in Data-driven, Data-dependent Computations", Proc. Int. Workshop on Systolic Arrays, Univ. of Oxford, 1986, pp. 303-312.
- 203. Melkemi, L., "Reseaux systoliques pour la resolution de problemes lineaires", Doctoral Thesis, Dept. of Applied Mathematics, University of Grenoble, France, 1986.
- 204. Melkemi, L., Tchuente, M., "Algebraic and Combinatorial Aspects of Systolic Algorithms for some Linear Problems", in "Parallel Algorithms and Architectures, Cosnard, M., et al. (eds.), North-Holland, 1986, pp. 269-279.
- 205. Melkemi, L., Tchuente, M., "Complexity of matrix product on orthogonally connected systolic arrays", (to appear in <u>IEEE</u>, <u>Trans. on Computers</u>, 1987).
- 206. Midwinter, J.E., "Light electronics, myth or reality?", Proc. IEE, 1985(132-J), pp. 371-383.
- 207. Moldovan, D.I., "On the Analysis and Synthesis of VLSI algorithms", <u>IEEE</u>, <u>Trans. on Computers</u>, 1982(C-31), pp. 1121-1126.
- 208. Moldovan, D.I., "A comparison between Parallel Processing of Numeric and Symbolic Algorithms", in "Parallel

Algorithms and Architectures", Cosnard, M., et al. (eds.), North-Holland, 1986, pp. 325-333.

- 209. Moldovan, D.I., Fortes, J.A.B., "Partitioning and mapping algorithms into fixed size systolic arrays", <u>IEEE</u>, <u>Trans. on Computers</u>, 1986(C-35), pp. 1-12.
- 210. Moler, C., Van Loan, C., "Nineteen dubious ways to compute the exponential of a matrix", <u>SIAM</u> <u>Review</u>, 1978(20), pp. 801-838.
- 211. Moraga, C., "On a case of symbiosis between systolic arrays", <u>INTEGRATION</u>, <u>the VLSI journal</u>, 1984(2), pp. 243-253.
- 212. Moraga, C., "Systolic Algorithms", Technical Report, Dept. of Computer Science, Univ. of Dortmund, F.R.G., 1984.
- 213. Moraga, C., "Design of a Multiple-valued Systolic System for the Computation of the Chrestenson Spectrum", IEEE, Trans. on Computers, 1986(C-35), pp. 183-188.
- 214. Nakamura, S., "Computational Methods in Engineering and Science, with applications to Fluid Dynamics and Nuclear Systems", John Wiley, 1977.
- 215. Navarro, J.J., Llaberia, J.M., Valero, M., "Computing Size-independent Matrix Problems on Systolic Array Processors", Proc. 13th Ann. Int. Symp. on Computer Architecture, Tokyo, 1986, pp. 271-278.
- 216. Nash, J.G., Hansen, S., "Modified Fadeev Algorithm for Matrix Manipulation", Proc. SPIE, RTSP VII, 1984(495), pp. 39-46.
- 217. Neff, J.A., "Major initiatives for optical computing", Optical Engineering, 1987(26), pp. 2-9.
- 218. Ohhashi, M., Schneider, R.E., "High-speed computation of Unary functions", Proc. 7th Symposium on Computer Arithmetic, 1985, pp. 82-85.
- 219. O'Keefe, M.T., Fortes, J.A.B., "A comparative study of two systematic design methodologies for Systolic Arrays", in "Parallel Algorithms and Architectures", Cosnard, M., et al. (eds.), North-Holland, 1986, pp. 313-324.
- 220. O'Leary, D.P., "Ordering Schemes for parallel processing of certain mesh problems", <u>SIAM J. Sci. Stat. Com-</u> put., 1985(5), pp. 620-632.

221. O'Leary, D.P., "Systolic Arrays for Matrix Transpore

- 544 -

and other reorderings", <u>IEEE</u>, <u>Trans</u>. <u>on</u> <u>Computers</u>, 1987(C-36), pp. 117-122.

- 222. Owechko, Y., et al., "Representation of Bipolar and Complex Data in the PRIMO Optical Matrix Multiplier", Proc. SPIE, Optical Computing, 1986(625), pp. 72-78.
- 223. Pan, V., Reif, J., "Efficient Parallel Solution of Linear Systems", Proc. 17th Annual Symposium on Theory of Computing, 1985, pp. 143-152.
- 224. Papadopoulou, E.P., "VLSI Structures and Iterative Analysis for Large Scale Computation", Ph.D. Thesis, Dept. of Mathematics and Computer Science, Clarkson Univ., 1986.
- 225. Parlett, B.N., "Laguerre's method applied to the Matrix Eigenvalue problem", <u>Maths. Comput</u>. 1964(18), pp. 464-485.
- 226. Parlett, B.N., "Remarks on Matrix Eigenvalue Computations", in "VLSI and Modern Signal Processing", Kailath, T., et al. (eds.), Prentice-Hall, 1985, pp. 106-120.
- 227. Peters, G., Wilkinson, J.H., "Practical Problems arising in the solution of polynomial equations", J.Inst. Math. Applic., 1971(8), pp. 16-35.
- 228. Petkov-Turkedjiev, N., "Synthesis of Systolic Algorithms and Processor Arrays", Proc. CONPAR '86, Springer-Verlag, 1986, pp. 165-172.
- 229. Philippe, B., "Approximating the Square Root of the Inverse of a Matrix", Technical Report, 508, Centre for Supercomputing R&D, Urbana IL., 1985.
- 230. Phillips, G.M., Taylor, P.J., "Theory and Applications of Numerical Analysis", Academic Press, 1973.
- 231. Pizer, S.M., "Numerical Computing and Mathematical Analysis", Science Research Associates, 1975.
- 232. Priester, R.W., et al., "Problem adaptation to systolic arrays", Proc. SPIE, RTSP IV, 1981(298), pp. 33-39.
- 233. Priester, R.W., et. al., "Signal Processing with Systolic Arrays", Proc. Int. Conf. Parallel Processing, 1981, pp. 207-215.
- 234. Psaltis, D., "Two-dimensional optical processing using one-dimensional input devices", <u>Proc. IEEE</u>, 1984(72), pp. 962-974.

- 235. Psaltis, D., Athale, R.A., "High accuracy computation with linear analog optical systems: a critical study", Applied Optics, 1986(25), pp. 3071-3077.
- 236. Quinton, P., "The systematic design of Systolic Arrays", Technical Report, 193, IRISA, France, 1983.
- 237. Quinton, P., Jannault, B., Gachet, P., "A new matrix multiplication systolic array", in "Parallel Algorithms and Architectures", Cosnard, M., et al. (eds.), North-Holland, 1986, pp. 259-268.
- 238. Ramakrishnan, I.V., Fussell, D.S., Silbersachatz, A., "Mapping Homogeneous Graphs on Linear Arrays", <u>IEEE</u>, <u>Trans. on Computers</u>, 1986(C-35), pp. 189-209.
- 239. Ramamoorthy, P.A., Chen, T., "Systolic Architectures based on Barrel Shifters for Real-Time Signal and Image Processing", Proc. IEEE ICASSP, 1985, pp. 26.9.1-26.9.4.
- 240. Rao, S.K., Kailath, T., "What is a Systolic Algorithm?", Proc. SPIE, Highly Signal Processing Architectures, 1986(614), pp. 34-48.
- 241. Rhodes, W.T., Guilfoyle, P.S., "Acoustooptic Algebraic Processing Architectures", <u>Proc. IEEE</u>, 1984(72), pp. 820-830.
- 242. Robert, Y., "Block LU Decomposition of a Band Matrix on a Systolic Array", <u>Int. J. Comput. Math.</u>, 1985(17), pp. 295-315.
- 243. Robert, Y., "Systolic Algorithms and Architectures", Technical Report 397, IMAG, France, 1986.
- 244. Robert, Y., Tchuente, V., "Designing efficient systolic algorithms", Technical Report, 393, IMAG, France, 1983.
- 245. Robert, Y., Trystram, D., "An Orthogonal Systolic Array for the Algebraic Path Problem", Technical Report, 553, IMAG, France, 1985.
- 246. Rogers, M.H., "Specification of Algorithms for Systolic Array Elements", in "VLSI Architecture", Randell, B., et al. (eds.), Prentice-Hall, 1983, pp. 212-224.
- 247. Rosenberg, A.L., "Diogenes, Circa 1986", Proc. AWOC '86, Springer-Verlag, 1986, pp. 109-118.
- 248. Rote, G., "A Systolic Array Algorithm for the Algebraic Path Problem (Shortest Paths; Matrix Inversion)", <u>Com</u>puting, 1985(34), pp. 191-219.

- 249. Rote, G., "On the connection between hexagonal and unidirectional rectangular Systolic Arrays", Technical Report, 86-71, Institute of Mathematics, Technical Univ. of Graz.
- 250. Sami, M., Stefanelli, R., "Reconfigurable Architectures for VLSI Processing Arrays", Proc. IEEE, 1986(74), pp. 712-722.
- 251. Samwell, P.M., "Experience with OCCAM for simulating systolic and wavefront arrays", <u>Software Engineering</u> Journal, 1986(5), pp. 196-204.
- 252. Saridakis, Y.G., "Parallelism, Applicability and Optimality of Modern Iterative Methods", Ph.D. thesis, Dept. of Mathematics and Computer Science, Clarkson Univ., 1985.
- 253. Savage, C., "A Systolic Data Structure Chip for Connectivity Problems", in "VLSI Systems and Computations" Kung, H.T., et al. (eds.), CMU, 1981, pp. 296-300.
- 254. Sawchuk, A.A., Jenkins, B.K., "Dynamic Optical Interconnections for Parallel Processor", Proc. SPIE, Optical Computing, 1986(625), pp. 143-153.
- 255. Scheid, F., "Numerical Analysis", McGraw-Hill, 1968.
- 256. Schmeck, H., "A comparison-based instruction systolic array", in "Parallel Algorithms and Architectures", Cosnard, M., et al. (eds.), North-Holland, 1986, pp. 281-292.
- 257. Schreiber, R., "Systolic arrays for eigenvalue computation", Proc. SPIE, RTSP V, 1982(341), pp. 26-34.
- 258. Schreiber, R., "On the Systolic Arrays of Brent, Luk and Van Loan", Proc. SPIE, RTSP VI, 1983(441), pp. 72-76.
- 259. Schreiber, R., "Computing Generalised inverses and eigenvalues of symmetric matrices using systolic arrays", in "Computing methods in Applied Sciences and Engineering, VI", Glowinski, R., et al. (eds.), North-Holland, 1984, pp. 285-295.
- 260. Schreiber, R., "Implementation of Eigenvector Methods", Proc. SPIE, RTSP VII, 1984(495), pp. 3-6.
- 261. Schreiber, R., "On Systolic Array Methods for Band Matrix Factorizations", <u>BIT</u>, 1986(26), pp. 303-316.
- 262. Schreiber, R., Kuekes, P.J., "Systolic Linear Algebra Machines in Digital Signal Processing", in "VLSI and

Modern Signal Processing", Kailath, T., et al. (eds.), Prentice-Hall, 1985, pp. 389-405.

- 263. Schreiber, R., Tang, W.P., "On Systolic Arrays for updating the Cholesky Factorization", <u>BIT</u>, 1986(26), pp. 451-466.
- 264. Seitz, C.L., "Concurrent VLSI Architectures", <u>IEEE</u>, Trans. on <u>Computers</u>, 1984(C-33), pp. 1247-1265.
- 265. Serbin, S.M., Blalock, S.A., "An Algorithm for Computing the Matrix Cosine", <u>SIAM J. Sci. Stat. Comput.</u>, 1980(1), pp. 198-204.
- 266. Shapiro, E., "Systolic Programming: a paradigm of Parallel Processing", Technical Report, CS84-16, Weizmann Institute of Science, Rehovot, Israel, 1984.
- 267. Shepherd, T., McWhirter, J., "A systolic array for linearly constrained least-squares optimisation", Proc. Int. Workshop on Systolic Arrays, Univ. of Oxford, 1986, pp. 151-159.
- 268. Singer, B., Spilerman, S., "The representation of social processes by Markov Models", <u>American J. of</u> <u>Sociology</u>, 1976(28), pp. 1-55.
- 269. Snyder, L., "Introduction to the configurable highly parallel computer", <u>IEEE</u>, <u>Computer</u>, 1982(15), pp. 47-56.
- 270. Snyder, L., "Supercomputers and VLSI: the effect of LSI on Computer Architecture". Advances in Computers, 1984(23), pp. 1-33.
- 271. Snyder, L., "Programming Environments for Systolic Arrays", Proc. SPIE, Highly Parallel Signal Processing Architectures, 1986(614), pp. 134-144.
- 272. Sorensen, D.C., "Analysis of pairwise pivoting in Gaussian Elimination", <u>IEEE</u>, <u>Trans</u>. on <u>Computers</u>, 1985(C-34), pp. 274-278.
- 273. Speiser, J.M., Whitehouse, H.J., "Parallel Processing Algorithms and Architectures for Real-Time Signal Processing", Proc. SPIE, RTSP IV, 1981(298), pp. 2-9.
- 274. Speiser, J.M., Whitehouse, H.J., "A Review of Signal Processing with Systolic Arrays", Proc. SPIE, RTSP VI, 1983(441), pp. 2-6.
- 275. Stallard, R.P., "OCCAM: a brief introduction OCCAM: The Loughborough Implementation", Technical Report, Dept. of Computer Studies, LUT, 1985.

- 276. Stallard, R.P., "Loughborough OCCAM Compiler, Version 5.0 Documentation", Technical Report, Dept. of Computer Studies, LUT, 1986.
- 277. Swartzlander, E.E., "VLSI Architecture", in "VLSI Fundamentals and Applications", Barbe, D.F. (ed.), Springer-Verlag, 1982, pp. 178-221.
- 278. Symanski, J.J., "Implementation of Matrix Operations on the two-dimensional Systolic Array Testbed", Proc. SPIE, RTSP VI, 1983(441), pp. 136-142.
- 279. Takeda, M., Goodman, J.W., "Neural networks for computation: numbers representations and programming complexity", <u>Applied Optics</u>, 1986(25), pp. 3033-3046.
- 280. Tee, G.J., "An application of p-cyclic Matrices for solving periodic parabolic problems", <u>Numer. Math.</u>, 1964(6), pp. 142-159.
- 281. Tenorio, M.F.M., Moldovan, D.I., "Mapping production systems into Multiprocessors", Proc. Int. Conf. on Parallel Processing, 1985, pp. 56-62.
- 282. Todd, J. (ed.), "Survey of Numerical Analysis", McGraw-Hill, 1962.
- 283. Tomlin, J.A., "Modifying triangular factors of the basis in the Simplex Method", in "Sparce Matrices and their Applications", Rose, D.J., et al. (eds.), Plenum Press, 1972, pp. 77-85.
- 284. Uchida, S., "Toward a new generation of computer architecture", in "VLSI Architecture", Randell, B., et al. (eds.), Prentice-Hall, 1983, pp. 395-423.
- 285. Ullman, J.D., "Computational aspects of VLSI", Computer Science Press, 1984.
- 286. Umeo, H., "A class of SIMD Machines simulated by Systolic VLSI Arrays", in "VLSI Algorithms and Architectures", Bertolazzi, P., et al. (eds.), 1985, pp. 39-48.
- 287. Urquhart, R.B., Wood, D., "Systolic matrix and vector multiplication methods for signal processing", <u>Proc</u>. IEEE, 1984(131-F), pp. 623-631.
- 288. Van Loan, C., "A Note on the Evaluation of Matrix Polynomials", <u>IEEE</u>, <u>Trans. on Automatic</u> <u>Control</u>, 1979(AC-24), pp. 320-321.
- 289. Varga, R.S., "Matrix Iterative Analysis", Prentice-Hall, 1962.

- 290. Verber, C.M., "Integrated-optical approaches to numerical optical processing", <u>Proc.</u> <u>IEEE</u>, 1984(72), pp. 942-953.
- 291. Verber, C.M., "Integrated optical architectures for matrix multiplication", <u>Optical Engineering</u>, 1985(24), pp. 19-25.
- 292. Verber, C.M., et. al., "Suggested integrated optical implementation of pipelined polynomial processors", Proc. SPIE, Optical Information Processing, 1983(388), pp. 221-227.
- 293. Wallach, Y., "Alternating Sequential-Parallel Processing", Lecture Notes in Computer Science, 127, Springer-Verlag, 1982.
- 294. Ward, J.S., McCanny, J.V., McWhirter, J.G., "A Systolic Implementation of the Winograd Fourier Transform Algorithm", Proc. IEEE ICASSP, 1985, pp. 38.5.1.-38.5.4.
- 295. Weiser, U., Davis, A., "A wavefront notation tool for VLSI Array design", in "VLSI Systems and Computations", Kung, H.T., et al. (eds.), Computer Science Press, 1981, pp. 226-234.
- 296. Wilkinson, J.H., "Calculation of the eigenvectors of a symmetric tridiagonal matrix by inverse iteration", Numer. Math., 1962(4), pp. 368-376.
- 297. Wilkinson, J.H., "Rounding Errors in Algebraic Processes", N.P.L. Notes on Applied Science, No. 32, H.M.S.O., London, 1963.
- 298. Wilkinson, J.H., "The Algebraic Eigenvalue Problem", Claredon Press, Oxford, 1965.
- 299. Yalamanchili, S., Aggarwal, J.K., "Reconfiguration strategies for parallel architectures", <u>IEEE</u>, <u>Computer</u>, 1985(18), pp. 44-61.
- 300. Young, D., "Iterative solution of large linear systems", Academic Press, 1971.
- 301. Young, T.Y., Liu, P.S., "VLSI Arrays for Pattern Recognition and Image Processing: I/O Bandwidth Considerations", in "VSLI for Pattern Recognition and Image Processing", Fu, K.S. (ed.), Springer-Verlag, 1984, pp. 25-42.
- 302. Yung, H.C., et al., "A recursive design methodology for VLSI: theory and example", <u>INTEGRATION</u>, the VLSI journal, 1984(2), pp. 213-225.

303. Zak, S.H., Hwang, K., "Polynomial division on Systolic Arrays", <u>IEEE</u>, <u>Trans. on Computers</u>, 1985(C-34), pp. 577-578.

· · · · ·

,

APPENDIX

This Appendix comprises a brief introduction to the OCCAM programming language, followed by the description of the Loughborough implementation of OCCAM. Further, there is a selection of programs, simulating some of the systolic designs discussed in this thesis.

I. BRIEF INTRODUCTION TO OCCAM

In OCCAM processes are connected to form concurrent systems, each process can be regarded as a black box with an internal state which can communicate with other processes via point to point communication channels. The processes themselves are finite. Each process starts, performs a number of actions then terminates. An action may be a set of parallel processes to be performed at the same time. As a process is itself composed of processes which may themselves be executed in parallel a process allows internal concurrency which varies with time.

Processes: All processes are constructed from three primi-

tive processes, assignment, input and output. An assignment is indicated by the symbol ':=', for example, v:=e sets variable v to the value of the expression e and then terminates. An input is indicated by the symbol '?', for example, c?x inputs a value from a channel c assigning it to x and then terminating. An output is indicated by the symbol '!' and cle outputs the expression e to channel c, and then terminates.

A pair of concurrent processes communicate using a one way channel connecting the two processes. One process outputs a message to the channel, the other process inputs the message from the channel. A particular process can be ready to communicate on one or more of its channels any time between its start and termination, but a communication only takes place when both it and the process sharing one of its channels is ready. Where a number of connected processes are ready simultaneously communication can occur in parallel.

<u>Constructs</u>: A number of processes can be combined to form a construct which is itself a process and can be used as a component for other constructs. Each component process is indented by two spaces from the left hand margin indicating which construct it is part of. There are only four basic construct types, sequential, parallel, conditional, and alternative.

SEQ: is the keyword for a sequential construct denoted

- 553 -

seq P1 P2 P3

where the component processes p_1, p_2, p_3, \ldots are executed in strict sequence with process p_i finishing before p_{i+1} starts and after p_{i-1} terminates. Sequential constructs are similar to programs written in conventional programming languages.

PAR: is the keyword for a parallel construct of the form

par P1 P2 P3

and in contrast to seq, here all the component processes P_1, P_2, P_3, \ldots are executed concurrently. The par construct terminates when all the component processes have finished.

IF: is the keyword for a conditional construct with the appearance

if condition 1 condition 2 P₂

This means that p_1 is executed if condition 1 is true, otherwise p_2 iff condition 2 is true, etc. Notice the strict

sequential ordering of tests. Only one of the processes p_i is executed and the 'if' construct terminates when the process finishes.

ALT: is the keyword for the alternative construct

This construct waits until one of input 1, input 2, . . is ready. If input 1 is ready first, input 1 is performed and on completion p_1 is executed. Similarly if input i is ready first input i is performed and p_i is executed. Only one of the inputs is performed and its corresponding process executed before the construct terminates. If more than one input becomes ready at the same time the one executed is chosen arbitrarily.

<u>Repetition</u>: There is only one explicit construction for repetition denoted by

while condition p

which repeatedly executes process p until the value of the condition is false. Observe that p itself can be a composition of sequential and parallel constructs.

<u>Replication</u>: A replicator is used with a constructor to replicate the component process a number of times. With

'seq' a standard for loop

seq i=[0 for n]
p

is created executing process p sequentially n times. When used with 'par' an array of concurrent processes with the form

```
par i=[0 for n]
    p;
```

is created such that n similar processes p_0, p_1, \dots, p_{n-1} are executed in parallel. Notice that i=0(1)n-1 not n, thus if generally i=[base for count] there are base+count-1 values i takes starting with i=base.

Declarations: A declaration introduces a new identifier for use in the process that follows it, and defines the meaning the identifier will have within the process. If the new identifier is the same as one already in use, all subsequent occurrences of the identifier in the process will refer to the meaning of the most recent declaration. Declarations are of four basic types 'var', 'chan', 'def' and 'proc' linked following process by a colon (:) at the last line of to a the declaration. The process follows on the next line at the indentation as the keyword declaration. For same level of example:

var x: P - 556 -

declares variable x to be used in process p, and

```
chan c:
p
```

defines a channel c to be used in communication for p. A variable vector declaration introduces an identifier to be used as a vector of variables, viz.

```
var list [16]:
p
```

for a vector named list of 16 variables indexed as list[0], list[1], ..., list[15]. Likewise a channel vector declaration introduces a new identifier as a vector of channels for communicating between concurrent processes

```
chan c[n]:
p
```

'Def' associates a name with a constant value, or with a table of constant values, e.g.

```
def a=1, b=2:
```

associating a with 1 and b with 2, using these identifiers within a process yields the associated values.

The 'proc' declaration introduces an identifier to name the process which follows, indented, on the succeeding lines. The process is termed the named process and is itself followed by a process in which the named process will be used. The named process can have parameters which are declared with the declaration of the named process and are called formal parameters. The named process text will be substituted for all occurrences of the process name in subsequent processes, the 'var' and 'chan' variables substituted in place of the formal parameters are called actual parameters. For example,

```
proc buffer(chan in, out) =
  while true
    var x:
    seq
    in?x
    out!x :
chan c,c1,c2 :
    par
    buffer(c1,c)
    buffer(c,c2)
```

declares two buffer processes executed concurrently, buffer the named process with formal channel parameters in and is out. In the following process c_1, c_2 , are actual parameters and on execution the 'while' loop will replace the procedure calls and c,c1,c2 will replace in and out channels. The size a vector is not specified in the formal parameters of a of named process and different sized vectors may be used as actual parameters on different substitutions. In addition to the standard declarations 'var' and 'chan', a 'value' parameter may also be used, as either an ordinary or vector formal parameters and cannot be changed within a process by assignment or input.

Finally, an identifier which is used but not declared in a named process is termed a free identifier. Any free

- 558 -

identifier in use when a named process substitution takes place must be the same as a variable already in use. The free variable then takes on the most recent incarnation of the variable at the point where the process substitution takes place.

<u>Program</u> Format: In OCCAM indentation from the left hand margin indicates program structure. Each process starts on a new line, at an indentation level indicated by the following rules:

<u>Constructs</u>: The construct keyword (and the optional replicator) occupies the first line. Each of the component processes start on a new line and are indented by two spaces more than the keyword.

<u>Conditionals</u>: The condition expression occupies the first line, and the component process starts on the next line indented by two more spaces.

<u>Alt inputs</u>: The expression and its associated input occupy the first line and the component process starts on the next line indenting two more spaces.

<u>Declarations</u>: Each declaration starts on a new line, at the same level of indentation as the process it prefixes, the final line of the declaration being terminated by a colon. Blank lines can be inserted anywhere and are ignored.

A construct can be broken to occupy more than one line,

- 559 -

with line breaks occurring after comma, semicolon and before the second operand of an operator (requiring two operands). The continued line must be more indented than the first line of the construct.

<u>Comments</u>: Comments are denoted by double hyphen (--) and terminate at the end of a line. All characters of a comment are ignored. A comment may follow an OCCAM construct on the same line or be on a line by itself.

This summary of OCCAM is taken from [140], [152] and 'proto-OCCAM'. A more sophisticated version implements is now available providing Real, Integer, and OCCAM-2 Boolean types, as well as 2-dimensional arrays. We remark that the programming in this thesis was performed on a VAX machine under UNIX using Loughborough OCCAM as implemented by R.P. Stallard, [275-276]. The Loughborough version of OCCAM, is now discussed, and particularly its extensions of proto-OCCAM to allow real variables and non-standard OCCAM features. Then, a selection of programs, simulating some of the systolic designs discussed, is given.

- 560 -

II. LOUGHBOROUGH IMPLEMENTATION OF OCCAM

Help for running the occam compiler

A source 'occam' file (OOCAM and INMOS are trademarks of the INMOS group of companies) must be of the form '*.occ', to compile it to form an 'a.out' command file use the default options. For example to compile 'my_first.occ' :-

occam my_first.occ

An executable object 'a.out' is produced. As a shortcut you can omit the '.occ' affix and just say 'occam my_first', the compiler will add on the affix for you.

If a program is split into several files these can be separately compiled and linked together using the occam compiler and built in linker.

Each previously compiled occam program is specified in the command line in the form '*.o' e.g. ;-

occam main.occ numericlib.o screenlib.o

This will compile the source of 'main' and link it in with the pre compiled library occam files 'numericlib.occ' 'screenlib.occ'. The -l option is used to generate new versions of library file objects.

Various switch options are provided, mainly for compiler debugging. Plage can either be put separately ('-g - 1') or together and in any order ('-lg', '-gl'). The following switches may be useful :-

-8:

occam -g fast.occ

Compile the occam program as before but run the resulting program immediately (a compile, load and go option). If flag options are specified that apply to the run of the program these will be passed on as in 'occam "goc fast'.

-1 :

occam -1 new lib

Compile the program and produce object but do not link the object files together to produce an object program. This option is used for building up libraries of routines or to cut down the compilation time for compiling one long program.

-0:

occam keep_it -o saverun

Ocmpile the program as normal but place the object program in the file 'saverun' rather than the default 'a.out'. Useful for saving several occam object files at the same time.

-x :

occam -x old fashioned.occ

Compile according to the strict Inmos occam specification, LUT extensions (see file 'occamversion') currently include :-

Multiple source file cross linking. Dynamic features. Variable PAR replicator counts. Floating point arithmetic.

-c :

a.out ~c

Num the object program with cursor addressable facilities enabled, the standard library procedures 'goto.x.y' and 'clear.screen' require these facilities.

-G: occan -G error prone

Compiles the file as normal but generates a symbol file as well (in this case it would be 'error_prone.sym'), this is used by the run-time system to inspect the values of variables.

-a :

a.out -q

Run the object program without producing any characters to the screen other than those output by the program (unless CTRL c used). This enables occars programs to dump output that can be processed by other occam programs.

-P and -N :

occam -F num.occ

"-P" Includes the floating point library routines to provide a simple real number arithmetic capability. "-N" includes both the floating point and mathematical library routines to provide mathematical library routines.

-I :

This provides the features of the Inmos proto-occam definition (see 'occam version') such as STOP and TIME, it should be used where possible as it is closer to the occam-2 definition.

Full list of compiler option flags

The full (often cryptic) range of switch options are as follows. Several switch flags can be given, in any order and either separately or together. The mnemonic character giving the switch is highlighted by a capital letter.

They are divided into sections - user defined flags, and system defined options, which are selected by prefixing with '.

User Plags

- -% The next flag(s) are system flags switch flag mode.
- -c Run the program with Cursor addressable options enabled. The library routines 'clear.screen' and 'goto.x.y' need this flag set. If used for the compiler must also give the -g option.
- -e Produce object/run object for Execution tracing. The resulting object file is then run with the '-e' option. This utility is described in 'tracerinfo'.
- -f Force full occam semantic cbeck on use of variables. A variable (not vectors though) can not be set within a PAR construct if the declaration is outside the PAR. This applies equally to procedure calls that change global variables.
- -g Run the resulting object file if compilation succeeded. The program Goes immediately it is ready to.
- -h Print out this 'Help' information.
- -1 Force an Interrupt immediately before start of execution immediately displays the debug help menu. This enables break and trace points to be setup prior to anything being executed.
- -1 Compile but do not link the occam source. Needed when using multiple occam source Library files.
- -m Check that every channel Match properly on execution, channels can have only one input and one output process during execution.
- -o Produce an Object program with name given by the non-switch argument following this switch. Enables you to choose an object file name other than 'a.out'.
- -q Run the program without outputting some non occam program produced messages - e.g. 'OCCAM Start Run'. Must give -g option as well 'q' stands for Quiet. Useful when producing output to be piped or processed by other programs.

- -w Suppress the Warning messages from the compiler when you have seen these warnings once you may find it less irritating to suppress them on subsequent compilations - does not affect error reporting or any other compiler action.
- -x Do not permit any local LUT extensions in the source text. See 'occinfo' for information about these - for example recursion and EXTERNAL procedure definitions. Useful if moving an occam program for use on another occam compiler system.
- -F Include the standard Ploating point library routines. Provides routines to read or write floating point routines to channels.
- -G Produce a symbol table file (with affix '.sym') for use with the 'm' option in the dynamic debugger for symbol value examination.
- -I Permit the use of INMOS proto-occam version 2. These changes include the use of 'TIME' instead of 'NOW', the 'STOP' primitve and the use of 'Stopping IP' - an alternative without any TRUE conditions will STOP.
- -L Use Long winded load, all the 'C' libraries are added at the last momment rather than using the pre-linked object, this may be useful if a user occam/C library calls a 'C' routine that is not used in the occam run time system. See 'libraryhelp' for more info.

CT I

σ

N

- -M Include the Mathematical library and floating point routines.
- -O Produce optimized object. May improve run time by 20%.
- -R Use Randomized scheduling when running the program the same scheduler choices will not be made on separate executions. This gives non-deterministic execution and will be slightly slower but may be useful occasionally.
- -S Do not include the Standard I/O routines with the object. This library is included by default, there is no reason not to want to include it unless you want to devise a totally new one.
- -T The next argument is a Timing definition file built by the 'timebuild' utility to be used in conjuntion with the '-e' option, supplying '-T' automatically selects '-e'. If this option is not selected the execution timings are taken from the source library file 'times'. Look at the 'timerinfo' help file for more details.
- -V The compiler will normally desist reporting errors and warnings after the first fifty or so, with this option all the errors will be reported. May produce Very Verbose output.
- -W Give Warning messages about declarations that turn out not to have been used at all. This may highlight misspelt declarations or existence of no longer used procedures.

System Flags

- -5 Switch back to expecting 'user' mode flag options. This means you can replace -C%v by -%v%G.
- Enable Analysis of the usage of channels this facility is still -2 under test. -n
- Check the source occam for syntax errors, but do Not produce any object data from it.
- Print out the program in the form just after it was Transformed. -t Not generallt useful as the program has changed so much.
- Give Verbose information at each stage when running the compiler -will print out a more accurate description of the system commands it is calling and all the files it accesses. Also switches on a full print out of the occam link information.

- Produce the object code ('C' or Assembler) in a permanent file -A so that it can be inspected.
- Produce 'C' rather than assembler output from the occam compiler -C then compile and link it. There will be *.o and *.c containing the object and compiler generated source created in the directory. The 'C' and assembler code produced will be similar and there is little point in producing 'C' unless to waste time ! (as the 'C' compilation phase takes a long time). If the compiler is ported to a non-VAX system then this option will automatically be solected.
- Switch on variable name and line number Dumping in the C/Assembler -D 'object' file so that the object can be tied in with the source.
- -H Undocumented feature under test.
- Produce an occam-'C' interface Library, the two files ending '-c.c' and -L '.occ' are linked together, the occan can refer directly to the 'C' routines.
- Run the compiler showing the steps it would execute but without -N actually doing anything - like '-n' in the UNIX 'make' command. Useful when options start getting complicated. A No operation facility.

Undocumented feature under test. -Q

- -8 Do not apply some Simplifying transformations on the program. These currently remove constructs with no processes in them and redundant SEQ and PAR headers. These save a small amount of space and time at run and compile time and there is little point in turning off this option.
- -X-Print out the procedures that have been defined in the link files but has not been referenced - detects extra procedures defined across files but not used.
- Produce the linker assembler output in a permanent file rather than -Y in a temporary file on '/tmp'. Enables the output from the linker to be debugged. -Z
- Get the linker to print out all the definitions it is told about.

Description of the library routines

Standard Library

Provide commonly used routines to read and write to the keyboard and screen channels. The routines are written in 'C' and occam and use standard C or 'curses' I/O routines. There are also general routines for use to pause or abort a program as well as to use the 'C' random number routines. They are available by default to all programs unless the -S compiler flag is used to override their inclusion.

EXTERNAL PROC str. to.screen (VALUE s []) :

Output the string s (a byte array with byte 0 as the length). The whole string is guaranteed to be printed in one sequence. two concurrent calls to str. to.screen will not interleave. Equivalent to the program fragment :-

t

UT

٥, ω

PROC str.to	•••	BCreen (VALUE s []) =	
_SEQ_n_=[1	for a [BYTE 0]]	
SC reen	1	a (BYTE n) :	

EXTERNAL PROC num. to.screen (VALUE n) :

Output a number to the screen. The number can be signed, and uses the minimum number of characters (no leading spaces). Equivalent to the 'C' language 'printf ("%d",n);' statement.

EXTERNAL PROC str. to.chan (CHAN o.VALUE & []) :

Output the string s to a channel 'c'. The call 'str.to.chan (screen, "fred")' is identical to 'str.to.screen (fred)'. Useful for string output to files.

EXTERNAL PROC num.to.chan (CHAN c, VALUE n) :

Output ascil string for the number 'n' to channel 'c'. Like 'str.to.chan' but for numbers not channels.

EXTERNAL PROC num. to.screen.f (VALUE n,d) :

Output a number to the screen in a field of width 'd'. If the number is too big for the field the number is written out in full regardless, the routine call num.to.screen.f (n.1) is equivalent to num.to.screen (n). The routine uses the 'C' language printf format and where a is the field width.

EXTERNAL PROC goto.x.y (VALUE x,y) :

Use the 'curses' package to implement a cursor 'goto' facility. No error checking is made that the move is within the screen area. The x-axis is across the screen and y-axis down, co-ordinate (0,0) is in the top left hand corner of the screen. The first line is used by the run time system to print messages.

EXTERNAL PROC clear.screen :

Use curses to clear the screen, if cursor addressable option not used this will still try to clear the screen using the curses "CL" termcap defined string.

EXTERNAL PROC num.from.keyboard (VAR n) :

Read a number from the keyboard and assign to variable 'n'. The routine is not very sophisticated. It will read negative numbers (start '-') and ignore any leading 'space' characters. The number must be followed by a non-digit, this character is read by the routine and not available on a subsequent 'Keyboard? ch' process. There is no check that the number is too big for the number range. It will expect at least one digit otherwise it will give an error message.

EXTERNAL PROC num.from.chan (CHAN c.VAR n) ;

Read a number from a channel 'c'. If 'c' is the keyboard this is equivalent to calling 'num.from.keyboard'.

ECTERNAL PROC abort.program :

Force the program to abort execution. An explanatory message is printed so that the cause will be known.

EXTERNAL PROC force.break :

Perform the same action as if 'CTRL-C' was pressed at the terminal. The user interface routines can then be run under the menu selection facility provided.

EXTERNAL PROC random (VALUE d, VAR n) :

Return a pseudo random number in the range 0 to d-1 by using the 'C' 'random ()' function in the variable n. The VALUE of d must not be zero. The sequence of random numbers will be modified if the '-R' run option is used.

EXTERNAL PROC init.random (VALUE n) :

Initialise the seed for the random number generator for subsequent calls to the procedure 'random'. Uses the 'C' language routine 'srandom ()'. EXTERNAL PROC trace.value (VALUE n) :

Print out the integer value of 'n' on the screen with the prefix string 'Trace value : ' - this makes debugging a little easier.

EXTERNAL PROC open.file (VALUE path.name [], access [], CHAN io.chan) :

Connect the channel 'io.chan' to a UNIX file. The procedure must be provided with the pathname of the file as a string, and the access mode ("r" read access,"w" write access,"a" append access). Subsequent input or output on 'io.chan' will fetch/put a single character from/to the file. Attempts to input past the end of file will receive the value -1.

EXTERNAL PROC close.file (CHAN io.chan) :

Cease connection of the channel with its currently open file.

EXTERNAL PROC open.pipe (VALUE command.name [],access [],CHAN io.chan) :

Connect the channel 'io.chan' to a UNIX pipe running command 'command.name'. The procedure must be provided with the UNIX command name and 'r' to read from it, or 'w' to write to it). Subsequent input or output on 'io.chan' will fetch/put a single character from/to the file. Attempts to input past the end of file will receive the value -1.

EXTERNAL PROC close.pipe (CHAN io.chan) :

Cease connection of the channel with its currently active command.

EXTERNAL PROC system.call (VALUE command [], VAR code) ;

Execute the UNIX command contained in the string 'command' and return the value in 'code' TRUE if the command succeeded without error and FALSE otherwise.

EXTERNAL PROC set.timers (VALUE init.value) ;

Set up the interval timers ITIMER REAL, ITIMER VIRTUAL to the given start value. These are used for timing sections of code on the VAX. Uses 'setitimer' call. Note that using 'WAIT' primitive will reset the timer so it can only be used for simple sections of code. It should also be noted that it times the whole program and not a single occam process.

EXTERNAL PROC get.real.timer (VAR secs,micro.secs) :

Get the current elapsed timer values in seconds and microseconds. Timers count downwards and are not especially accurate. Uses 'getitimer' call.

EXTERNAL PROC get.cpu.timer (VAR secs,micro.secs) :

Get the current executed CPU timer values in seconds and microssconds. Timers count downwards and are not especially accurate.

1

Floating Point Library

Routines to perform floating point input/output. They are available by giving the compiler flag '-P' when linking an occam program.

Ploating point value can be assigned and transmitted via channels just like normal integer values, see the file 'occanversion' for details as to the language extensions introduced to support them.

Input/Output Routines

EXTERNAL PROC fp.num.to.screen (VALUE FLOAT f) :

Print out the floating point number in 'C' language float format "16.6f". If the number is too small or too big the standard 'C' action will be taken.

EXTERNAL PROC fp. num. to.screen.f (VALUE FLOAT f, VALUE *.d) :

Print out the floating point number in 'C' real format "Tw.df". If the number is too small or too big problems will arise.

EXTERNAL PROC fp.num.to.screen.g (VALUE FLOAT f) :

Print out the floating point number in 'C' real format "Xg". This will use the most appropriate format - exponent form if necessary.

EXTERNAL PROC fp.num.to.chan (CHAN c, VALUE FLOAT f) :

Write a number to a channel. If channel is 'screen' this is equivalent to 'fp.num.to.screen'. Useful for writing data to files.

EXTERNAL PROC fp.num.from.keyboard (VAR FLOAT f) :

Read in a floating point number. The number is expected to begin with a digit or '.' (indicating 0.), leading spaces are ignored. The number ends on a non-digit and this character will not be available to subsequent reads from the keyboard channel. The following are valid input numbers followed by the interpreted value for the input.

45.35 (45.35) 0.0004 (0.0004) .0 (0.0) 1. (1.0) 124 (124.0)

EXTERNAL PROC fp.num.from.chan (CHAN c, VAR FLOAT f) :

Read a floating point number from a channel 'c'. If channel is keyboard this is equivalent to 'fp.num.from.keyboard'.

Mathematical Routine Library

double procision 'C' routines are called.

Return e to the power 'a' in 'res'.

Natural logarithm of 'a' in 'res'.

Mathematical routines from the UNIX '-lm' library. These are included by specifying the '-M' flag. They are all in single precision even though EXTERNAL PROC (p.sine (VALUE FLOAT a, VAR FLOAT res) : Return the sine of 'a' in 'res'. Angles are in radians. EXTERNAL PROC fp.coside (VALUE FLOAT a, VAR FLOAT res) : Return the cosine of 'a' in 'res'. Angles are in radians. EXTERNAL PROC fp.arc.side (VALUE FLOAT a, VAR FLOAT res) : Return the arc sine of 'a' in 'res'. Angles are in radians. EXTERNAL PROC fp.arc.cosine (VALUE PLOAT a, VAR FLOAT res) : Return the arc cosine of 'a' in 'res'. Angles are in radians. EXTERNAL PROC fp.arc.tan (VALUE FLOAT a, VAR FLOAT res) : Return the arc tangent of 'a' in 'res'. Angles are in radians. EXTERNAL PROC 1p.exp (VALUE FLOAT a, VAR FLOAT res) : EXTERNAL PROC fp.log (VALUE FLOAT a, VAR FLOAT res) : EXTERNAL PROC fp.sqrt (VALUE FLOAT a, VAR FLOAT ros) :

ហ

δ

տ

1

Square root of 'a' in 'res'. Returns an occam error if 'a' is negative.

The run time system

As you might hope when an occam program is executed it will follow the program execution until one of three things happen.

- 1] The program terminates
- 2] CTRL-C is pressed on the keyboard
- 3] An error is detected.

In the case of (2) and (3) a debug option will be displayed, this allows you to abort the program, ignore the interrupt (continue), and to restart the program again. Other options control the '-e' trace output, provide a 'system' debug option (which is only really useful to someone who knows their way around the compiler), an option to specify which source file you want to debug and the 'screen animated debug'. This later option should be of most use and is described in detail in the next section.

Errors come in two types 'Fatal Errors' and just 'Errors', it is not possible (or wise) to continue execution after the former, but the latter may be ignored if the symptom is expected.

The run time display debugger

The utility requires the use of a cursor addressable terminal. The system provides selective display of the source file(s) that were compiled to form the program together with a column showing the currently existing processes on those particular lines of the source file.

When initially entered by pressing 'CTRL-C' the program execution will be halted, the execution can be restarted in 'stepped mode' so that the display will be updated every occam scheduler action.

Breakpoints and trace points can be added at selected line numbers. Break points cause the debug display to be automatically entered when any of the process executes any of the source lines on which a break point is set. Trace points cause temporary entry into the debug display before resuming normal execution after five seconds pause.

If a file has been compiled with the '-G' flag then the value of occam variables and the status of channels can be printed. Because an occam program can have several processes running with different values to the same identifiers (e.g. within PAR n = [0 FOR 7], 'n' has a different value for each separate process) a single process must be selected as before this facility can be used. When selected a second window within the debug display is opened and the values printed by the program are placed within it. Straightforward use of the debug display will normally entail running a program and pressing CTRL-C when a dubious section of code is about to be executed and entering the debug display ('z' command). Thereafter the commands 'p' to find the next process, 'f' and 'b' might be used to see whereabouts the process is executing. The program can then be single stepped through using the 'r' command to start execution and 's' command to stop execution. Eventually exit of the debug displayer can be made with the 'x' command.

There are two special markers that are used, '>' on a line indicates the currently selected line and '-' the currently selected process.

The commands where practical have been made similar to those in UNIX 'vi'. (UNIX is a trademark of A.T. & T.).

Available commands

Moving about within the file

1D- Hove forward half a page of source text.

- TP- Move forward a page of source text.
- 1U- Move backward half a page of source text.
- 18- Move backward a page of source text.
- : (number) Move to given line (number) in file.
- k (or fK) Move down one line.

j - (or fJ) Move up one line.

- /(string) Find given (string) in file from current position.
- n Find next string occurrence for match string selected by '/' command.

F

σ

δ

σ

1

p - Find the next process in the file.

Trace/Breakpoints

- b Add breakpoint at currently selected line.
- t Add tracepoint at currently selected line.
- d Delete the trace/break point at the selected line.
- c Delete all the points in the current file.
- C Delete all the points in all the files.
- P Print process status of the currently selected process
- D Deselect the current debug occam process.
- S Select the current debug occam process.
- N Select next process on the same line, if there are several processes that are shown as executing on the same line then 'S' will make an arbitrary choice, 'N' can be used to override this and step through the processes until the one that is desired is selected.

This utility that runs under the run time system enables users to look at the status of the processes during execution of a program.

Symbol inspection

- m Select a symbol to display, if no symbols have been selected before then the symbol window is opened and the value of the variable or the status of a channel.
- M Repeat the previous 'm' command. To find the value of the same variable name again.

Execution control

- a Abort the run.
- r Run debug display if a debug process is selected the debug display will be re-entered every time that process is run, otherwise the debug display will be run each time any process is run.
- > Execute in single step mode. Only a single step is executed.
- s Stop the debug display from running temporarily after a 'r' or 'x'command.
- u Change display step interval (initial step interval is 1), this permits the location of processes to be seen after 'n' steps rather than after each and every time it is executed. Not particularly useful.
- x Exit display debugger, program will proceed normally until a trace/break point is found or 'tC' is pressed.
- X = Exit to main '' menu so that program restart, abort, file selection or system debug can be done. Used when you wish to debug a different file or to set things going again after setting up breakpoints.

Miscellanoous

- ? Print out this help information.
- +L- (or +R) Redraw the current displayed information.
- 1 Buffer keyboard channel input text for the program.
- O Print overall data about the processes currently executing how many are in each process status, stack use and clock time.
- V Display the occam program's current screen output temporarily
- v Invoke the 'view' command on the occam source file (this is just like 'vi' but with read only access to the file - This can be used to provide more powerful string search facilities when debugging.

Display key

The column between the line number and the text is used to display the number and status of processes executing on that line. Because of the compilation these may be out by a line or two in some circumstances. Most sequential code will be executed as a single block - so a process will not move through a SEQ block one step at a time necessarily.

The special symbol 'P' does not represent a process, it indicates that a procedure has been called at that point. 'P' therefore represents the 'call point' of the procedure.

The following symbols are used to represent the various process statii :-

- * An active process may be chosen for execution at any time.
- a Process waiting for one or more ALT guards to become TRUE.
- w Process waiting for a clock time or for input/output.
- c Process is waiting for one or more child PAR processes to terminate.

In addition break and trace points are indicated in the column by giving a 'T' for a trace point and 'B' for a break point.

So a display of :-

316:3*w : occam.s ? razor

indicates that there are three active processes and one process waiting input on line 316.

Keyboard and Screen input/output

Because the debug display routine is fully interactive the screen and keyboard data from the program can not be handled in the same manner as normal. Input for the keyboard must be input using the 'i' command - a whole line can be input and will be buffered up for program input in this way. Screen output should be displayed as it is produced (but a copy of it will be sent to the screen image that will redisplayed on exit from the display debugger) or the 'V' command. Strings can have escapes in them ''n' means newline, ''r' carriage

- 567 -

Non standard occam features

This compiler to the best of my knowledge (Nr.R.P. Stallard of the Department of Computer Studies, Loughborough University of Technology, U.K.) implements the occam language as defined in the occam programming manual published by INMOS limited subject to a few restrictions and extensions that are described in this file. These differences are intended to make transfer of occam programs from different implementations feasible.

It is intended to be compatible to the INMOS booklet version and the Prentice Hall book definition. OCCAN, INMOS and Transputer are registered trademarks of the INMOS Group of Companies.

INHOS proto-occam language revisions

The following additional features introduced into INMOS occam products can now be selected by the compiler flag option '-1'.

STOP primitive.

TIME channel.

IF on finding mone of the conditions TRUE STOPs.

Restrictions

These restrictions are either optional features as described in the published language definition or compiler restrictions unlikely to limit ordinary use of occas.

No configuration section rules.

The operator '>>' uses VAX shift right operator. No prioritized PAR, all parallel processes have equal priority. Number of arguments to a procedure limited to 255 maximum. AFTER returns a time difference not a boolean value.

Extensions

PAR replicator count and base can be variables A variable number of processes can be created by replicated PAR.

Recursive calls to procedures pennitted A procedure can call itself.

Screen channel can be used by more than one process

The special screen channel can be accessed by any number of different occam processes. This facilitates debugging of occam programs and is not difficult to implement. Multiple source file compilation

Procedures and Variables can be defined in one file and referenced in another.

The definition is preceded by the new keyword 'LIBRARY' before 'PROC' and the definition must be at the outer level of program mesting.

References to procedures in other files are defined by preceding 'PROC' by 'EXTERNAL' and replacing the '=' start of procedure definition by ':' to indicate end of definition.

File main.occ	File sub.occ
EXTERNAL PROC f (value n) : SEQ	LIBRARY PROC f (value n) = SEQ
f (27)	num.to.screen (s*102)
	str.to.screen ("Enter next"):

The two files can be compiled by :-

 occam main.occ sub.occ
 to compile both together

 occam sub.occ -1
 to compile sub.occ separately

 occam main.occ sub.oc
 to link in the pre-compiled sub.occ file

 $_{\rm M}$ 3.0 dits has been extended to variables and channels, in the case of vectors of variables and channels the size need not be specified but the type must be :-

Defining file :-

LIBRARY CHAN network.comms [56] : LIBRARY VAR blot [BYTE 4].spot [42] : LIBRARY VAR FLOAT hyper,bolic [2],active [17] :

Referring file :-

EXTERNAL CHAN network.comme [] : EXTERNAL VAR blot [SYTE].spot [],bolic [FLOAT] : EXTERNAL VAR FLOAT hyper,active [] :

Ploating point arithmetic

The compiler permits the use of floating point numbers and arithmetic operators. The compiler uses 32 bit VAX floating point throughout.

Ploating point numbers are declared by following VAR by the new keyword float :-

VAR FLOAT x,y, factor : -- Floating point number declaration VAR num, ply : --- Normal occam variables. F

Floating point number constants are supported these may be in two forms with decimal point or with decimal point and exponent :-

x := 1.45y := 2.3e-23 + 3.4e+1 -- Note that the exponent must be given a sign

The following operators may be used on floating point numbers (both operands must be floating point)

```
+ - • / < > <= >= <> - (monadic minus)

x := 1.3 + (y • factor)

IP

x > 67.8

y := -3.4 --- Note use of monadic minus.
```

Parameters to procedures must also have type set to VAR FLOAT or VALUE FLOAT - the actual parameters must be of the same type.

Ploating values may be transmitted along channels - but there are no checks that the sender and receiver both expect floating point values. Input of floating point numbers can be carried out by calling the library routine 'fp.num.from.keyboard' and output by the routine 'fp.num.to.screen'.

Interconversion of floating point and integers is performed by the assignment operator :-

num := x --- Convert floating 'x' to integer 'num'
y := num --- Convert integer 'num' to floating 'y'

Attempts to use logical and shift operators on floating point numbers are flagged as errors.

III. SOFT-SYSTOLIC SIMULATION PROGRAMS

The logical structure of the simulation programs is explained in section 3.4. Notice that, in some cases, the physical arrangement of the procedures may not coincide with the logical structure, mainly because more than one logical parts are incorporated into the same procedure. However the overall structure remains the same.

In some programs presented, minor non-standard features OCCAM are used, allowed by the Loughborough implementaof tion. However, the conversion to standard OCCAM, or OCCAM-2 is straightforward. The main effort of the simulation is the testing and partial verification of the correctness of the systolic design. Thus, all auxiliary features, such as user interface and host-system interface, are kept as simple as possible, in order not to interfere with the development of the main part of the simulation program, i.e. the systolic design itself. Further, the documentation of the code is rather limited; but it should be easy to follow in conjunction with the corresponding systolic algorithm description in the main part of the thesis.

The external procedures used, can be distinguished in two groups. First, there are some basic routines for: character and string i/o; number conversion; basic unary functions; graphical output. These routines are described in section II of the Appendix. Second, there are some primitive library routines, developed specially for soft-systolic simulation. These procedures are given in A.6, and can be classified as follows:

<u>i/o</u> routines for data streams: they accept/send data streams from/to the user, i.e. they form a basic user interface. Further, there are routines sending/accepting control streams, or single quantities.

<u>sources/sinks</u>: they send/accept data streams to/from the systolic design, i.e. they form a basic host interface. Further, there are sources/sinks that perform 'dummy' operations, or send/receive control streams, as well as data streams.

<u>delays</u>, <u>links</u>: they simulate simple bulding blocks required for the interconnection of cells or arrays, when pipelines are formed.

<u>cells</u>: mainly IPS cells in several configurations, e.g. for linear, orthogonal, hexagonal or engagement (iterative) designs.

optical components: they simulate pixels of light sources, light modulators, static and shift detectors; they may have multiple inputs and/or outputs, that simulate light beams or electric signals.

Notice that the use of library routines is not uniform in all programs: the more recent ones make heavier use of external procedures, while the initial ones, are comparatively 'stand-alone' programs. This fact partially reflects the development of the soft-systolic simulation process through the course of this study.

Finally, of special help are the debugging facilities provided by the Loughborough implementation of OCCAM, as explained in section II of the Appendix. However, it should be noted that, given the complexity of the soft-systolic simulation programs (and the parallel programs in general), two simple rules have been extensively used. First, the gradual construction of the systolic design, from its primitive components, where each intermediate design is tested: thus, the error sources can be easily located.

Second, the formulation of basic systolic arrays as 'building blocks' (procedures) that can be 'plugged' (called) into a program; thus we can create several levels of parallelism, where we always work at the higher level only, as the correctness of the lower levels is given. This concept is illustrated by the 'subsystem' concept in the logical structure of the programs. Notice, however, that this technique tends to produce a large amount of nested procedure calls and concurrent processes. This fact reveals another general problem of simulating systolic algorithms in OCCAM (and, possibly, of parallel programming in OCCAM, in general). On the one hand, it is important for the program clarity to have small and clearly defined procedures. On the other hand, it is important to minimize the procedure calls

and concurrent process creation, so that to avoid excessive run-time overheads. Thus, it may be preferable to simulate in separate programs the several levels of parallelism (or pipelining) that can be seen in a systolic system.

Examples of the latter technique can be found in A.2, A.3 and A.4. In A.2.2 and A.2.3 the block (2x2) partitioning of linear system solution is simulated; notice that the simulation suggests totally parallel communication and computation, for all the components of (2x2)а submatrix. Alternatively, partial serialisation of communication and computation could be implemented, as shown in section 5.2. Finally, total serialisation of the communication can be also attempted, although it would lead to delays in the computation. Observe that the mapping of channels and processes is not expressed by a uniform relation, mainly because there are no two-dimensional arrays in OCCAM.

The same problem is addressed in A.3 and A.4, with the systolic pipeline designs. In these cases, two mapping functions are used: one for the pipleline as a whole, and one for each of the arrays, that comprise a pipeline block. Special links are used to 'translate' the pipeline mapping into the block (array) mapping.
-- A.1.1 -- Systolic System implementing the Bernoulli's Method for the -- calculation of the dominant zero of a polynomial.A Systolic -- Ring calculates the coefficients defined by Newton's Theorem : -- Given $f(x) = x^{**n} + a[1]^*x^{**}(n-1) + ... + a[n-1]^*x + a[n]$ -- if S(p) = r(1)**p + r(2)**p + ... + r(n)**p, where -- r[1], r[2], .., r[n] are the roots of the polynomial, which -- are all real and |r[1]| > |r[2]| > ... > |r[n]. Then -- S[k] = - a[1]*S[k-1] - a[2]*S[k-2] - .. - a[k-1]*S[1] - a[k]*k-- for k = 1, 2, ..., n, and -- S[n+j] = - a[1]*S[n+j-1] - a[2]*S[n+j-2] - .. - a[n]*S[j]-- for i = 1, 2, ... --- Initially the quantities a[1]*1, a[2]*2, .., a[n]*n enter the -- ring together with a[0], a[1], a[2], ..., a[n], where a[0]=0 works -- as a flag controlling the operation of the Ring. The Multiplexer -- restores the normal ring operation and the Divider calculates -- the sequence : -- S[2]/S[1], S[3]/S[2], .., S[i]/S[i-1], for i=1, 2, .., that -- converges to r[1]. --external proc fp.float(value i, var float f): external proc get(var v, value s[]): external proc fp.get.n(var float v[], value n,s[]); external proc fp.put(value float v, value s()): -- Max degree of the polynomial; max size of the ring. def no = 10. so = 6 : -- Coefficients of the polynomial var float a(no) : -- Degree of the polynomial; size of the ring; input-output time. var n, size, intime, outime : -- Channels chan a.c[(2*so)+2], x.c[so+2], s.c[so], r.c[so+2] : -- Inner Product Step Cell (Subtraction): accumulates the inner -- product coefficients in negative form.It inputs a, x; if f=l, i.e -- for a=a(0)=0 it outputs s, saves x as new s, and propagates a, 0; -- otherwise it outputs 0 and propagates a, x. proc ips (chan ain, xin, aout, xout, sout)var float a[2], x[2], s[2] ; var f[2] : seq -- initialisation par i=[0 for 2] par a[i] := 0.0f[i] := 0x[i] := 0.0 s(i) := 0.0 -- main operation while true seg -- i/o par ain 7 a[0]; f[0] xin 7 x[0] aout 1 a[1]; f[1] xout 1 x[1]sout 1 s(1); f(1) -- calculation par if

f[0] = 1par p = a [0]a =: [1]s s[0] := x[0] x[1] := 0.0true par s[1] := 0.0x(1) := x[0] - (a[0] + a[0])a[1] := a[0]f(1) := f(0) :Delay Cell : propagates its input with one cycle delay. proc delay (chan ain, aout)= var float a[2] : var f[2] : seq -- initialisation par i=[0 for 2] par a[i] := 0.0 f[i] := 0-- main operation while true seq -- 1/0 par ain ? a[0]; f[0]aout 1 a[1]; f[1] -- calculation a[1] := a[0]f[1] := f[0] :-- Register for the systolic collection and output of the results.A -- valid result is signalled by the flag f; if no valid result is -- collected the value of the preceding register is propagated. proc recl (chan celin, regin, regout)= var float cel, reg[2] : var f : seq -- initialisation par cel := 0.0 par i=[0 for 2] reg[i] := 0.0£ := 0 while true seq par celin 7 cel; f regin ? reg[0] regout | reg[1] if f = 1 reg[1] := celtrue reg[1] := reg[0] : -- Divider for two successive valid results.It works after an initial -- delay of "output time" and receives results in groups of "size" and -- with intervals of "size".

1

G

~1

ھ

t

proc divide (chan rin, rout,

```
value outime, size)+
 var float r[2], d :
 pea
   -- initialisation
   Dar
     par i=[0 for 2]
       r[i] := 0.0
     d := 0.0
   -- initial delay
   seg i=[0 for outime]
     par
        rin ? anv
        rout I d
   while true
      seq
        -- accept results
        seq i=[0 for size]
          sea
            r[1] := r[0]
            par
              rin 7 r[0]
              rout I d
            if
              r[1] = 0.0
                d := r[0] / 0.000001
              true
                d := r[0] / r[1]
        -- wait for results
        seg i=[0 for size]
          par
            rin ? any
            rout I d:
---
-- Source of the ring : pumps in the coefficients of the polynomial
-- as a[0]=0, a[1], ..., a[n] and the initial values for s[1], ..., s[n],
-- as 0, a[1]*1, a[2]*2, ..., a[n]*n; it also pumps in zeros for the
-- registers collecting the results.
proc source (chan aout, xout, rout,
             value float afl.
             value n)=
  var i i
  seq
   í 1= 0
    while true
      seq
        par
          if
            i = 0
              par
                aout 1 0.0: 1
                xout 1 0.0
            i <= n
              Dar
                aout 1 a[i-1]: 0
                var float temp :
                sea
                  fp.float(i, temp)
                  temp := (-(a[i-1] * temp))
                  xout ! temp
            true
              par
                aout 1 0.0; 0
                xout 1 0.0
          rout 1 0.0
        i := i + 1 :
```

.

```
-- Sink of the ring : it accepts the output from the divider.
proc sink (chan din,
           value outime, size)-
  var float d :
  sea
    seq i=[0 for (outime+1)]
      din 7 any
    while true
      seq
        seq i=[0 for size]
          seq
            din 7 d
            fp.put(d, " root ")
        seq 1=[0 for size]
          din 7 any :
-- Multiplexer of the ring : for the "input time" cycles it accepts
-- input from the source; then the ring is closed. No delay is caused.
proc mux (chan asin, arin, xsin, xrin, aout, xout,
          value intime)=
  var float a, atemp, x, xtemp :
  var f. ftemp, i :
  seq
    i := 0
    while true
      seq
        Īf
          i < intime
            par
              asin 7 a; f
              arin ? atemp; ftemp
              xsin ? x
              xrin ? xtemp
          true
            par
              asin 7 atemp; ftemp
              arin ? a; f
              xsin 7 xtemp
              xrin 7 x
        par
          aout l a; f
          xout 1 x
        i := i + 1 :
-- Ring configuration : (n+1)/2 ips cells and collecting and propagating
-- registers is required, for n=odd, and (n+2)/2 for n=even;
-- One delay between each cell for a. For n=even a last dummy coefficient
-- is required, thus the input time is (n+2). The output time is the sum
-- of the delay for the release of the first valid result plus its
-- propagation time through the registers.
proc system =
  seg
    Ϊf
      (n \setminus 2) = 0
        seq
          intime := n + 2
      true
        intime := n + 1
    size := (intime / 2)
    outime := (intime + 2 ) + size
    par
      source(a.c(0), x.c(0), r.c(0), a, n)
```

1

CFI CFI

~

U

```
mux(a.c[0], a.c[(2*size)+1], x.c[0], x.c[size+1], a.c[1], x.c[1],
          intime)
      par i=[0 for size]
        par
          ips(a.c[(2*i)+2], x.c[i+1], a.c[(2*i)+3], x.c[i+2], s.c[i])
                                                                                    ---
          delay(a.c((2*i)+1), a.c((2*i)+2))
          recl(s.c[i], r.c[i], r.c[i+1])
                                                                                    --
      divide(r.c[size], r.c[size+1], outime, size)
      sink(r.c[size+1], outime, size) :
                                                                                    -
proc getdata =
  seq
    get(n, " degree of polynomial ")
                                                                                   -
    fp.get.n(a, n, " of coefficients ") :
----
seq
  getdata
 system
```

```
-- A.1.2
-- Systolic array for the Graeffe (Root Squaring) Method.
-- It performs one iteration of the algorithm, i.e for a given polynomial
     f(x) = a[0]*x**n + a[1]*x**(n-1) + ... + a[n]
-- it produces the polynomial
     \hat{g}(x) = b[0]*y**n + b[1]*y**(n-1) + ... + b[n]
-- which has as roots the squares of the roots of f(x).
external proc get(var v, value s[]):
external proc fp.get.n(var float v[], value n,s[]):
external proc fp.put.n(value float v(], value n,s(]):
-- Max degree of polynomials and max number of ips cells.
def no = 10, so = 5 :
-- Vectors of coefficients for polynomials f(x), g(x).
var float a[no+1], b[no+1] :
-- Actual degree of polynomials and actual number of ips cells.
var n, s,
-- Overall operation time, and initial delay.
   time, del :
-- Channels for fast-moving a's, slow-moving a's and b's.
chan af.c[so+2], as.c[(3*so)+1], b.c[(2*so)+1] :
-- The first cell of the array: produces the squared coefficient and defines
-- the sign of it and of a-slow, which is also multiplied by two.
proc square (chan ain, afout, asout, bout,
             value time)=
 var float a[4], b :
  var neg :
  seq
   -- initialisation
   par
     par i=[0 \text{ for } 4]
       a[i] := 0.0
     b := 0.0
     neg := false
    -- main operation
   seq i=[0 for time]
     seq
        -- i/o
        par
         ain 7 a[0]
          afout [ a[1]
          asout 1 a[2]
          bout 1 b
        -- calculation
        if
          neg
            -- negate a-slow on alternate cycles.
            par
              a[3] := -a[0]
              neg := false
          true
           par
              a[3] := a[0]
              neg := true
        par
          a[1] := a[0]
          a[2] := (2.0 + a[3])
          b := (a[0] + a[3])
```

1

σι

-1

δ

-- Inner Product Step Cell: accumulates the inner product coefficients in b -- and propagates a-fast, a-slow and b.

proc ips (chan afin, asin, bin, afout, asout, bout, value time)= var float a[4], b[2] : seq -- initialisation par par i=[0 for 4] a[i] := 0.0par i=[0 for 2] b[i] := 0.0-- main operation seq i=[0 for time] seq -- 1/0 par afin 7 a[0] asin ? a[1] bin 7 b[0] afout 1 a[2] asout 1 a[3] bout 1 b[1] -- calculation Dar a[2] := a[0]a[3] := a[1]b[1] := b[0] + (a[0] + a[1]) :-- Delay Cell : propagates its input with one cycle delay. proc delay (chan xin, xout, value time)= var float x[2] : seq -- initialisation par i=[0 for 2] x[i] := 0.0-- main operation seq i=[0 for time] seq -- i/o par xin 7 x[0] xout I x[1] -- calculation x[1] := x[0] :-- Source of the array : pumps the coefficients of the original polynomial proc source (chan aout, value float a[], value n,time)seg i=[0 for time] íf. i <= n aout ! a[1] true aout 1 0.0 : -- Sink of the array : collects the coefficients of the new polynomial after -- an initial delay. proc sink (chan afin, asin, bin, var float b[]. value delay, time)= seg i=[0 for time]

par afin ? anv asin 7 any if i < delav bin 7 any true bin ? b[i-delay] : ----- Array configuration : only [n/2] ips cells are required. -- Two delays between each cell for a-slow and one delay for b. -- The initial delay is 2*[n/2]; the overall operation time is n+2*[n/2]. proc system = seq if $(n \setminus 2) = 0$ s := (n / 2) true s := ((n-1)/2)del := (2 * s) + 1 time := (n + del) + 1par source(af.c[0], a, n, time) square(af.c(0), af.c(1), as.c(0), b.c(0), time) par i=[0 for s] par ips(af.c[i+1], as.c[(i*3)+2], b.c[(i*2)+1], af.c[i+2], as.c[(i*3)+3], b.c[(i*2)+2], time) par j=[0 for 2]delay(as.c[(i*3)+j], as.c[(i*3)+(j+1)], time) delay(b.c[i*2], b.c[(i*2)+1], time) sink(af.c[s+1], as.c[3*s], b.c[2*s], b, del, time) : ----proc getdata = seq get(n, " degree of polynomial ") fp.get.n(a, (n+1), * of coefficients *) : proc putdata seq fp.put.n(b, (n+1), " of coefficients ") : seq

1

СЛ

~1

~1

getdata system

putdata

----- A.1.3 ------ Systolic ring for the calculation of the Eigenvalues of a -- Symmetric Tridiagonal matrix with diagonal coefficients a[1], a[2], ..., a(n)and off-diagonal coefficients --b[1], b[2], ..., b[n-1]--- The n eigenvalues are calculated in parallel using the -- properties of the Sturm sequence of polynomials and the -- method of Bisection. external proc num.to.screen(value n): external proc fp.num.to.screen(value float n): external proc str.to.screen(value s[]): external proc get(var v, value s[]): external proc fp.get(var float v, value s[]): external proc fp.get.n(var float v[], value n,s[]): -- Max size of matrix. def no = 10 : -- Vectors of matrix diagonal and off-diagonal coefficients. -- Vector b starts with a zero. -- Bisection interval of the form : point = (a+b)/2, -- distance = (b-a)/4. var float a(no), b(no), x, d : -- Actual size of matrix. var n : -- Channels for eigenvalues, bisection distance, sturm polunomial -- value, number of sign changes; i/o channels. chan x.c[no+4], d.c[no+4], g.c[(3*no)+1], s.c[no+2], in.c[2], out.c : -- Bisection Check cell: it calculates the new bisection interval -- for each eigenvalue based on the nubmer of sign changes detected. -- A tag associated with x resets the counter that keeps account -- of the order of the eigenvalues. ----proc check(chan xin,din,sin,xout,dout)= var float x[2], d[2]: var sign, tag[2], count : seq -- initialisation par par i=[0 for 2] par x[i] := 0.0tag[1] := 2 d[i] := 0.0 sign := 0 count := 0 -- main operation while true seq -- i/o par xin ? x[0] ; tag[0]din ? d[0] sin ? sign xout i x(1) ; tag(1)dout | d[1]-- calculation if sign > count -- more than required eigenvalues in the interval x[1] := x[0] - d[0]

true x[1] := x[0] + d[0]Dar tag[1] := tag[0]d[1] := d[0] / 2.0if $tag[0] \leftrightarrow 0$ -- last eigenvalue or reset count := 0 true count := count + 1: ----- Sturm Sequence Polynomial calculation cell: for given x, g[i-1] -- it clculates q[i] = (a[i]-x)-(b[i]**2/q[i-1]). The values of -- a[i] and $b[i]^{*+2}$ are preloaded into the cell. If q[i-1] = 0 then -- it is replaced with a small quantity. proc sturm (chan xin, gin, xout, gout, value float a,b)= var float x[2], q[2]: var tag[2] : seq -- initialisation par i=[0 for 2] par x[i] := 0.0tag[1] := 2 q[i] = 1.0 -- main operation while true seq -- 1/0 par xin ? x[0] ; tag[0] qin 7 q[0] xout 1 x[1] ; tag[1] gout 1 g[1] -- calculation Dar x[1] := x[0]tag[1] := tag[0]seq íf. q[0] = 0.0-- q[i-1] is replaced by e q[0] := 0.00001q[1] := (a - x[0]) - (b / q[0]) :-- Sturm Sequence Sign calculation cell : for given g[i], it -- checks if q[i] is negative. If yes, the number of sign changes -- is incremented by one. proc sign(chan gin, sin, sout)= var float q : var s[2] ; seq --initialisation par q := 0.0par 1=[0 for 2] 6[i] := 0 -- main operation while true seq -- i/o par

- 578 -

```
qin 7 q
          sin 7 s[0]
          sout [ s[1]
        -- calculation
        if
          g < 0.0
            s[1] := (s[0] + 1)
          true
            (0]_{a} =: (1]_{a}
-- Delay Cell for d: propagates d with one cycle delay.
--
proc delay (chan xin, xout)=
  var float x[2] :
  sea
    initialisation
    par i=[0 for 2]
      x[i] := 0.0
    -- main operation
    while true
      seq
        -- i/o
        par
          xin 7 x[0]
          xout 1 x[1]
        -- calculation
        x[1] := x[0] :
-- Delay Cell for x: propagates x and tag.
_
proc delay.x (chan xin, xout)=
  var float x(2) :
  var tag[2]:
  seq
    -- initialisation
    par i=[0 for 2]
      par
        x[i] := 0.0
        tag(1) := 2
    -- main operation
    while true
      seq
        -- i/o
        par
          xin 7 x(0) ; tag[0]
          xout 1 x[1] ; tag[1]
         -- calculation
         par
          x[1] := x[0]
           tag[1] := tag[0] :
-- Branching : produces two channels from one without delay.
proc branch(chan xin, xlout, x2out)=
  var float x :
  seq
     -- initialisation
    x := 0.0
    -- main operation
    while true
       seg
         -- in
         xin 7 x
         --out
         xlout 1 x
         x2out 1 x :
```

```
-- I/O Controller : accepts n pairs of the initial interval and
-- then allows for normal ring operation to take place.
___
proc control(chan xlin,dlin,x2in,d2in,xlout,x2out,d2out)=
  var float x, xtemp, d :
  var tag, tagtemp, start :
  seq
    -- initialisation
    x := 0.0
    xtemp := 0.0
    d := 0.0
    tag := 2
    tagtemp := 1
    start := false
    -- main operation
    while true
      seq
        -- input
        if
          not start
            Dar
              xlin ? x ; tag
              dlin ? d
              x2in 7 xtemp; tagtemp
              d2in 7 any
          true
            par
              xlin ? xtemp; tagtemp
              dlin ? any
              x2in 7 x ; tag
              d2in ? d
        -- output
        par
          if
            not start
              xlout 1 xtemp ; tagtemp
            true
              xlout 1 x ; tag
          x2out 1 x ; tag
          d2out 1 d
        -- control
        if
          tag = 1
            start := true :
-- Source of the ring : pumps in pairs of the original bisection
-- interval in the form
-- x = bisection point, d = bisection distance. The nth x has
-- tag bit equal to 1.
___
proc source (chan xout, dout,
             value float x, d,
             value n)=
  var i :
  seq
   -- initialisation
   i := 0
   -- main operation
   while true
      seq
        par
          if
            i = (n - 1)
              xout 1 x ; 1
            true
```

- 579 -

```
xout 1 x / 0
         dout 1 d
        i := (i + 1) :
-- Produce q[0] = 1 and s[0] = 0 all the time.
proc feed(chan gout, sout)=
 while true
    par
     gout 1 1.0
      sout 1 0 :
-
-- Sink of the ring : collects the new approximations for the
-- eigenvalues that are produced for each new bisection in an
-- array. The initial dummy values are discarded.
---
proc sink (chan xin)=
  var float x :
  var tag :
  seq
    Initialisation
    par
      x := 0.0
      tag := 2
    -- main operation
    while true
      seg
        xin 7 x ; tag
        if
          tag <> 2
             -- no dummy value
             seq
              fp.num.to.screen(x)
              str.to.screen(" ")
         i£
           tag = 1
             -- last eigenvalue
             str.to.screen{ *c *n *) :
 -- Receive q[n] all the time.
 proc qn(chan qin)=
  while true
     gin ? any :
 -
 -- Ring configuration
 ---
 proc system =
   par
     -- array for Sturm Sequence computation
     par i=[0 for n]
       par
         delay(d.c[i],d.c[i+1])
         sturm(x.c[i],q.c[3*i],x.c[i+1],q.c[(3*i)+1],a[i],b[i])
         branch(q.c[(3*i)+1],q.c[(3*i)+2],q.c[(3*i)+3])
         sign(q.c((3*i)+2),s.c(i),s.c[i+1))
     -- bisection test and i/o controller
     delay(d.c[n],d.c[n+1])
     delay.x(x.c[n],x.c[n+1])
     check(x.c[n+1],d.c[n+1],s.c[n],x.c[n+2],d.c[n+2])
     control(in.c[0], in.c[1], x.c[n+2], d.c[n+2],
             out.c,x.c[0],d.c[0])
     -- i/o of the ring
     source(in.c[0],in.c[1],x,d,n)
     sink(out.c)
     feed(q.c[0],s.c[0])
```

```
seg
getdata
system
```

```
-- A.1.4
-- Systolic array for the evaluation of a polynomial and its
-- derivatives using the nested multiplication (Horner) scheme.
-- For a polynomial p(x) of degree d, a triangular array of
--- area = ((d+1)**2)/2 IPS , produces for a given point x, the
--- quantities p(x)/01, p'(x)/11, p''(x)/21, p'''(x)/31, .... in
-- time = (d+2) IPS cycles.
external proc get ( var v, value s[] ) :
external proc fp.get ( var float v, value s[] ) :
external proc fp.get.n ( var float v[), value n, s[] ) :
external proc fp.put.n ( value float v[], value n, s[] ) :
external proc fp.so.x ( chan xout, value float x(), value time ) :
external proc fp.so.d ( chan xout, value time ) :
external proc fp.si.x ( chan xin, var float x[], value time ) :
external proc fp.si.d ( chan xin, value time ) :
external proc fp.lk ( chan xin, xout, value time ) :
def no = 10 :
-- Basic cell : Inner Product with modified output
-- b is propagated in two directions, x in one.
proc hip ( chan ain, bin, xin, aout, bout, xout,
           value time ) -
  var float a, b(2), x(2) :
  seq
    -- initialise
    par
      a = 0.0
      par i=[0 for 2]
        par
          b[i] := 0.0
          x[i] := 0.0
    -- main operation
    seq i=[0 for time]
      seq
        -- 1/0
        par
          ain 7 a
          bin 7 b[0]
          xin ? x[0]
          aout | b[1]
          bout | b[1]
          xout [ x[1]
        -- calculation
        par
          b[1] := a + (b[0] + x[0])
          x[1] := x[0] :
-- Pipeline configuration
-- Row i of the array has m = (d+1)-i hip cells, each one
-- accepting a coefficient in skewed fashion; b, x are
-- pipelined. The partial results (b), are also produced as
 -- the coefficints for row i+1 of the array.
proc pipe ( chan co.io[], x.in, b.out,
            value m, base, time ) =
   chan b.c[no], x.c[no] :
  par
    par i=[0 for m]
      hip ( co.io[base+i], b.c[i], x.c[i],
             co.io{(base+m)+(i+1)], b.c[i+1], x.c[i+1], time )
    fp.so.d ( b.c[0], time )
    fp.lk ( x.in, x.c[0], time )
```

```
fp.lk ( b.c[m], b.out, time )
   fp.si.d ( x.c(m), time ) :
-- System configuration
\sim A triangular array of rows of size d+1, d, d-1, d-2, ..., 1
-- Input : the coefficients of p(x) and x; output in b.
-- A dummy channel in the first row for uniformity.
proc syst ( chan a.in[], x.in[], b.out[],
            value n, time ) =
  chan co.c(no*no) :
  var base[no] :
 seq
   base[0] := 0
    seq i=[0 for n]
      base[i+1] := base[i] + ((n + 1) - i)
    par
      par i=[0 for n]
        par
          pipc ( co.c, x.in[i], b.out[i], (n-i), base[i], time )
          fp.si.d ( co.c[base(i+1]-1], time )
      par i=[0 \text{ for } n]
        fp.lk ( a.in[i], co.c[base[0]+i], time )
      fp.so.d ( co.c[base[1]-1], time )
      fp.si.d { co.c[base[n]], time ) :
-- Skewed source for a, x.
proc so.ax ( chan axout,
             value float ax,
             value t, time ) =
  var float temp[1] :
  seq
    temp[0] := ax
    fp.so.d ( axout, t)
    fp.so.x ( axout, temp, 1)
    fp.so.d ( axout, (time-(t+1))) :
-- Delayed sink for b.
proc si.b ( chan bin.
            var float b.
            value time ) -
  var float temp[1] :
  sea
    fp.si.d ( bin, (time-1) )
    fp.si.x ( bin, temp, 1 )
    b := temp[0] :
-- System interface-driver
proc driv ( value float a[], x,
            value n.
            var float b[] ) =
  chan a.c[no], x.c[no], b.c[no] :
  var time :
  seq
    time := n + 1
    par
      par i={0 for n}
        par
          so.ax { a.c[i], a[i], i, time }
          60.ax ( x.c[i], x, i, time )
          si.b ( b.c[i], b[i], time )
      syst ( a.c, x.c, b.c, n, time ) :
```

581 .

-- Main var float a[no], x, b[no] : var d : seq get (d, " degree of polynomial ") fp.get.n (a, (d+1), " polynomial coeffs ") fp.get (x, " evaluation point ") driv (a, x, (d+1), b) fp.put.n (b, (d+1), " results ")

```
-- A.1.5
 -
 -- Systolic Pipelines for the Bairstow Method.
 -- For a polynomial f(x) of degree d, and a quadratic
 -x^{**2} = p^{*x} - q = [x - (a + b^{*i})]^{*}[x - (a - b^{*i})], the system
 -- produces the coefficients for the calculation of
 -- f(a + b*i) and f'(a + b*i).
 -- Area = 2*d IPS cells, Time = 2*d+6 IPS steps.
 _
  external proc get ( var v, value s[] ) :
 external proc fp.get ( var float v, value s[] ) :
 external proc fp.get.n ( var float v[], value n, s[] ) :
 external proc fp.put.n ( value float v(), value n, s[] ) :
 external proc fp.so.x ( chan xout, value float x(), value time ) :
  external proc fp.so.d ( chan xout, value time ) :
  external proc fp.si.x ( chan xin, var float x(), value time ) :
  external proc fp.si.d ( chan xin, value time ) :
  external proc fp.lk ( chan xin, xout, value time ) :
  def no = 10 :
  -- Basic cell : Modified Inner Product
  -- Cycle t : b = a+q*b[-2]; cycle t+l : b = b+p*b[-1];
  -- cycle t+2 : output b[-1] ; cycle t+3 : output b.
  -- Cycles t+2, t+3 are overlapped; b is propagated in 2 directions.
  proc bip ( chan ain, bin, pin, aout, bout, pout,
             value time ) =
    var float a, b[3], p[3]:
    seq
-
      -- initialise
      par
        a := 0.0
        par i=[0 for 3]
          par
            b[i] := 0.0
             p(1) := 0.0
       -- main operation
       seq i=[0 for time]
         seq
           -- 1/o
           par
             ain 7 a
             bin ? b[0]
             pin 7 p[0]
             aout 1 b[2]
             bout 1 b[2]
             pout 1 p[2]
           -- calculation
           par
             if
               (i \setminus 2) = 0
                 seq
                   b[2] := b[1]
                   b[1] := a + (b[0] + p[0])
               true
                 seq
                   b[1] := b[1] + (b[0] + p[0])
                   b(2) := b(0)
             seq
               p(2) := p(1)
               p(1) := p(0) :
   -- Pipeline configuration
   -- The coefficients enter the pipeline in skewed fashion;
   -- p,q and b travel along the pipeline.
```

L

 СТ 8

N

t

```
___
proc pipe { chan co.io[], p.in, b.out,
            value m, base, time ) =
  chan b.c[no], p.c[no] :
  par
    par i=[0 for m]
      bip ( co.io[base+i], b.c[i], p.c[i],
            co.io((base+m)+(i+1)), b.c(i+1), p.c(i+1), time )
    fo.so.d ( b.c[0], time )
    fp.lk ( p.in, p.c(0), time )
    fp.lk ( b.c[m], b.out, time )
    fp.si.d ( p.c(m), time ) :
---
-- Skewed source for coefficients,0 and p,q.
proc so.ap ( chan apout,
             value float ag, p,
             value t, time ) =
  var float temp[2] :
  seq
    temp[0] := aq
    temp[1] := p
    fp.so.d ( apout, t)
    fp.so.x ( apout, temp, 2)
    fp.so.d ( apout, (time-(t+2))) :
-- Delayed sink : collects for the last two cycles.
_
proc si.b ( chan bin,
            var float b[],
            value i, time } =
  var float temp[2] :
  seq
    fp.si.d ( bin, (time-2) )
    fp.si.x ( bin, temp, 2 )
    seg j=[0 for 2]
      b[(1+2)+j] := temp[j] :
-- System configuration
-- Two pipelines of size (d+1) and d; the coefficients of the
-- polynomials move across them. The first cell of the second
-- pipeline is dummy for synchonization; a dymmy input channel
-- for the first pipeline for uniformity.
-
proc syst ( chan a.in[], p.in[], b.out[],
             value n, time ) =
   chan co.c[3*no] :
  par
    pipc ( co.c, p.in[0], b.out[0], n, 0, time )
    pipc ( co.c, p.in[1], b.out[1], (n-1), (n+1), time )
    fp.so.d ( co.c(n), time )
    fp.si.d ( co.c(n), time )
    par i=[0 for n]
       par
         fp.lk ( a.in[i], co.c[i], time )
         fp.si.d ( co.c[(2*n)+i], time ) :
 -- System interface-driver
 proc driv ( value float a[], p, g,
             value n,
             var float b(] ) =
   chan a.c(no), p.c(2), b.c(2) :
   var time :
   seq
    time := (2 + n) + 2
```

```
par
      par i=[0 for n]
         so.ap ( a.c[i], a[i], 0.0, (2*i), time )
       par i=[0 \text{ for } 2]
         par
           so.ap ( p.c[i], q, p, (2*i), time )
           si.b ( b.c[i], b, i, time )
       syst ( a.c, p.c, b.c, n, time ) :
-----
-- Main
----
var float a(no), p, q, b(4) :
var d :
seq
  qet ( d, " degree of polynomial " )
  fp.get.n ( a, (d+1), " polynomial coeffs " )
  fp.get ( p, "p")
fp.get ( q, " q " )
driv ( a, p, q, (d+1), b )
  fp.put.n ( b, 4, " results " )
```

```
-- A.1.6
-
-- Systolic Array for the calculation of the characteristic
-- polynomial of an Lower Hessenberg matrix.
external proc get ( var v. value s[] ) :
external proc fp.get ( var float v, value s[] ) :
external proc fp.get.n ( var float v[], value n, s[] ) :
external proc fp.put.n ( value float v[], value n, s[] ) :
external proc fp.br ( chan xin, xout1, xout2, value time ) :
external proc fp.so.d ( chan xout, value time ) :
external proc fp.so,x ( chan xout, value float x(), value time ) :
external proc fp.si.d ( chan xin, value time ) :
external proc fp.si.x ( chan xin, var float x(), value time ) :
def no = 10, to = ((2*no)-1) :
-
-- Bi-Directional Inner Product Step cell
___
proc bdips ( chan hin, hout, yin, yout, uin, uout,
             value time ) =
  var float h[2], y[2], u[2] :
  seq
    -- initialise
    par i=[0 for 2]
      par
        h[i] := 0.0
        y[1] := 0.0
        u[i] := 0.0
    -- main operation
    seq i=[0 for time]
      seq
        -- i/o
        par
         • hin 7 h[0]
          yin ? y[0]
          uin 7 u(0)
          hout 1 h[1]
          yout 1 y[1]
          uout | u[1]
        -- calculation
        par
          y[1] := y[0] + (h[0] * u[0])
          h(1) := h(0)
          u[1] := u[0] :
-- Boundary cell
---
proc bound ( chan hin, hout, yin, uout,
             value float z,
             value time ) =
  var float h[2], y, u :
  seq
    -- initialise
    par
      par i=[0 for 2]
        h[i] := 0.0
      u := 1.0
      y := 0.0
     -- main operation
    seg i=[0 for time]
      seq
        -- i/o
        par
          hin 7 h[0]
          yin 7 y
```

~ ----

```
hout 1 h[1]
          uout 1 u
        -- calculation
        if
          (1 \setminus 2) = 1
            u := (v - (z + u)) / (-h(0))
        h(1) := h(0) :
-- Linear array configuration
___
proc bdarr ( chan hio[ ), uout,
             value float z.
             value n, time ) =
  chan y.c[no], u.c[no+1] :
 par
   par i=(1 \text{ for } (n-1))
      bdips ( hio[i], hio[n+i], y.c[i], y.c[i-1],
              u.c[i], u.c[i+1], time )
    bound ( hio[0], hio[n], y.c[0], u.c[0], z, time )
    fp.br ( u.c(0), u.c[1], uout, time )
    fp.so.d (y.c[n-1], time)
    fp.si.d (u.c[n], time) :
---
-- System configuration
proc syst ( value float h[), z,
            var float u[],
            value n. time ) -
  chan h.c[2*no], u.c :
 par
   par i=[0 for n]
      par
        var float h.temp[to] :
        seq
          seq j=[0 for time]
            h.temp[j] := h[(i*time)+j]
          fp.so.x ( h.c[i], h.temp, time )
        fp.si.d ( h.c[n+i], time ]
    bdarr ( h.c, u.c, z, n, time )
    fp.si.x (u.c, u, time) :
----
-- Main
--
var float h(no*to), z, u(to) :
var d, time :
seq
  get ( d, " order of lower hessenberg matrix " )
  time := (2 + d) + 1
  fp.get.n ( h, ((d+1)*time), " diagonal seg - upper first " )
  fp.get ( z, " evaluation point " )
  syst ( h, z, u, (d+1), time )
  fp.put.n ( u, time, " result " )
```

```
- 584 -
```

-- A.2.1 --- Preprocessor for the systolic array performing block -- (2x2) R+F LU/LDU decomposition. Two control signals are -- used: c1, to align all elements of a submatrix in the -- same line; c2, to achieve the correct spacing between -- submatrices. Further, there are reformatting delays. -- Time is determined by the maximum delay that may occur. external proc get (var v, value s[]) : external proc get n (var v[), value n, s[]) : external proc fp.get.n (var float v[], value n, s[]) : external proc fp.put.n (value float v[], value n, s[]) : external proc so.x (chan xout, value x[], value time) : external proc fp.so.x (chan xout, value float x[], value time) : external proc si.d (chan xin, value time) : external proc fp.si.d (chan xin, value time) : external proc fp.si.x (chan xin, var float x[], value time) : external proc dl (chan xin, xout, value d, time) : def no = 10, to = (n_0+6) : -- Demultiplexer for the alignement of the submatrix elements -- Reformatting delays are configured, according to switch. --proc ali (chan xin, xout, ain, left, right, value switch, time) = var float a, l, r, reg[4] : var x[2] : seq --- initialisation par a := 0.0 1 := 0.0 r := 0.0par i=[0 for 4] reg[i] := 0.0 par i=[0 for 2] x[i] := 0-- main operation seg i={0 for time} seq -- input/output par xin 7 x[0] xout ! x[1] ain 7 a left | 1 right | r -- calculation x[1] := x[0]if x[0] = 1par reg[0] := 0 req[1] := a true par reg[0] := a reg[1] := 0 if switch = 0sea $\tilde{1} := req[2]$ r := reg[3]reg[2] := reg[0]

reg[3] := reg[1]true seq 1 := reg[0] r := reg[3] reg[3] := reg[2]reg[2] := reg[1] ; -- Delay for correct spacing of submatrices. -- Reformatting delays are configured, according to switch. proc spa { chan ain{], aout[], xin, xout, value basein, baseout, switch, time) var float a0[4], a1[4], a2[4], a3[4], a4[4] : var x[2], overwrite : seq -- initialisation par par i=[0 for 4] par a0[1] := 0.0al[i] := 0.0a2(1) := 0.0a3[i] := 0.0a4(i) := 0.0par i=[0 for 2] x[i] := 0overwrite := false -- main operation seg i=[0 for time] seq -- input/output par par i=[0 for 4]par ain[basein+i] ? a0[i] aout[baseout+i] 1 a4[i] xin 7 x[0]xout 1 x[1] -- calculation x[1] := x[0]if x[0] = 1par par i=[0 for 4] par a2[i] := 0.0al[i] := a0[i]overwrite := true true seg Ϊf overwrite par i=[0 for 4] a2[i] := a1[i]true par i=[0 for 4]a2[i] := a0[i]overwrite := false if switch = 0par i=[0 for 4] seq a4[i] := a3[i]a3[i] := a2[i]true

585

Т

par i=[0 for 4]a4[i] := a2[i] :--- System configuration, for block tridiagonal matrix -proc system (var float block[], value float point(), value c1[], c2[], time) =chan ain[7], a.c[14], aout[12], cl.c[8], c2.c[7] : par -- data sources par i=[0 for 7]var float temp.point[to] : seg seq j=[0 for time] temp.point(j) := point((i*time)+j) fp.so.x (ain[i], temp.point, time) -- control sources so.x (cl.c(0), cl, time) so.x (c2.c[0], c2, time) -- preprocessor par i = [0 for 7]ali (cl.c[i], cl.c[i+1], ain[i], a.c[(2*i)+1], a.c[2*i], (1\2), time) par i=[0 for 3]par spa (a.c, aout, c2.c[2*i], c2.c[(2*i)+1], ((4*i)+1), (4*i), (i\2), time) d1 (c2.c[(2*i)+1], c2.c[(2*i)+2], 1, time) -- data sinks par i=[0 for 12]var float temp.block[to] : sea fp.si.x (aout[i], temp.block, time) seq j=[0 for time] block[(i*time)+j] := temp.block[j] fp.si.d (a.c[0], time) fp.si.d (a.c(13), time) -- control sinks si.d (cl.c[7], time) si.d (c2.c[6], time) : ----- Main var float point[7*to], block[12*to] : var cl[to], c2[to], n, time : seq get (n, " size of matrix ") time := n + 6fp.get.n (point, (7*time), " input seq: upper diag first ") getin (c1, time, " c1 ")
getin (c2, time, " c2 ") system (block, point, cl, c2, time) fp.put.n (block, (12*time), " block output ")

-- A.2.2 -- Systolic array for block (2x2) R+F LU/LDU decomposition -- of a (2x2) block tridiagonal matrix, external proc get (var v, value s(]) : external proc fp.get.n (var float v[], value n, s[]) : external proc fp.put.n (value float v[], value n, s[]) : external proc fp.so.x (chan xout, value float x(), value time) : external proc fp.si.x (chan xin, var float x[], value time) : def no = 10, to = (no + 6) : -- Inner Product calculation for (2x2) submatrices : -- A = A + B + C.proc block.ip (value float b[], c[], var float a[]) = par a[0] := ((b[0] * c[0]) + (b[1] * c[2])) + a[0]a[1] := ((b[0] * c[1]) + (b[1] * c[3])) + a[1]a(2) := ((b(2) * c(0)) + (b(3) * c(2))) + a(2)a[3] := ((b[2] + c[1]) + (b[3] + c[3])) + a[3] :--- Calculation of the Determinant of a (2x2) matrix. proc det (value float a[], var float dt) seq dt := (a[0] + a[3]) - (a[1] + a[2]) :-- Inversion of a (2x2) matrix. proc inv (value float a[], dt, var float ai[]) = if dt <> 0.0 par ai[0] := a[3] / dtai[1] := (-a[1]) / dtai[2] := (-a[2]) / dtai[3] := a[0] / dttrue par i={0 for 4} ai(i) := 0.0 : -- Processor sll : accepts a (2x2) matrix A and its -- determinant d, and produces A/d. proc sl1 (chan ain[], dtin, aout[], value time) = var float a0(4), a1(4), dt : seg -- initialisation par par i=[0 for 4]par a0[i] := 0.0 a1[i] := 0.0dt := 0.0 -- main operation seq j=[0 for time] seg . -- i / o par par i=[0 for 4]par ain[i] 7 a0[i] aout[i] 1 al[i]

586

```
dtin ? dt
        -- calculation
        if
          dt <> 0.0
            par i=[0 for 4]
              al[i] := a0[i] / dt
          true
            par i=[0 for 4]
              a1[i] := 0.0:
-- Processor s12 : accepts a (2x2) matrix A and its
-- determinant d, and produces them unchanged. A is
-- delayed for one cycle.
-
proc s12 ( chan ain(], dtin, aout[), dtout1, dtout2,
           value time ) =
  var float a0[4], a1[4], a2[4], dt[2] :
  seq
    -- initialisation
    par
      par i=[0 for 4]
        par
          a0[i] := 0.0
          a1[i] := 0.0
          a2[i] := 0.0
      par i=[0 for 2]
        dt[1] := 0.0
    -- main operation
    seq j=[0 for time]
      seq
        -- 1/0
        par
          par i=[0 for 4]
            par
              ain[i] 7 a0[i]
              aout[1] 1 a2[1]
          dtin 7 dt[0]
          dtout1 | dt[1]
          dtout2 1 dt[1]
        -- calculation
        par i=[0 for 4]
          a2[i] := a1[i]
        par i=[0 for 4]
          a1[i] := a0[i]
        dt[1] := dt[0] :
-- Processor s13 : accepts a (2x2) matrix A and its
-- determinant d, and produces dA.
proc s13 ( chan ain[], dtin, aout[],
            value time ) =
  var float a0[4], a1[4], dt :
  seq
    -- initialisation
    par
      par i=[0 for 4]
        par
          a0[i] := 0.0
          al[i] := 0.0
    dt := 0.0
    -- main operation
    seq j=[0 for time]
      seg
        -- i/o
        par
          par i=[0 \text{ for } 4]
```

.

```
par
              ain[1] ? a0[1]
              aout[i] [ al[i]
          dtin 7 dt
        -- calculation
        par i=\{0 \text{ for } 4\}
          al[i] := a0[i] + dt :
-- Processor s21 : accepts two (2x2) matrices A and U,
-- and produces dL = d(U^{**}-1)A, where d is the determinant
-- of U.
--
proc s21 ( chan ain[], uin[], dlout1[], dlout2[],
           value time ) =
  var float a0[4], a1[4], u[4], d1(4] ;
  seq
    -- initialisation
    par i=[0 for 4]
      par
        a0[1] := 0.0
        al[i] := 0.0
        u[i] := 0.0
        dl[i] := 0.0
    -- main operation
    seg j=[0 for time]
      seq
        -- i / o
        par i=[0 for 4]
          par
            ain[i] 7 a0(i)
            uin[i] 7 u[i]
             dlouti[i] i dl[i]
            dlout2[i] 1 dl[i]
        -- calculation
        inv ( u, 1.0, al )
        par i=[0 \text{ for } 4]
          dl[i] := 0.0
        block.ip ( a1, a0, d1 ) :
-- Processor s22: accepts a (2x2) matrix, and outputs
-- the matrix and its determinant
proc s22 ( chan ain[], aout1[], aout2[], dtout1, dtout2,
            value time ) =
  var float a0[4], a1[4], dt :
  seg
    -- initialisation
    par
      par i=[0 \text{ for } 4]
        par
          a0[i] := 0.0
          al[i] := 0.0
      dt := 0.0
    -- main operation
    seq j=[0 for time]
      seg
        -- i / o
        par
          par i=[0 for 4]
             ain[i] 7 a0[i]
           par i=[0 for 4]
             par
               aoutl[i] [ al[i]
               aout2[i] 1 a1[i]
           dtoutl | dt
           dtout2 I dt
```

587 -

```
-- calculation
        par i=[0 for 4]
          al[i] := a0[i]
        det ( a0, dt ) :
-- Processor s23 : accepts a (2x2) matrix A and its
-- determinant d, and produces A/d.
proc s23 ( chan ain[], dtin, aout1[], aout2[],
           value time ) -
  var float a0[4], a1[4], dt :
  seq
    initialisation
    par
      par i=[0 for 4]
        par
          a0[1] := 0.0
          al(i) := 0.0
      dt := 0.0
    -- main operation
    seq j=[0 for time]
      seq
         -- i / o
         par
          par i=[0 for 4]
            par
               ain[1] 7 a0[1]
               aoutl[i] | al[i]
               aout2[i] ! al[i]
           dtin 7 dt
         -- calculation
         1f
           dt \leftrightarrow 0.0
             par i=[0 for 4]
               a1[i] := a0[i] / dt
           true
             par i=[0 for 4]
               a1[i] := 0.0:
 -- Processor s31 : accepts three (2x2) matrices, A, L, U
 -- and produces A = A + L * U. It is assumed that the
 -- computation requires two cycles.
 -----
 proc s31 ( chan ain[], lin[], uin[], aout[],
            value time ) +
   var float a0(4], a1(4], a2(4), 1(4), u(4) :
   seq
     Initialisation
     par i=[0 for 4]
       par
         a0[i] := 0.0
         a1[i] := 0.0
         a2[i] := 0.0
         1[i] := 0.0
         u[i] := 0.0
     -- main operation
     seq j=[0 for time]
       seq
          -- i / o
          par i=[0 for 4]
            par
              ain[1] 7 a0[1]
              lin(i) ? l(i)
              uin[i] ? u[i]
              aout[i] 1 a2[i]
          -- calculation
```

```
par i=[0 for 4]
          a2(1) := a1(1)
        par i=[0 \text{ for } 4]
          a1[i] := a0[i]
        block.ip ( 1, u, al ) :
-- System configuration
proc system ( value float ain1[], ain2[], ain3[],
              var float aout1[], aout2[], aout3[],
              value time ) =
  chan cllo[4], cl2o[4], cl3o[4], cl211d, cl213d,
       c2111[4], c2212[4], c2212d, c2313[4], c2221[4], c2223d,
       c3122[4], c2131[4], c2331[4], c211[4], c231[4], c311[4] :
  par
    -- sources
    par i=[0 for 4]
      var float temp.a[to] :
      sea
        seq j={0 for time}
          temp.a[j] := ain1[(i*time)+j]
         fp.so.x ( c23i(i), temp.a, time )
     par i=[0 \text{ for } 4]
      var float temp.a[to] :
       seq
         seq j=[0 for time]
           temp.a[j] := ain2[(i*time)+j]
         fp.so.x ( c31i[i], temp.a, time )
     par i=[0 for 4]
       var float temp.a[to] :
       seq
         seg j=[0 for time]
           temp.a[j] := ain3[(i*time)+j]
         fp.so.x ( c2li[i], temp.a, time )
     -- arrav
     s11 ( c2111, c1211d, c11o, time )
     s12 ( c2212, c2212d, c120, c1211d, c1213d, time )
     s13 ( c2313, c1213d, c130, time )
     s21 ( c211, c2221, c2111, c2131, time )
     s22 ( c3122, c2221, c2212, c2212d, c2223d, time )
     s23 ( c231, c2223d, c2313, c2331, time )
     s31 ( c31i, c2131, c2331, c3122, time )
     -- sinks
     par i=[0 for 4]
       var float temp.a[to] :
        seq
          fp.si.x ( c13o[i], temp.a, time )
          seq j=[0 for time]
            aout1[(i*time)+j] := temp.a[j]
      par i=[0 for 4]
        var float temp.a[to] :
        sea
          fp.si.x ( cl2o[i], temp.a, time )
          seq j=[0 for time]
            aout2[(i*time)+j] := temp.a[j]
      par i=[0 for 4]
        var float temp.a(to) :
        seq
          fp.si.x ( cllo[i], temp.a, time )
          seq j=[0 for time]
            aout3[(i*time)+j] := temp.a[j] :
  -- Main
  var float ain1[4*to], ain2[4*to], ain3[4*to],
             aout1(4*to), aout2(4*to), aout3(4*to) :
```

588 -

var n, time :
seq
get (n, " problem size ")
time := n + 6
fp.get.n (ain1, (4*time), " a1 input seq ")
fp.get.n (ain2, (4*time), " a2 input seq ")
fp.get.n (ain3, (4*time), " a3 input seq ")
system (ain1, ain2, ain3, aout1, aout2, aout3, time)
fp.put.n (aout1, (4*time), " a1 output seq ")
fp.put.n (aout2, (4*time), " a2 output seq ")
fp.put.n (aout3, (4*time), " a3 output seq ")

---- A.2.3 -- Systolic array for (2x2) block backsubstitution. The input -- matrix can be produced by block (2x2) R+F LU/LDU decomposition. external proc get (var v, value s(]) : external proc fp.get.n { var float v[], value n, s[]) : external proc fp.put.n (value float v(], value n, s(]) : external proc fp.so.x (chan xout, value float x[], value time) : external proc fp.si.x (chan xin, var float x[], value time) : def no = 10, to = (no + 5) : -- Inner Product calculation for (2x2) submatrix, and -- (2x1) subvetors : y = Ax+y. -proc block.ip (value float a[], x[], var float y[]) = par y(0) := ((a[0] * x[0]) + (a[1] * x[1])) + y[0]y(1) := ((a(2) + x(0)) + (a(3) + x(1))) + y(1) :-- Calculation of the Determinant of a (2x2) matrix. proc det (value float a[], var float dt) = seg dt := (a[0] + a[3]) - (a[1] + a[2]):------ Inversion of a (2x2) matrix. proc inv (value float a[], dt, var float ai[]) = if dt <> 0.0 par ai[0] := a[3] / dtai(1) := (-a[1]) / dt ai[2] := (-a[2]) / dtai[3] := a[0] / dttrue par i=[0 for 4] ai(i) := 0.0 : -- Processor s22: accepts a (2x2) matrix, and outputs -- the matrix and its determinant proc s22 (chan ain[], aout[], dtout, value time) = var float a0[4], a1[4], dt : seq -- initialisation par par i=[0 for 4] par a0(i) := 0.0a1(i) := 0.0dt := 0.0 -- main operation seq j=[0 for time] pea -- i / o par par i=[0 for 4]ain[i] 7 a0[i] par i=(0 for 4)aout[i] | a1[i] dtout 1 dt -- calculation

```
par i=[0 \text{ for } 4]
          al[i] := a0[i]
        det ( a0, dt ) :
-- Processor s23 : accepts a (2x2) matrix and its determinant
-- and it produces its inverse.
proc s23 ( chan ain[], dtin, aout[],
           value time ) =
  var float a0[4], a1[4], dt :
  seg
    Initialisation
    par
      par i=[0 for 4]
         par
           a0[i] := 0.0
           a1(i) := 0.0
       dt := 0.0
     -- main operation
     seq j=[0 for time]
       seg
         -- i / o
         par
           par i=[0 for 4]
             ain(i) ? a0(i)
           dtin ? dt
           par i=[0 for 4]
             aout[i] 1 al[i]
         -- calculation
         inv ( a0, dt, al ) :
 -- Processor sd : calculates a (2x1) solution subvector x
 -- as x = (A^{++-1})(b-y).
 proc sd ( chan ain{], bin[], yin{], xout[},
            value time ) =
   var float a[4], b[2], y[2], x[2], id[4] :
    sea
      1 initialisation
      Dar
        par i=[0 for 4]
          a[1] := 0.0
        par i=[0 \text{ for } 2]
          par
            b[i] := 0.0
            y[i] := 0.0
            x(i) := 0.0
        id[0] := 1.0
        id[1] := 0.0
        id[2] := 0.0
        id[3] := 1.0
      -- main opertaion
      seg j=[0 for time]
         seg
           -- i/o
           par
             par i=[0 for 4]
               ain[1] ? a[1]
             par i=[0 for 2]
               par
                 bin[i] ? b[i]
                 yin[i] ? y[i]
                 xout[i] 1 x[i]
           -- calculation
           par i=[0 for 2]
             par
```

```
y[i] := (-y[i])
           x(i) := 0.0
        block.ip ( id, b, y )
        block.ip \{a, y, x\}:
-- Processor ma : it performs a block (2x2) IPS, in the
-- form y=Ax+0.
___
proc ma ( chan ain[], xin[], xout[], yout[],
          value time ) =
  var float a[4], x0[2], x1[2], y[2] :
  seg
    -- initialisation
    par
      par i=[0 for 4]
        a[i] := 0.0
      par i=[0 for 2]
        par
          x0(1) := 0.0
          x1(i) := 0.0
          y[i] := 0.0
    -- main operation
    seg j=[0 for time]
       seq
        -- i / o
        par
          par i=[0 for 4]
             ain[i] 7 a[i]
           par i=[0 for 2]
             par
               xin[i] 7 x0[i]
               xout[i] 1 x1[i]
               yout[i] | y[i]
         -- calculation
         par i=[0 for 2]
           par
             x1[i] := x0[i]
             v[1] := 0.0
         block.ip(a, x0, y):
 -- System configuration.
 - --
 proc system ( value float al[], a2[], b[],
               var float x[],
               value time ) -
   chan as22[4], as23[4], asd[4], ama[4], dts23,
        bsd[2], ysd[2], xsd[2], xma[2] :
   par
     -- sources
     par i=[0 for 4]
       var float temp.al[to] :
        seq
         seq j=[0 for time]
           temp.al(j) := al((i*time)+j)
          fp.50.x ( as22[i], temp.al, time )
      par i=[0 for 4]
       var float temp.a2[to] :
        seq
          seq j=[0 for time]
            temp.a2[j] := a2[(i*time)+j)
          fp.so.x ( ama[i], temp.a2, time )
      par i=[0 for 2]
        var float temp.b[to] :
        seq
          seq j=[0 for time]
            temp.b[j] i= b[(i*time)+j]
```

- 590 -

fp.so.x (bsd[i], temp.b, time) -- array s22 (as22, as23, dts23, time) s23 (as23, dts23, asd, time) sd (asd, bsd, ysd, xsd, time) ma (ama, xsd, xma, ysd, time) -- sink par i=[0 for 2] var float temp.x(to) : seq fp.si.x (xma[i], temp.x, time) seq j=[0 for time] x[(i*time)+j] := temp.x[j] : ------ Main --var float a1[4*to], a2[4*to], b[2*to], x[2*to] ; var n, time : seg get (n, " size of problem ") time := (n + 5)fp.get.n (a1, (4*time), " a1 input seq ")
fp.get.n (a2, (4*time), " a2 input seq ")
fp.get.n (b, (2*time), " b input seq ") system (a1, a2, b, x, time)
fp.put.n (x, (2*time), " x output seg ")

___ -- A.2.4 ----- Square array of processors for the updating of the LU -- factors in Simplex method. Matrix U is shifted one column -- to the left and a new nth column is added. ___ external proc get (var v, value s[]) : external proc fp.get.n (var float v(), value n, s()) : external proc fp.put.n (value float v(), value n, s()) : -- Maximum size of problem. def no = 10 : -- Diagonal cell. proc diag (var float u, eo, se, wo, value float nw, n, ei, value start. t =if t = start if nw = 0.0wo := 0.0 true wo := (u / nw) t = (start + 1)seq par u := ei - (wo + n)eo := wo se := u : --- Upper Diagonal cell. -proc updi (var float u, eo, s, wo, value float n, ei, wi, value start, t) = 1f t = start wo := u t = (start + 2)seq par eo := wi u := ei - (wi + n)s := u : ----- First Lower Diagonal cell. _ proc ldil (var float 1, s, value float ei, wo, seg wo:=1 value start, t) = if t = start seq 1 := (1 + ei)s := ei : -------- Lower Diagonal cell. proc lodi (var float 1, s, value float n, ei, wo, seg wors] value start, t) = if t = start seg

.

Ŧ

```
1 := 1 + (n + ei)
        s := n :
-- Generic cell for main diagonal.
proc gend ( chan nwin, nin, eout, ein, seout, sout, wout, win,
            var float x,
            value start, time ) =
  var float nw, n, eo, ei, se, s, wo, wi :
  seq t=[0 for time]
    seq
      par
        nwin 7 nw
        nin 7 n
        eout ! eo
        ein ? ei
        seout ! se
        sout i s
        wout 1 wo
        win 7 wi
      diag(x, eo, se, wo, nw, n, ei, start, t) ;
-- Generic cell for other diagonals.
proc gene { chan nin, eout, ein, sout, wout, win,
             var float x,
             value type, start, time ) =
   var float n, eo, ei, s, wo, wi :
   seq t=[0 for time]
     seq
       par
         nin 7 n
         eout 1 eo
         ein ? ei
         sout [ $
         wout 1 wo
         win 7 wi
       ΞĒ
         type = 0
           updi(x, eo, s, wo, n, ei, wi, start, t)
         type = 1
            ldil(x, s, ei, ostart, t)
         true
           lodi(x, s, n, ei,w,start, t) :
 -- Dummy source.
 proc soud ( chan xout,
              value time } =
   seg t=[0 for time]
     xout 1 0.0 :
  -- Dummy sink.
  proc sind ( chan xin,
              value time ) =
    seq t=[0 for time]
      xin ? any :
  -- Source for elements of new column a.
  proc soua ( chan xout,
              value float x,
              value start, time ) -
    seq t=[0 for time]
if
```

```
t = start
       xout I X
      true
        xout 1 0.0 :
-- System configuration.
proc syst ( var float lu[],
            value float a(),
            value n, time ) =
  chan v.c[no*(no+1)], hl.c[no*(no+1)], h2.c[no*(no+1)], d.c[no+1] :
  -- Main Array.
  ---
  proc sqar ( var float lu[],
               value n, time ) =
    par i=[0 for n]
      par j=[0 for n]
        var k :
        seq
          \vec{k} := (i + n) + j
           if
                                     -- main diagonal cells
             i = i
               gend(d.c[i], v.c[k], hl.c[(k+1)+i], h2.c[(k+1)+i], d.c[i+1],
                    v.c[k+n], h2.c[k+i], h1.c[k+i], lu[k], (2*i), time)
                                     -- upper diagonal cells
             1 < 1
               gene(v.c[k], h1.c[(k+1)+i], h2.c[(k+1)+i], v.c[k+n],
                    h2.c[k+i], h1.c[k+i], lu[k], 0, ((i+j)-1), time)
                                     -- first lower diagonal cells
             i = (j + 1)
               gene(v.c[k], h1.c[(k+1)+i], h2.c[(k+1)+i],v.c[k+n],
                    h2.c[k+i], h1.c[k+i], lu[k], 1, ((2*i)+1), time)
                                     -- rest lower diagonal cells
             true
               gene(v.c[k], hl.c[(k+1)+i], h2.c[(k+1)+i],v.c[k+n],
                    h2.c[k+i], h1.c[k+i], lu[k], 2, ((i+j)+2), time) :
   -- Sources - Sinks.
   proc sosi ( value float a[],
               value n, time ) =
     par
       par i=[0 for n]
         par
           soud(v.c[i], time)
           sind(v.c[(n*n)+i], time)
           soud(h1.c[(n+1)*i], time)
           sind(h1.c[((n+1)*i)+n], time)
           soua(h2,c[((n+1)*i)+n], a[i], (n+i), time)
           sind(h2.c[(n+1)*i], time)
       soud(d.c(0), time)
       sind(d.c[n], time) :
   -----
   par
     sgar(lu, n, time)
     sosi(a, n, time) :
 -- Main.
 ----
 var float lu[no*no], a[no] :
 var n, time :
 seq
   get(n, " problem size ")
   fp.get.n(lu, (n*n), " LU row-wise ")
   fp.get.n(a, n, " new column ")
   time := (2 * n)
   syst(lu, a, n, time)
   fp.put.n(lu, (n*n), " LU updated ")
```

I.

S

Q

N

```
-- A.2.5
---
-- Systolic Array for the Updating of the LU factors
-- of a full (n*n) matrix A: the first column of A is
-- removed and a new nth column (An) is added (Simplex).
-- For the updating: the new nth column in the form
-- a = ({L})^{**-1} is added to U and the main diagonal is
-- removed; L is modified accordingly. It is assumed that
-- no permutations needed.
___
external proc get ( var n, value s[] ) :
external proc fp.get.n ( var float v(), value n, s() ) :
external proc fp.put.n ( value float v(), value n, s() ) :
def no = 10, to = (2 * no) + 1 :
-- Divider calculating r := u[i,i] / u'[i-1,i-1];
-- Adder calculating l'[i,i-1] := l[i,i-1] + r.
 -- Output : 1'[1,1-1]; r.
 _----
proc dva ( chan ein, win, sin, eout, wout, nout,
            value time )=
   var float r, u[2], 1[2] :
   seq
     - initialisation
     par
       par i=[0 for 2]
         par
           u[i] := 0.0
           1(1) := 0.0
       r := 0.0
     -- main operation
     seq i=[0 for time]
       seq
         -- 1/0
         par
           ein 7 u[1]
            win 7 1(0)
            sin 7 u[0]
            eout l r
            wout 1 r
            nout | 1[1]
          -- calculations
          if
            u[1] = 0.0
             r := 0.0
            true
              r := (u(0) \neq u(1))
          1(1) := (1(0) + r):
  -- Inner Product calculating : u'[i,j] := u[i,j] - r * u'[i-1,j]
  -- Propagates u'[i,j] and r in opposite directions.
  --
  proc ipu ( chan ein, win, sin, eout, wout, nout,
             value time )-
    var float r[2], u[3] :
    seq
      __ initialisation
      par
        par i=[0 for 2]
          r[i] := 0.0
        par i=[0 for 3]
           u[i] := 0.0
       -- main operation
       seq i=[0 for time]
```

```
seq
       - i/o
       par
          ein ? u[1]
          win 7 r[0]
          sin 7 u[0]
          eout 1 r[1]
          wout | u[2]
          nout 1 u[2]
        -- calculation
        par
          u[2] := u[0] - (u[1] * r[0])
          r(1) := r(0) :
-- Inner Product calculating : l'[i,j] := l[i,j] + r * l[i,j+1]
-- Propagates 1(i, j+1), r in opposite directions.
proc ipl ( chan ein, win, sin, eout, wout, nout,
           value time )=
  var float 1[4], r[2] :
  seq
    __ initialisation
    par
      par i=[0 for 4]
        1[i] := 0.0
      par i=[0 for 2]
        r[i] := 0.0
     -- main operation
     seg i=[0 for time]
       seq
         -- 1/o
         par
           ein 7 r[0]
           win 7 1(1)
           sin ? 1[0]
           eout 1 1[2]
           wout [ r[1]
           nout 1 1[3]
         -- calculation
         par
           [1(3) := 1(1) + (r(0) + 1(0))
           1[2] := 1[0]
           r(1) := r(0) :
 ---
 -- Delay-branching cell.
 proc dba ( chan ain, ulout, u2out,
            value time ) =
   var float a[2] :
   seq
     initialisation
     par i=[0 for 2]
       a(i) := 0.0
     -- main operation
     seq i=[0 for time]
       seg
          -- i/o
          par
           ain 7 a[0]
            ulout [ a[1]
            u2out ! a[1]
          -- calculation
          a[1] := a[0] :
  -- Source for a vector x.
  ~-
```

- 593 -

```
proc srx ( chan xout ,
           value float x().
           value time ) -
  sed i=[0 for time]
    xout | x[i] :
-- Sink for a vector x.
proc six ( chan xin,
           var float x[],
           value time ) =
  seq i=[0 for time]
    xin 7 x(i) :
-- Dumm sink.
proc sid ( chan xin,
           value time ) =
  seg i={0 for time}
    xin ? any :
-- Delay cell.
proc del ( chan xin, xout,
           value time ) -
  var float x[2] :
  seq
    par i=[0 for 2]
      x(i) := 0.0
    seg i=[0 for time]
      seq
        par
          xin 7 x[0]
          xout | x[1]
        x[1] := x[0]:
-- System configuration : n-1 ipu, 1 dva and n-2 ipl linearly
-- interconnected; L, U matrices enter the array diagonally,
-- together with the new column a. Output the updated matrices
-- L1, U1 .
proc system ( var float 11(], u1(),
               value float 1[], u[],
               value n, time ) -
  -- data communication.
  chan ru.c[no], rl.c[no-1], ui.c[no+1], uo.c[no], li.c[no-1],
        lo.c[no-1], u.c[no], l.c[no-1] :
  par
    -- n vector sources / sinks for U-diagonals; the source
    -- vectors have last element from new column a.
    par i=[0 for n]
      var float uti[to], uto[to] :
       seg
         par j=[0 for time]
           uti[j] := u[(i*time)+j]
         par
           srx(ui.c[i], uti, time)
           six(uo.c[i], uto, time)
         par j=[0 for time]
           u1[(i*time)+j] := uto[j]
     -- 1 vector source for the first element of new column a.
    var float uti[to] :
     seq
       par j=[0 for time]
         uti[j] := u((time*n)+j)
       srx(ui.c[n], uti, time)
```

-- n-1 vector sources / sinks for L-diagonals. par i=(0 for (n-1))var float lti[to], lto[to] : seq par j=[0 for time] lti[j] := l((i*time)+j) par srx(li.c[i], lti, time) six(lo.c[i], lto, time) par j=[0 for time] 11((i+time)+j) := lto[j] -- 2 dummy vector sinks for r. sid(ru.c[n-1], time) sid(rl.c[n-2], time) -- delay-branching cell for the first element of column a.. dba(ui.c(n), u.c(n-1), uo.c(n-1), time) -- delay for lowest L-diagonal. del(li.c[n-2], l.c[n-2], time) -- main systolic array. par i=[0 for (n-1)]ipu(u.c[i+1], ru.c[i], ui.c[i+1], ru.c[i+1], u.c[i], uo.c[i], time) par i=[0 for (n-2)]ipi(rl.c(i], l.c(i+1], li.c(i], l.c(i), rl.c(i+1], lo.c(i+1], time) dva(u.c(0), 1.c(0), ui.c(0), ru.c(0), rl.c(0), lo.c(0), time) : ----- Main --var float l[(no-1)*to], u[(no+1)*to], ll[(no-1)*to], ul[no*to] : var n, time : seg get(n, " size of matrix ") time := (2 * n) + 1fp.get.n(u, ((n+1)*time), " U data seq ")
fp.get.n(l, ((n-1)*time), " L data seq ") system(11, u1, 1, u, n, time) fp.put.n(ul, (n*time), " Ul data seg ") fp.put.n(11, ((n-1)*time), " L1 data seg ")

- 594

```
-- A.2.6
-- Systolic array performing Gauss elimination with
-- partial pivoting on a tridiagonal matrix. An upper
-- triangular matrix, with three diagonals is produced;
-- also the multipliers and the pivoting information.
-- Computation time: 2n+2.
-- The array can be modified to accept symmetric
-- tridiagonal matrices; and can be extended to modify
-- the r.h.s vector, at the same time.
external proc get (var v, value s[]) :
external proc fp.get.n (var float v[], value n, s[]) :
external proc fp.put.n (value float v[], value n, s[]) :
external proc put.n (value v[], value n, s[]) :
external proc fp.so.x (chan xout, value float x[], value time ):
external proc fp.so.d (chan xout, value time ):
external proc fp.si.x (chan xin, var float x[], value time ):
external proc si.x (chan xin, var x[], value time ):
def to = 20:
---
-- Boundary cell calculating multipliers and pivoting control.
proc pmc ( chan uin, bin, pout, mout, cout,
           value time )=
  var float u[2], b[2], p[2], m, x :
  var c:
  seq
    -- initialisation
    par
       par 1=[0 for 2]
        par
           u[i] := 0.0
           b[i] := 0.0
          p[i] := 0.0
       m := 0.0
       x := 0.0
      c := 0.0
     -- main operation
     seq i={0 for time}
       seq
      -- input / output
         par
           uin 7 u[0]
           bin 7 b[0]
           pout 1 p[0]
           mout 1 m
           cout I c
         -- calculation
         par
           i£
             u(0) < 0.0
               u(1) := (-u(0))
             true
               u[1] := u[0]
           if
             b[0] < 0.0
               b[1] := (-b[0])
             true
               b[1] := b[0]
         1f
           b[1] \rightarrow u[1]
             par
               c := 1
```

p[0] := b[0]x := u[0]true par c := 0 p(0) := u(0)x := b[0]1f p[0] = 0.0p[1] := 0.000001 true p[1] := p[0] m := x/p[1] :-- IPS cell producing one off-diagonal element (q,r) -- for given c,m and v,a. proc gr (chan vin, cin, min, ain, vout, cout, mout, gout, value time) = var float v(2), m(2), a, q : var c[2] : seq --- initialisation par par i=[0 for 2] par v(i) := 0.0m[i] := 0.0c[i] := 0a := 0.0 g := 0.0 -- main operation seq i=[0 for time] seq -- input / output par vin ? v[0] cin ? c[0] min 7 m[0] ain 7 a vout [v[1] cout 1 c[1] mout [m[1] qout 1 q -- calculation par c[1] := c[0]m[1] := m[0]if c[0] = 1par g := a v[1] := v[0] - (m[0] * a)true par q := v[0]v[1] := a - (m[0] + v[0]) :-- System configuration proc system (var float p[], g[], r[], m[], var c[], value float a[], bu[], bl[], value time } = chan a.c, bu.c, bl.c, p.c, q.c, r.c, m.c[3], c.c[3], u.c[3] :par

595 -

pmc (u.c[0], bl.c, p.c, m.c[0], c.c(0], time) gr (u.c[1], c.c[0], m.c[0], a.c, u.c[0], c.c[1], m.c[1], g.c, time) gr (u.c[2], c.c[1], m.c[1], bu.c, u.c[1], c.c[2], m.c[2], r.c, time) fp.so.x (a.c, a, time) fp.so.x (bu.c, bu, time) fp.so.x (bl.c, bl, time) fp.so.d (u.c[2], time) fp.si.x (p.c, p, time) fp.si.x (q.c, q, time) fp.si.x (r.c. r, time) fp.si.x (m.c[2], m, time) si.x (c.c[2], c, time) : ----- Main var float a(to), bu(to), bl(to), p(to), q(to), r(to), m(to) : var c(to), n, time : seq get (n, " size of matrix ")
time := (2 * n) + 2 fp.get.n (a, time, " main diagonal stream ") fp.get.n (bu, time, " upper diagonal stream ") fp.get.n (bl, time, " lower diagonal stream ") system (p, q, r, m, c, a, bu, bl, time) system (p, q, t, m, c, a, bu, bi, time ;
fp.put.n (p, time, " main diagonal stream ")
fp.put.n (q, time, " first upper diagonal stream ")
fp.put.n (r, time, " second upper diagonal stream ")
fp.put.n (m, time, " multipliers stream ") put.n (c. time, " pivoting control stream ")

-- A.2.7 ----- Systolic Array for backsubstitution, for an upper -- triangular matrix, with three diagonals. It is used for the -- second step of the Inverse Iteration method. -- Time : 2n+2. external proc get (var v, value s[]) : external proc fp.get (var float v, value s[]) : external proc fp.get.n (var float v[], value n, s[]) : external proc fp.put.n (value float v[], value n, s[]) : external proc fp.so.d (chan xout, value time) : external proc fp.so.x (chan xout, value float x(), value time) : external proc fp.si.d (chan xin, value time) : external proc fp.si.x (chan xin, var float x[], value time) : def no = 10, to = ((2*no)+2): -- Bi-Directional Inner Product Step cell proc bdips (chan ain, xin, xout, yin, yout, value time } = var float a, x[2], y[2] : seq -- initialise par a :+ 0.0 par i=[0 for 2] par x[i] := 0.0y(i) := 0.0-- main operation seq i=[0 for time] seq -- i/o Dar ain 7 a xin 7 x[0] yin 7 y[0] xout [x[1] yout ! y[1] -- calculation par y[1] := y[0] + (a + x[0]) $\bar{x}[1] := \bar{x}[0] :$ -- Boundary cell ----proc bound (chan ain, bin, yin, xout, value time) = var float a, b, y, x : seq -- initialise par a := 0.0 b := 0.0 y := 0.0x := 0.0-- main operation seg i=[0 for time] seg -- i/o par ain 7 a bin 7 b

ហ

Q

δ

1

```
yin 7 y
          xout 1 x
        -- calculation
        if
          a = 0.0
            x := 0.0
          true
            x := (b - y) / a :
-- System configuration
proc syst ( value float a[], b[],
            var float x[].
            value time ) -
  chan a.c[3], b.c, x.c[3], y.c[3] :
  par
    -- sources
    par i=[0 for 3]
      var float a.temp[to] :
      sea
        seq j=[0 for time]
          a.temp[j] := a[(i*time)+j]
         fp.so.x ( a.c(i), a.temp, time )
    fp.so.x ( b.c, b, time )
    fp.so.d ( y.c[2], time )
    -- array
    bound ( a.c[0], b.c, y.c[0], x.c[0], time )
bdips ( a.c[1], x.c[0], x.c[1], y.c[1], y.c[0], time )
    bdips ( a.c[2], x.c[1], x.c[2], y.c[2], y.c[1], time )
    -- sink
    fp.si.x ( x.c[2], x, time ) :
---
-- Main
var float a[3*to], b[to], x[to] :
var n, time :
seq
  get ( n, " order of matrix " )
  time := (2 * n) + 2
  fp.get.n ( a, (3*time), " diagonal seg - main first " )
  fp.get.n ( b, time, " r.h.s vector seg " )
  syst ( a, b, x, time )
  fp.put.n ( x, time, " result " )
```

```
-- A.2.8
-- Systolic Algorithm for a system of linearly
-- connected systolic processors. It calculates
-- the eigenvector of a symmetric tridiagonal
-- matrix for a given eigenvalue by means of the
-- method of inverse iteration.
external proc str.to.screen(value s[]):
external proc num.to.screen(value n):
external proc num.from.keyboard(var n):
external proc fp.num.to.screen(value float n):
external proc fp.num.from.keyboard(var float n):
    -- Max size of matrix.
def n = 10:
          -- the three diagonals of the matrix
          -- b1[0] = br(n-1] = 0,
          -- a[i] = a[i] - eigenvalue.
var float bl[n], a[n], br[n],
          -- multiplier vector
          m[n].
          -- control vector declared as var float
          -- for uniformity
          c[n],
          -- eigenvector
          x[n]
    -- Actual size of matrix.
var nl:
     --- Channels for m, c transfer
chan m.c(n+1], c.c(n+1],
     -- Not dedicated channels
     11.c[n+1], 12.c[n+1],
     r1.c[n+1], r2.c[n+1] :
-- Process performing the calculations that
-- take place during the inverse iteration in
-- row i of the matrix. Initially loaded with
-- bl(i), a(i), br(i) in p, q, r it finally
 -- contains x[i] in x.Positional information
 -- and the size of the matrix are given also.
---
proc invit(chan min, cin, mout, cout,
                 ill, il2, oll, ol2,
                 irl, ir2, or1, or2,
            var float p, q, r, m, c, x,
            value position, nl)=
  var float temp[6] :
  var ig, ib, t, tgl, tbl, tg2, tb2 :
  -- State performing gauss elimination with
  +- partial pivoting. It produces P, an upper
   -- triangular matrix with three diagonals
   -- stored in p, q, r.Also the multiplier
   -- and the row-exchange control is calculated.
   proc gauss1=
    proc gauss1.calc=
       seg
         -- absolute values for b[i], u
         -- temp[3]=abs(b[i]), temp[4]=abs(u)
         par
           if
             p < 0.0
               temp[3] := -p
```

597

```
true
         temp[3] := D
     if
       temp[0] < 0.0
         temp[4] := -temp[0]
       true
         temp[4] := temp[0]
   -- if abs(b[i]) >= abs(u) then c=1
   1 £
     temp[3] >= temp[4]
       temp[4] := 1.0
     true
       temp[4] := 0.0
   -- if c=0 swap p, q, r with u, v, w; w=0
   1 f
     temp[4] = 0.0
       seq
         par
           temp[3] := temp[0]
           temp[5] := temp[1]
           temp[2] := 0.0
         par
           temp[0] := p
           temp[1] := q
           temp[2] := r
         par
           p := temp[3]
           g := temp[5]
           r := temp[2]
   -- m=u'/p' where u' and p' are the values of
   -- u and p after the interchange. Check first
   -- for zero
   par
     if
       temp[0] = 0.0
          temp[3] := 0.000001
        true
          temp[3] := temp[0]
     if
       p = 0.0
          temp[5] := 0.000001
        true
          temp[5] := p
    temp[3] := temp[3] / temp[5]
    -- new u=v'-m*g' and new v=w'-m*r'
    par
      temp[1] := temp[1] - (temp[3]*q)
      temp[2] := temp[2] - (temp[3]*r) :
-- gauss1 control and i/o
if
  t = ig
    par
      -- send a[i], b[i] to i-l process
      -- b[i-1] := b[i]
      oll ! q
      o12 1 r
      p := r
  t = (1g + 1)
    seq
      par
        -- receive m[i], c[i] and u, v from i-1
        -- and a[i+1], b[i+1] from i+1 process
        min ? m
        cin ? c
        ill ? temp[0]
        il2 ? temp[1]
```

```
it1 7 q
         ir2 ? r
       -- now p=b[i], g=a[i+1], r=b[i+1]
        --- temp[0]=u, temp[1]=v
        gauss1.calc
   t = (iq + 2)
     par
        -- send m[i+1], c[i+1], u, v to i+1 from
        -- locations returned by gauss1.calc
        mout 1 temp[3]
        cout 1 temp[4]
        orl | temp[1]
        or2 1 temp[2] :
-- State performing back substitution for
-- the system P * x = e, where P is the upper
-- triangular matrix produced from gauss1 and
-- e is the unity vector.
--
proc back1=
 proc back1.calc=
   seq
      par
        seq
          __ accumulate q*x[i+1]+r*x[i+2]
          temp[2] := 0.0 + (q * temp[0])
          temp[3] := temp[2] + (r * temp[1])
        -- check for zero
        1 f
          p = 0.0
            temp[4] := 0.000001
          true
            temp[4] := p
      -- calculate x[i]
      x := (1.0 - temp[3]) / temp[4] :
  -- back1 control and i/o
  if
    t = (ib + tgl)
      seq
        par
          -- receive x[i+1], x[i+2] from i+1
          ir1 ? temp[0]
          ir2 7 temp[1]
        back1.calc
    t = ((ib + 1) + tg1)
      par
        -- send x[i], x[i+1] to i-1 process
        ollix
        ol2 ! temp[0] :
-- State performing a forward substitution on
-- vector x produced by back1, by means of the
-- multiplier and control vectors produced by
-- gaussl.
proc gauss2=
  proc gauss2.calc=
    seg
       -- check for interchange
      if
        c = 1.0
           seg
             temp{1} := temp[0]
             temp[0] := x
             x := temp[1]
       -- calculate x
```

x := x - (m + temp[0]) :-- gauss2 control and i/o if t = (ig + tbl)seq -- receive x[i-1] from i-1 process ill ? temp[0] qauss2.calc t = ((iq + 1) + tb1)Dac -- send new x[i-1] to i-1 and -- x[i] to i+1 oll [temp[0] . or1 1 x t = ((ig + 2) + tbl)-- receive new x[i] from i+1 ir1 ? x : -- State performing back substitution for the -- system $P + x = \hat{y}$, where y is vector x as -- transformed after gauss2 and x is the same -- vector with the final solution. ---proc back2= proc back2.calc= seg par pea -- accumulate g*x[i+1]+r*x[i+2] temp[2] := 0.0 + (q + temp[0])temp[3] := temp[2] + (r + temp[1])-- check for zero if p = 0.0temp[4] := 0.000001 true temp[4] := p-- calculate x[i] x := (x - temp[3]) / temp[4] :-- back2 control and i/o if t = (ib + tg2)seg par -- receive x[i+1], x[i+2] from i+1 irl ? temp[0] ir2 ? temp[1] back2.calc t = ((ib + 1) + tg2)Dar -- send x[i], x[i+1] to i-1 oll ! x o12 ! temp[0] : -- The process has four states, gauss1, back1, gauss2, -- back2, each one implementing one part of the inverse -- iteration procedure. The state switching is controlled -- by the global clock t, as each state if completed in -- known time tgl, tbl, tg2, tb2. The position of the -- processor within the linear array of processors is -- given by ig (ib gives the reverse order) seq -- initialisation par par i={0 for 6}

proc back2= if t = (((n1 - 1) + 1) + tq2)par -- receive x{1}, x(2) irl ? any ir2 ? any : ---seq - initialisation par par i=[0 for 2] temp[i] := 0.0 t := 0 tq1 := n1 + 2tb1 := (2 + n1) + 3tg2 := (3 * n1) + 5 tb2 := (4 + n1) + 6-- state switching while t < tb2 seq if t < tgl gauss1 t č tbl backl t < tq2 gauss2 true back2 t := t + 1 : -- Boundary process for the right end of the array. -- It suplies process n1-1 with the necessary 1/0 so -- that calculation continues smoothly. It has the -- same structure as the main process but no -- clculations are required. proc bound.r(chan min, cin, ill, il2, ol1, ol2, value n1)= var float temp : var t, tg1, tb1, tg2, tb2 : -- gaussi 1/0 proc gauss1= if t = ((n1 - 1) + 1)par -- send a[n+1], b[n+1] 011 1 0.0 012 1 0.0 t = ((n1 - 1) + 2)par -- receive m[n+1], c[n+1], u, v min ? any cin ? any ill ? any il2 ? any : -- backl i/o proc back1= i£ t = tglpar -- send x[n+1], x[n+2] oll 1 0.0 012 1 0.0 :

- 599 -

```
temp[1] := 0.0
     ig := position
     ib := (n1 - 1) - position
     t := 0
     tq1 := n1 + 2
     tb1 := (2 + n1) + 3
     tq2 := (3 + n1) + 5
     tb2 := (4 + n1) + 6
   -- state switching
   while t < tb2
     seq
if
         t < tql
            qaussl
          t č tbl
            back1
          t < tq2
            gauss2
          true
            back2
        t := t + 1 :
-- Boundary process for the left end of the array.
-- It suplies process 0 with the necessary i/o so
-- that calculation continues smoothly. It has the
-- same structure as the main process but no
-- clculations are required.
proc bound.l(chan mout, cout,
                  irl, ir2, or1, or2,
             value n1)=
  var float temp[2] :
  var t, tgl, tbl, tg2, tb2 :
  -- gaussi i/o
  proc gauss1=
    if
      t = 0
         par
           -- receive and save a[1], b[1]
           ir1 ? temp[0]
           ir2 ? temp[1]
       t = 1
         par
           -- send m[1], c[1], a[1], b[1]
           mout 1 -1.0
           cout 1 1.0
           orl | temp[0]
           or2 1 temp[1] :
   -- backl i/o
   proc backl=
     if
       t = (((n1 - 1) + 1) + tg1)
         par
            -- receive x[1], x[2]
           irl ? any
           ir2 7 any :
   -- gauss2 i/o
   proc gauss2=
     if
        t = tbl
         -- send x{0}
          or1 1 0.0
        t = (tb1 + 1)
          -- receive new x[0]
          irl 7 any :
    -- back2 1/0
```

```
-- gauss2 1/0
proc gauss2=
  if
    t = \{((n1 - 1) + 1) + tb1\}
      -- receive and save x[n1]
      ill 7 temp
    t = (((n1 - 1) + 2) + tb1)
      -- send back x[n1]
      oll ! temp :
-- back2 1/0
proc back2=
  i£
    t = tg2
       par
         -- send x[n+1], x[n+2]
         oll ! 0.0
         012 1 0.0 :
 ---
 seq
   2_ initialisation
   par
     temp := 0.0
     t := 0
     tg1 := n1 + 2
     tb1 = (2 + n1) + 3
     tq2 := (3 * n1) + 5
     tb2 := (4 + n1) + 6
   -- state switching
   while t < tb2
     seq
       íf.
         t < tgl
           gaussi
          t č tbl
           backl
          t < tq^2
            gauss2
          true
            back2
        t := t + 1 :
-- The system comprises n1 linearly connected
-- processors with two boundary processors .
proc system=
  par
    bound.1(m.c[0], c.c[0],
            r1.c[0], r2.c[0],
            11.c(0), 12.c(0), n1)
    par i=[0 for n1]
      invit(m.c[i], c.c[i], m.c[i+1], c.c[i+1],
            11.c(i), 12.c(i), r1.c(i), r2.c(i),
            r1.c[i+i], r2.c[i+1], 11.c[i+1], 12.c[i+1],
            bl[i], a[i], br[i], m[i], c[i], x[i], i, ni)
    bound.r(m.c(n1), c.c(n1),
            11.c[n1], 12.c[n1],
            rl.c[n1], r2.c[n1], n1) :
proc get.f(var float v[],value n,s[])=
  seq
    str.to.screen("*c *n Input stream ")
     str.to.screen(s)
     seg i=[0 for n]
       seq
         fp.num.from.keyboard(v[i])
         fp.num.to.screen(v[1])
```

- 600 -

```
str.to.screen(" "):
proc get(var v[],value n,s[])=
  seq
     str.to.screen("*c *n Input stream ")
     str.to.screen(s)
     seg i=[0 for n]
        seq
          num.from.keyboard(v[i])
          num.to.screen(v[i])
          str.to.screen(" "):
proc get.i(var v, value s[])=
   seq
      str.to.screen("*c *n Input ")
      str.to.screen(s)
      num.from.keyboard(v)
      num.to.screen(v) :
 ----
 proc getdata =
    seg
      eg
get.i(nl, " n ")
get.f(bl, nl, " bl ")
get.f(a, nl, " a ")
get.f(br, nl, " br ") :
 proc put.f (value float v[],value n,s[])=
       str.to.screen("*c *n Output stream " )
     pea
       str.to.screen(s)
        seg i=[0 for n]
          seq
            fp.num.to.screen(v[i])
            str.to.screen(" ") :
   ---
  proc put (value v[],n,s[])=
        str.to.screen("*c *n Output stream *)
     seg
        str.to.screen(s)
        seg i=[0 for n]
           seq
             num.to.screen(v[i])
str.to.screen("") :
   proc putdata =
        seq
put.f(bl, nl, " bl ")
put.f(a, nl, " a ")
put.f(br, nl, " br ")
put.f(m, nl, " m ")
put.f(c, nl, " c ")
put.f(x, nl, " x ");
      seq
    --
    seq
       getdata
       system
       putdata
```

σ 01 T.

```
-- A.3.1
-- Systolic array for banded myips, y = y + C * x.
-- Matrix C has size (n*n) and bandwidth w = p+q-1.
-- Vectors x, y move in the same direction.
-- Computation time = n + w + p - 1, area = w.
external proc get(var v, value s[]):
external proc fp.get.n(var float v[], value n,s[]):
external proc fp.put.n(value float v(), value n,s()):
external proc fp.dl(chan xin, xout, valued, time):
external proc fp.lk(chan xin, xout, value time):
external proc fp.so.x(chan xout, value float v(), value time):
external proc fp.si.x(chan xin, var float v[], value time):
external proc fp.si.d(chan xin, value time):
external proc ips.1 (chan cin, xin, yin, xout, yout,
                     value time):
def no = 10, wo = ((2*no)-1), to = (no+(2*wo)) :
-- Array configuration : w ips cells are required.
-- One delay after each cell for x.
----
proc system (chan cin[], xin, yin, yout,
             value w, time } =
  chan c.c[wo], x.c[(2*wo)+1], y.c[wo+1] :
  par
    par i=[0 for w]
      par
        ips.l(c.c[i], x.c[i*2], y.c[i],
              x.c[(i*2)+1], y.c[i+1], time)
        fp.dl(x.c[(i*2)+1], x.c[(i*2)+2],1,time)
        fp.lk(cin[i], c.c[i], time)
    fp.lk(yin, y.c(0), time)
    fp.lk(xin, x.c[0], time)
    fp.lk(y.c[w], yout, time)
    fp.si.d(x.c[2*w], time) :
-----
   Source for matrix C
-- Initial delay of p-1 cycles for the uppermost diagonal;
-- additional delay of 1 cycle for the diagonals up to
-- the main; from then on 2 additional delays per diagonal.
proc source.c (chan cout[],
               value float c{],
               value n, w, p, time)-
  par j=[0 for w]
    var float c.temp[to] :
    var del :
     seq
       sec i=[0 for time]
        c.temp[i] := 0.0
       if
         j < p
           del := (p - 1) + j
         true
           del := (2 * i)
       seq i=[del for n]
         c.temp[i] := c[(j*n)+(i-del)]
       fp.so.x(cout[j], c.temp, time) :
 -- Source for vectors x, y.
 -- Initial delay of p-1 cycles for y.
 proc source.xy (chan xyout,
```

```
value float xy[],
                value n, del, time)=
 var float xy.temp(to) :
  seq
    seq i=[0 for time]
      xy.temp[1] := 0.0
    seg i=[del for n]
      xy.temp(i) := xy[i-del]
    fp.so.x(xyout, xy.temp, time) :
-- Sink for vector y.
-- Initial delay of w + p - 1 cycles.
proc sink.y (chan yin,
              var float y(),
              value n, del, time ) =
  var float y.temp[to] :
  seq
    seg i={0 for time}
      y.temp[i] := 0.0
    fp.si.x(yin, y.temp, time)
    seg i=[0 for n]
      v[i] := y.temp[i+del] :
-- Main
chan c.c[wo], x.c, y.c[2]:
var float c[no*wo], x[no], yi[no], yo[no] ;
var n, w, p, time :
seq
  get(n, " size of matrix C ")
get(w, " bandwidth of matrix C ")
get(p, " size of upper semiband ")
  fp.get.n(c, (w*n), " of matrix diagonals, upper diag. first ")
  fp.get.n(x, n, " vector x ")
  fp.get.n(yi, n, " vector y ")
  time := (n + w) + (p - 1)
  par
    source.c(c.c, c, n, w, p, time)
     source.xy(y.c[0], yi, n, (p-1), time)
     source.xy(x.c, x, n, 0, time)
     system(c.c, x.c, y.c[0], y.c[1], w, time)
     sink.y(y.c[1], yo, n, (w+(p-1)), time)
```

fp.put.n(yo, n, " vector y ")

602 -

```
-- A.3.2
-- Systolic pipeline for Jacobi, JOR iterative methods for the
-- solution of a linear system A^*x = b. The methods have the form
--x' = C*x + y, where C, y are derived from A, b for a given
-- overrelaxation factor. Matrix C has size (n*n) and bandwidth
-- w = p+q-1. Computation time = n+k*(w+p-1), where k is the
-- number of iterations: area = k*w.
external proc get(var v, value s()):
external proc fp.get.n(var float v[], value n,s[]):
external proc fp.put.n(value float v[], value n,s[]):
external proc fp.so.x(chan xout, value float v(), value time):
external proc fp.si.x(chan xin, var float v(), value time):
external proc fp.si.d(chan xin, value time):
external proc fp.dl(chan xin, xout, value d, time):
external proc fp.lk(chan xin, xout, value time):
external proc fp.br(chan xin, xlout, x2out, value time):
external proc ips.1 (chan cin, xin, yin, xout, yout,
                     value time):
def no = 10, wo = ((2*no)-1), ko = 10, to = (no+(ko*(2*wo))) :
-- Block configuration
-- A mvm array performs the mvips operation x'(-y) = C^*x+y;
-- (w+1) branching elements for the lines of C and y:
-- (w+p-1) delays for each line of C and y to next block.
proc system (chan cin[], xin[], yin[], cout[], xout[], yout[],
             value stage, w, delay, time ) =
  chan c.c[wo], x.c[(2*wo)+1], y.c[wo+1], c.1[3*wo], y.1[3] :
  par
    -- mvm array
    par i=[0 for w]
     par
        ips.l(c.c[i], x.c[i*2], y.c[i],
              x.c[(i*2)+1], y.c[i+1], time)
        fp.dl(x.c[(i*2)+1], x.c[(i*2)+2], 1, time)
    -- brancing and delays
    par
      par i=[0 for w]
        par
          fp.br(c.1[i], c.1[i+w], c.c[i], time)
          fp.dl(c.l[i+w], c.l[i+(2*w)], delay, time)
      fp.br(y.1{0}, y.1(1], y.c{0}, time)
      fp.dl(y.1[1], y.1[2], delay, time)
    -- i/o links
    par
      par i=[0 for w]
        par
          fp.lk(cin{(stage*w)+i], c.l[i], time)
          fp.lk(c.l(i+(2*w)), cout(((stage+1)*w)+i), time)
      fp.lk(yin(stage), y.l(0), time)
      fp.lk(y.1[2], yout[stage+1], time)
      fp.lk(xin(stage), x.c[0], time)
      fp.lk(y.c(w), xout(stage+1), time)
      fp.si.d(x.c[2*w], time) :
-- Source for matrix C
-- Initial delay of p-1 cycles for the uppermost diagonal;
-- additional delay of 1 cycle for the diagonals up to
-- the main; from then on 2 additional delays per diagonal.
proc source.c (chan cout)].
               value float c[],
```

par i=[0 for w] var float c.temp[to] : var del : pea. seg i=[0 for time] c.temp[1] := 0.0 if. j < p del := (p - 1) + j true del := (2 * j) seq i={del for n} c.temp[i] := c[(j*n)+(i-del)] fp.so.x(cout[j], c.temp, time) : -- Source for vectors x, y. -- Initial delay of p-1 cycles for y. proc source.xy (chan xyout, value float xy[], value n, del, time)var float xy.temp[to] : seq seg i=[0 for time] xy.temp[1] := 0.0seq i=(del for n) xy.temp[i] := xy[i-del] fp.so.x(xyout, xy.temp, time) : -- Sink for vector x. -- Initial delay of k*(w + p - 1) cycles. proc sink.x (chan xin, var float x[], value n, del, time} = var float x.temp[to] : seq seg i=[0 for time] x.temp[i] := 0.0 fp.si.x(xin, x.temp, time) seq i=[0 for n] x[i] := x.temp[i+del] : -- Sink for matrix C proc sink.c (chan cin[], value k, w, time) = par j=[0 for w]fp.sl.d(cin[(k*w)+j], time) : -- Main chan c.c[ko*wo], x.c[ko], y.c[ko] : var float c(no*wo), y(no), xi[no], xo(no) : var n, w, p, k, delay, time : seg -- input get(n, " size of matrix C ") get(w, " bandwidth of matrix C ") get(p, " size of upper semiband ") fp.get.n(c, (w*n), " of matrix diagonals, upper diag. first ") fp.get.n(y, n, " of r.h.s vector y ") fp.get.n(xi, n, " of initial vector x ") get(k, " no of iterations ") delay := w + (p - 1)

value n, w, p, time)=

```
time := n + (k * delay)
-- process
par
source.c(c.c, c, n, w, p, time)
source.xy(y.c[0], y, n, {p-1}, time)
source.xy(x.c[0], xi, n, 0, time}
par i=[0 for k]
system(c.c, x.c, y.c, c.c, x.c, y.c, i, w, delay, time)
sink.c(c.c, k, w, time)
fp.si.d(y.c[k], time)
sink.x(x.c[k], xo, n, (k * delay), time)
-- output
fp.put.n(xo, n, * of solution vector x *)
```

```
-- A.3.3
-- Systolic Preprocessor for the formulation of matrix
-- C and vector y in the J, JOR iterative methods : for a
--- (n*n) system Ax=b with a[i,i] \iff 0 i=1,2, ..., n and
-- an overrelaxation factor w, 0 < w < 2, the
-- preprocessor forms matrix C and vector y with :
             - (w * a[i,j]) / a[i,i] i <> j
___
-- c[i,j] ={
             (1 - w)
--
                                      1 = j
--y[i] = (w + b[i]) / a[i,i] , i = 1, 2, ..., n.
-- Linear array of (wa + 1) cells, where wa = p + q - 1
-- the bandwidth of A. Input and output reformatting delays so
-- that the data sequence conforms to the requirements of the
-- pipeline following.
external proc get(var v, value s[]) :
external proc fp.get(var float v, value s()) :
external proc fp.get.n(var float v[], value n, s[]) :
external proc fp.put.n(value float v[], value n, s[]) :
external proc fp.so.x(chan xout, value float x[], value time):
external proc fp.si.x(chan xin, var float x(), value time):
external proc fp.si.d(chan xin, value time):
external proc fp.dl(chan xin, xout, value d, time):
def wo = 10, to = 20 :
_ _
-- Main diagonal cell : it calculates w/a[i,i] and
-- propagates it to the other cells; it outputs (1-w).
proc madi ( chan ain, cout, wout, eout,
            value float w.
            value time ) -
  var float r[3] :
  seq
    par
      par i=[0 for 2]
        r[1] := 0.0
      r[2] := (1.0 - w)
    seq t=[0 for time]
      seq
        par
          ain 7 r[0]
          cout | r(2)
          wout [ r[1]
          eout 1 r[1]
        if
          r[0] = 0.0
            r[1] := 1.0
          true
            r[1] := (w/r[0]) :
-- Diagonal cell : it receives w/a[i,i] and calculates
-- -(w * a[i,j]) / a[i,i].
proc diag ( chan ain, cout, in, out,
            value time ) =
  var float r[4] :
  seg
    par i=[0 \text{ for } 4]
      r(i) := 0.0
    seg t=[0 for time]
      seg
        par
```

ain ? r[0]

1

δ

04

Т

```
cout 1 r[1]
          in ? r[2]
          out I r[3]
        par
          r[1] := (-(r[0] * r[2]))
          r(3) := r(2) :
-- Vector cell : it receives w/a[i,i] and calculates
-- (w * b[1]) / a[1,1].
proc vect ( chan bin, yout, in,
            value time ) =
  var float r[3] :
  seq
    par i=[0 for 3]
      r[i] := 0.0
    seg t=[0 for time]
      seg
        par
          bin ? r[0]
          yout | r[1]
          in 7 r(2)
         par
          r[1] := (r[0] * r[2]) :
-- Array configuration.
-- In delays : vector and upper-main diag : p-1,p-2,...,0
-- In delays : lower diag : 2,4,...,2(q-1)
-- Out delays : vector and upper-main diag : 0,1,3,...,2p-1
 -- Out delays : lower diag : 2p-1
proc system ( chan ain[], bin, cout[], yout,
               value float w.
               value wa, p, time ) =
   chan m.c[wo+1], a.c[wo], c.c[wo], b.c, y.c :
  par
     -- array, delays for A
     par i=[0 for wa]
       if
         i < (p - 1)
           par
             fp.dl(ain[i], a.c[i], ((p-1)-i), time)
             diag(a.c[i], c.c[i], m.c[i+1], m.c[i], time)
             fp.dl(c.c[i], cout[i], ((2*i)+1), time)
         i = (p - 1)
           par
             fp.dl(ain[i], a.c[i], 0, time)
             madi(a.c(i), c.c(i), m.c(i+1), m.c(i), w, time)
             fp.dl(c.c(i), cout(i), ((2*p)-1), time)
         true
           par
             fp.dl(ain[i], a.c[i], (2*(i-(p-1))), time)
             diag(a.c[i], c.c[i], m.c[i], m.c[i+1], time)
              fp.dl(c.c[i], cout[i], ((2*p)-1), time)
     -- cell, dealys for b
     par
       fp.dl(bin, b.c, p, time)
       vect(b.c, y.c, m.c(0], time)
        fp.dl(y.c, yout, 0, time)
     -- dummy sink for last lower diagonal cell
      fp.si.d(m.c[wa], time) :
  -- Main
  chan a.c(wo), c.c(wo), b.c, y.c i
  var float alwo*to), b[to], c[wo*to], y[to], w :
```

```
var n, wa, p, time :
seq
  -- oetdata
  get(n, " size of matrix A ")
  get(wa, " bandwidth of matrix A ")
 get(p, " upper semiband of matrix A ")
  time := n + (2 * wa)
  fp.get.n(a, (wa*n), " matrix A upper diagonal first ")
  fp.get.n(b, n, " vector b ")
  fp.get(w, " overrelaxation factor ")
  -- driver
  par
    system (a.c, b.c, c.c, y.c, w, wa, p, time)
    -- source for matrix A
    par 1-[0 for wal
      var float tempa(to) :
      sea
        seq j={0 for time}
          tempa(j) := 0.0
        seq j=[0 for n]
          tempa(j) := a((i*n)+j)
        fp.so.x(a.c[i], tempa, time)
    -- sink for matrix C
    par i=[0 for wa}
      var float tempc[to] :
      seq
        fp.si.x(c.c[i], tempc, time)
        seg j=[0 for time]
          c[(i*time)+j] := tempc[j]
    -- source for vector b
    var float tempb[to] :
    sea
      seq j=[0 for time]
        tempb{j} := 0.0
      seg j=[0 for n]
        tempb(j) := b(j)
      fp.so.x(b.c. tempb, time)
    -- sink for vector y
    fp.si.x(y.c, y, time)
  -- putdata
  fp.put.n(c, (wa*time), " matrix C ")
  fp.put.n(y, time, " vector y ")
```

1

```
-- A.3.4
-- Systolic pipeline for Jacobi iterative method for the
-- solution of a linear system A*x - b, where A is a 2-cyclic
-- matrix and the Cyclic Reduction technique is used.
-- Two parallel and coupled pipelines solve the systems
-x[1]^{r} = C[1]*x[2] + y[1] and x[2]^{r} = C[2]*x[1]^{r} + y[2],
-- where C[i], y[i], i=1,2 are derived from A, b.
-- Matrix C[i] has size (n*n), bandwidth w[i] = p[i]+g[i]-1.
-- Computation time = n + k + m, where k is the number of
-- iterations, and m = max ((w[1]+p[1]-1),(w[2]+p[2]-1));
-- area = k * (w[1] + w[2]).
external proc get(var v, value s[]):
external proc fp.get.n(var float v[], value n,s[]):
external proc fp.put.n(value float v[], value n,s[]):
external proc fp.so.x(chan xout, value float v(), value time):
external proc fp.si.x(chan xin, var float v(), value time):
external proc fp.si.d(chan xin, value time):
external proc fp.dl(chan xin, xout, value d, time):
external proc fp.lk(chan xin, xout, value time):
external proc fp.br(chan xin, xlout, x2out, value time):
external proc ips.l (chan cin, xin, yin, xout, yout,
                      value time):
def no = 10, wo = ((2*no)-1), ko = 10, to = (no+(ko*(2*wo))) :
-- Block configuration
-- A mvm array performs the mvips operation x'(-v) = C*x+v;
-- (w+1) branching elements for the lines of C and y;
-- m delays for each line of C and y to next block, and
-- m-(w+p-1) delays for x'.
proc system (chan cio[], yio[], xin[], xout[],
              value stage, w, p, m, time ) =
  chan c.c[wo], x.c[(2*wo)+1], y.c[wo+1], c.1[wo], y.1 :
  par
     -- mvm arrav
     par i=[0 for w]
      par
         ips.l(c.c[i], x.c[i*2], y.c[i],
               x.c[(i*2)+1], y.c[i+1], time)
         fp.dl(x,c[(i+2)+1], x.c[(i+2)+2], 1, time)
     -- i/o links, branching, delays
     par i=[0 for w]
       par
         fp.br(cio{(stage*w)+i}, c.l[i], c.c[i], time)
         fp.dl(c.l(i), cio(((stage+1)*w)+i), m, time)
     fp.br(yio[stage], y.l, y.c[0], time)
     fp.dl(y.l, yio[stage+1], m, time)
     fp.lk(xin[stage], x.c[0], time)
     fp.dl(y.c[w], xout(stage+1), (m-((w+p)-1)), time)
     fp.si.d(x.c(2*w), time) :
 -- Source for matrix C
 -- Initial delay of p-1 cycles for the uppermost diagonal;
 -- additional delay of 1 cycle for the diagonals up to
 -- the main; from then on 2 additional delays per diagonal.
 proc source.c (chan cout[],
                value float c[],
                value n, w, p, time)=
   par j=[0 for w]
     var float c.temp(to) ;
     var del :
```

seq seg i=[0 for time] c.temp[i] := 0.0 i£ j < p del := (p - 1) + jtrue del := (2 * j)seq i=[del for n] c.temp[i] := c[(j*n)+(i-del)] fp.so.x(cout[j], c.temp, time) : -- Source for vectors x, y. -- Initial delay of p-1 cycles for y. proc source.xy (chan xyout, value float xv[]. value n, del, time)= var float xy.temp[to] : seq seq i=[0 for time] xy.temp[1] := 0.0 seq i=[del for n] xy.temp[i] := xy[i-del] fp.so.x(xyout, xy.temp, time) : -- Sink for vector x. -- Initial delay of (k * m) cycles. proc sink.x (chan xin, var float x[], value n, del, time) = var float x.temp[to] : seq seg i=[0 for time] x.temp[i] := 0.0 fp.si.x(xin, x.temp, time) seq i=[0 for n] x[i] := x.temp[i+del] : -- Sink for matrix C proc sink.c (chan cin[], value k, w, time) = par j=[0 for w] fp.si.d(cin[(k*w)+j], time) : -- Main --chan cl.c[ko*wo], xl.c[ko], yl.c[ko], c2.c[ko*wo], x2.c[ko], y2.c[ko] : var float cl(no*wo), yl(no), xil(no), xol(no), c2[no*wo], y2[no], xi2[no], xo2[no] : var n, w1, p1, w2, p2, k, m, time : seq -- input get(n, " size of matrices C1, C2 ") get(w1, " bandwidth of matrix C1 ") get(pl, " size of upper semiband ")
fp.get.n(cl, (wl*n), " of matrix diagonals, upper diag. first ") fp.get.n(y1, n, " of r.h.s vector y1 ")
fp.get.n(x11, n, " of initial vector x1 ") get(w2, " bandwidth of matrix C2 ") get(p2, " size of upper semiband ") fp.get.n(c2, (w2*n), " of matrix diagonals, upper diag. first ") fp.get.n(y2, n, " of r.h.s vector y2 ")

fp.get.n(xi2, n, " of initial vector x2 ") get(k, " no of iterations ") Ϊ₽ (w1 + p1) >= (w2 + p2)m := ((w1 + p1) - 1)true m := ((w2 + p2) - 1)time := n + (k + m)-- process Dar source.c(cl.c, cl, n, w1, p1, time) source.xy(y1.c(0), y1, n, (p1 - 1), time) source.xy(x1.c[0], xi1, n, 0, time) source.c(c2.c, c2, n, w2, p2, time) source.xy(y2.c(0), y2, n, (p2 - 1), time) source.xy(x2.c(0), xi2, n, 0, time) par i=[0 for k] system(cl.c, yl.c, x2.c, x1.c, i, w1, p1, m, time) par system(c2.c, y2.c, x1.c, x2.c, i, w2, p2, m, time) sink.c(cl.c, k, w1, time) fp.si.d(y1.c[k], time) sink.x(x1.c(k), xol, n, (k * m), time) sink.c(c2.c, k, w2, time) fp.si.d(y2.c[k], time) sink.x(x2.c[k], xo2, n, (k * m), time) fp.put.n(xo1, n, " of solution vector x1 ")
fp.put.n(xo2, n, " of solution vector x2 ") -- output

-- A.3.5 --- Systolic pipeline for JOR iterative method for the -- solution of a linear system A*x = b, where A is a 2-cyclic -- matrix and the Cyclic Reduction technique is used. Two -- parallel and coupled pipelines solve the systems --x[1]' = C[1]*x[2] + y[1] + orf x[1] and--x[2]' = C[2]*x[1] + y[2] + orf x[2], where C[1], y[1], i=1,2-- are derived from A, b, using the overrelaxation factor orf-1. -- Matrix C[i] has size (n*n), bandwidth w[i] = p[i]+q[i]-1. -- Computation time = n + k + (m + 1), where k is the number of -- iterations, and m = max ((w[1]+p[1]-1),(w[2]+p[2]-1)); -- area = k + (w[1] + w[2] + 2). external proc get(var v, value s[]): external proc fp.get(var float v, value s[]): external proc fp.get.n(var float v[], value n,s[]): external proc fp.put.n(value float v[], value n,s[]): external proc fp.so.x(chan xout, value float v(), value time): external proc fp.si.x(chan xin, var float v(), value time): external proc fp.si.d(chan xin, value time): external proc fp.dl(chan xin, xout, value d, time): external proc fp.lk(chan xin, xout, value time): external proc fp.br(chan xin, xlout, x2out, value time): external proc ips.l (chan cin, xin, yin, xout, yout, value time): def no = 10, wo = ((2*no)-1), ko = 10, to = $\{no+(ko*(2*wo))\}$: -- A mvm array performs the mvips operation x'1(-y) = C*x2+y;-- (w+1) branching elements for the lines of C and y; -- (+1) delays for each line of C and y to next block, and -- m-(w+p-1) delays for x1'. Before the mvm array an ips cell -- computes y=y+orf*x1. There are reformating delays for the -- x1 stream and synchronization delays for C, x2. Notice that -- there are two input and ouput x streams. proc system (chan cio[], yio[], xio[], orfio[], xin[], xout[], value stage, w, p, m, time) = chan c.c[wo], x.c[(2*wo)+1], y.c[wo+1], c.1[2*wo], y.1[2], x.1[3], orf.1[2] : par -- mym array par i=[0 for w] par ips.l(c.c[i], x.c[i*2], y.c[i], x_c[(i*2)+1], y.c[i+1], time) fp.dl(x.c[(i*2)+1], x.c[(i*2)+2], 1, time) fp.si.d(x.c[2*w], time) -- additional ips cell ips.l(orf.1[1], x.1[0], y.1[1], x.1[1], y.c[0], time) fp.si.d(x.l[1], time) -- i/o, branching, delays for C par i=[0 for w] par fp.br(cio[(stage*w)+i], c.l[i], c.l[w+i], time) fp.dl(c.l[w+i], c.c[i], 1, time) fp.dl(c.l[i], cio(((stage+1)*w)+i), (m+1), time) -- i/o, branching, delays for y fp.br(yio[stage], y.1[0], y.1[1], time} fp.dl(y.l(0), yio[stage+1], (m+1), time) -- i/o, branching, delays for orf fp.br(orfio[stage], orf.1[0], orf.1[1], time) fp.dl(orf.l[0], orfio[stage+1], (m+1), time)

-- i/o branching, delays for x1, x2

607 -

```
fp.dl(xio[stage], x.1[0], (p-1), time)
   fp.dl(xin(stage), x.c[0], 1, time)
   fp.dl(y.c[w], x.l[2], (m-((w+p)-1)), time)
   fp.br(x.1[2], xio[stage+1], xout[stage+1], time) :
-- Source for matrix C
-- Initial delay of p-1 cycles for the uppermost diagonal;
-- additional delay of 1 cycle for the diagonals up to
-- the main; from then on 2 additional delays per diagonal.
proc source.c (chan cout[],
               value float c[],
               value n, w, p, time)=
  par i=[0 for w]
    var float c.temp[to] :
    var del :
    seq
      seg i=[0 for time]
        c.temp[i] := 0.0
      if
        j<p
          del := (p - 1) + j
        true
          del := (2 * i)
      seq i=[del for n]
        c.temp[i] := c[(j*n)+(i-del)]
      fp.so.x(cout(j), c.temp, time) :
-- Source for vectors x, y.
-- Initial delay of p-1 cycles for y.
proc source.xy (chan xyout,
                value float xy[],
                value n, del, time)=
  var float xy.temp(to) :
  seq
    seg i=[0 for time]
      xy.temp[i] := 0.0
    seg i=[del for n]
      xy.temp[i] := xy[i-del]
    fp.so.x(xyout, xy.temp, time) :
-- Source for orf.
proc source.orf (chan orfout,
                value float orf,
                value time)=
  var float orf.temp[to] :
  seq
    seg i=[0 for time]
      orf.temp[i] := orf
     fp.so.x(orfout, orf.temp, time) :
 -- Sink for vector x.
 -- Initial delay of k * (m + 1) cycles.
 proc sink.x (chan xin,
             var float x[],
             value n, del, time) =
   var float x.temp[to] :
   seg
    seq i=[0 for time]
      x.temp[i] := 0.0
     fp.si.x(xin, x.temp, time)
     seq i=[0 for n]
      x[i] := x.temp(i+del) :
```

```
--- Sink for matrix C
proc sink.c (chan cin[],
             value k, w, time) =
  par j=[0 for w]
    fp.si.d(cin[(k*w)+j], time) :
-- Main
chan cl.c[ko*wo], xl.c[ko], yl.c[ko], orfl.c[ko], c2.c[ko*wo],
     x2.c[ko], y2.c[ko], orf2.c[ko], x12.c[ko], x21.c[ko] :
var float cl[no*wo], yl[no], xil[no], xol[no], c2[no*wo],
          y2[no], x12[no], xo2[no], orf :
var n, w1, p1, w2, p2, k, m, time :
seq
  -- input
  get(n, " size of matrices C1, C2 ")
  -- data for pipeline 1
  get(w1, " bandwidth of matrix C1 ")
get(p1, " size of upper semiband ")
fp.get.n(c1, (w1*n), " of matrix diagonals, upper diag. first ")
   fp.get.n(y1, n, " of r.h.s vector y1 ")
   fp.get.n(xil, n, " of initial vector x1 ")
   -- data for pipeline 2
   get(w2, " bandwidth of matrix C2 ")
   get(p2, " size of upper semiband ")
   fp.get.n(c2, (w2*n), " of matrix diagonals, upper diag. first ")
   fp.get.n(y2, n, " of r.h.s vector y2 ")
   fp.get.n(xi2, n, " of initial vector x2 ")
   fp.get(orf, " 1 - overrelaxation factor ")
   get(k, " no of iterations ")
   i£
     (w1 + p1) >= (w2 + p2)
       m := ((w1 + p1) - 1)
     true
       m := ((w2 + p2) - 1)
   time := n + (k + (m + 1))
   -- process
   par
     -- sources for pipeline 1
     source.c(cl.c, cl, n, wl, pl, time)
     source.xy(y1.c(0), y1, n, (p1 - 1), time)
     source.xy(x1.c(0], xil, n, 0, time)
     source.xy(x12.c[0], xi1, n, 0, time)
     source.orf(orf1.c[0], orf, time)
     -- sources for pipeline 2
     source.c(c2.c, c2, n, w2, p2, time)
     source.xy(y2.c(0), y2, n, (p2 - 1), time)
     source.xy(x2.c[0], xi2, n, 0, time)
     source.xy(x21.c[0], xi2, n, 0, time)
     source.orf(orf2.c[0], orf, time)
      -- pipelines
     par i=[0 for k]
        par
          system(cl.c, yl.c, xl.c, orfl.c, x21.c, x12.c,
                 i, w1, p1, m, time)
          system(c2.c, y2.c, x2.c, orf2.c, x12.c, x21.c,
                 i, w2, p2, m, time)
      -- sinks for pipeline 1
      sink.c(cl.c, k, w1, time)
      fp.si.d(y1.c(k), time)
      sink.x(xl.c[k], xol, n, (k * {m + 1}}, time)
      fp.si.d(x12.c[k], time)
      fp.si.d(orfl.c[k], time)
```

à.

. 608 -

```
-- sinks for pipeline 2
sink.c(c2.c, k, w2, time)
fp.si.d(y2.c[k], time)
sink.x(x2.c[k], xo2, n, (k * (m + 1)), time)
fp.si.d(x21.c[k], time)
fp.si.d(orf2.c[k], time)
-- output
fp.put.n(xo1, n, " of solution vector x1 ")
fp.put.n(xo2, n, " of solution vector x2 ")
```

-- A.3.6 ___ -- Systolic array for the multiplication of a banded --- matrix C of size (n*n), bandwidth w = p+q-1, with a -- vector x : C*x-y.Vectors x,y move in the same direction -- and the computation time is n + w + p - 1. The diagonals -- of matrix C are stored one in each cell and accessed -- through an "address" moving systolically with x, y. external proc get(var v, value s[]): external proc get.n(var v[], value n, value s[]): external proc fp.get.n(var float v[], value n,s[]): external proc fp.put.n(value float v[], value n,s[]): -- Max size of matrix C; max bandwidth; max size of upper -- semiband; max size of address vector. def no = 10, wo = ((2*no)-1), po = no, ado = (no + wo) : -- Vectors of matrix C diagonals ; Vector x ; Vector y. var float c[ado*wo], x[no], y[no] : -- Actual size of matrix C ; bandwidth w ; semiband p. var n, w, p, -- Overall operation time; address vector and it size. time, ad, a[ado] : -- Channels for x's, y's and a's. chan x.c[(2*wo)+1], y.c[wo+1], a.c[wo+1] : -- Inner Product Step Cell: accumulates the inner products -- x*c(address) in y and propagates x, y, address. proc ips (chan xin, yin, ain, xout, yout, aout, value float c(], value time}= var float x[2], y[2]: var a(2): pea -- initialisation par par i=[0 for 2] par x[i] := 0.0y(i) := 0.0a(i) := 0 -- main operation seg i=[0 for time] seg -- i/o par xin 7 x[0] yin ? y(0)ain 7 a(0) xout $1 \times [1]$ vout I v[1] aout 1 a[1] -- calculation par x[1] := x[0]a[1] := a[0]y(1) = y(0) + (x(0) + c(a(0))) =-- Delay Cell : propagates its input with one cycle delay.

.

- 609

Т
```
-- initialisation
   par i=[0 for 2]
     x[1] := 0.0
   -- main operation
   seq i=[0 for time]
     seq
       -- i/o
       par
         xin 7 x[0]
         xout 1 \times[1]
        -- calculation
        x[1] := x[0] :
-- Source of the array : vectors x, y and a.
-- No delay for x,a ; a has n+wivalid elements.
---
proc source (chan xout, yout, aout,
             value float x[],
             value a[], n, ad, time)=
 seq i=[0 for time]
    par
      if
        i >≠ n
          xout 1 0.0
        true
          xout L x[1]
      yout 1 0.0
       ١£.
        i>= ad
          aout 1 0
         true
          aout 1 a[i] :
-- Sink of the array : vectors x, y and a.
-- Initial delay of w + p - 1 cycles for y.
 ___
proc sink (chan xin, yin, ain,
            var float y[],
            value w, p, time )=
   -- initial delay
  var del :
   seq
    del := w + (p - 1)
     seg i=[0 for time]
       par
         xin 7 any
         if
           i < del
             yin ? any
           true
             yin 7 y[i-del]
         ain ? any :
 -- Array configuration : w ips cells are required.
 -- One delay after each cell for x.Each cell has
 -- n+w-1 memory locations for a diagonal of matrix C.
 proc system =
   par
     source(x.c[0], y.c[0], a.c[0], x, a, n, ad, time)
     par i=[0 for w]
       --- load diagonal to cell
       var float c.temp[ado] :
       seq
          seq j=[0 for ad]
            c.temp(j) := c[(i*ad)+j]
```

```
par
           ips(x.c[i*2], y.c[i], a.c[i],
                x.c[(i+2)+1], y.c[i+1], a.c[i+1],
                c.temp, time)
           delay(x.c[(i*2)+1], x.c[(i*2)+2], time)
    sink(x.c(2*w], y.c(w], a.c(w], y, w, p, time) :
proc getdata =
  seg
    get(n, " size of matrix C ")
get(w, " bandwidth of matrix C ")
     get(p, " size of upper semiband ")
     time := (n + w) + (p - 1)
     ad := n + (w - 1)
     fp.get.n(c, (w*ad), " of matrix diagonals ")
    fp.get.n(x, n, " vector x ")
get.n(a, ad, " address vector ") :
---
proc putdata =
  seq
     fp.put.n(y, n, " vector y ") :
---
seq
  getdata
  system
  putdata
```

```
-- A.3.7
-- Iterative systolic array for matrix-vector multiplication.
---
-- The array performs k successive mvms, producing (A**k)*x,
-- where A is a (m*m) matrix, and x is a vector of m elements.
-- Area = (m+1) IPS cells and (m*m) memory.
-- Time = (2*m)*(k+1)+1 cycles.
external proc get ( var v, value s[] ) :
external proc fp.put ( value float v, value s[] ) :
external proc fp.get.n ( var float v(), value n, s(] ) :
external proc fp.put.n ( value float v(), value n, s() ) :
external proc fp.so.x ( chan xout, value float x(), value time ) :
external proc fp.si.x ( chan xin, var float x{], value time ) :
external proc fp.so.d ( chan xout, value time ) :
external proc fp.si.d ( chan xin, value time ) :
def mo=10, ko=10, to={{(2*mo)*(ko+1})+1) :
---
-- Inner Product Step Cell : Multiply - Accumulate
--- When the last accumulation occurs the result is sent as feedback
-- and the accumulator is reset for next cycle of calculations.
proc ips ( chan win, nin, nout, sin, sout,
            value cycle, reset, time ) =
   var float w[2], n[2], s[2], acc :
   seq
     __ initialise
     par
       acc := 0.0
       par i=[0 for 2]
         par
           w[i] := 0.0
           n[i] := 0.0
           s[i] := 0.0
     -- main operation
     seg i=[0 for time]
       seg
         -- 1/0
         par
           win 7 w[0]
           nin ? n[0]
            sin 7 s[0]
           nout [ n[1]
            sout 1 s[1]
          -- calculation
          acc := acc + (w[1] + n[0])
          Dar
            w[1] := w[0]
            n{1} := s[0]
            s[1] := n[0]
          -- control
          if
            ( i \ cycle ) = reset
              seq
                n[1] := acc
                acc := 0.0 :
  -- Multiplexer
  -- The initial vector enters the array through s.
  proc mux ( chan sin, sout, fin, fout,
             value cycle, reset, time ) =
    var float s, f[2] :
    seq
```

```
-- initialise
   par
     par i=[0 for 2]
       f[i] := 0.0
     s := 0.0
   -- main operation
   seq i=[0 for time]
     seq
       -- i/o
        par
          sin ? s
          fin ? f(0)
          sout 1 f[1]
          fout 1 f[1]
        f(1) := f(0) + B:
-- Source for initial vector: operates every two steps.
-- For the first mpy cycle it sents the initial vector, and
-- then it sends dummy elements.
proc so.n ( chan nout,
            value float x[],
            value cycle, time ) =
  var float temp(2*mo) :
  seq
    seq j=[0 for cycle]
      if
        (j \setminus 2) = 0
          temp[j] := x[j/2]
        true
          temp[j] := 0.0
    fp.so.x( nout, temp, cycle )
    fp.so.d( nout, (time-cycle) ) :
-- Source for a row i of matrix A.
-- Operates after an initial delay of i and every two steps,
-- for k mpy cycles; then it sends dummy elements.
proc so.w ( chan wout[],
             value float a[],
             value k, m, cycle, time ) =
   par i=[0 for m]
    var float temp(2*mo) :
     seq
       seg j=[0 for cycle]
        if
           (j ∖ 2) = 0
             temp[j] := a[(i*m)+(j/2)]
           true
             temp[1] := 0.0
       fo.so.d( wout[i], i)
       seq j=[0 for k]
         fp.so.x( wout[i], temp, cycle )
       fp.so.d( wout[i], (time - ((k*cycle) + i)) ) :
 -- Sink for results on the west side
 -- Collects the result in the last mpy cycle; it works
 -- every two steps.
 proc si.n ( chan nin,
             var float x[],
              value m, delay, cycle, time ) =
   var float x.temp[2*mo] :
   seq
     fp.si.d( nin, delay )
     fp.si.x( nin, x.temp, cycle )
```

5

- 611

```
seg j=[0 for cycle]
      if
        (j ∖ 2) = 0
         x[j/2] := x.temp[j] :
----
-- Array Configuration
-- Linear array of m IPS cells; north side has a Mux cell.
-- m west sources for matrix A; north source-sink for
-- initial and final vector; dummy source-sink in south.
-- Cycle is the time for one matrix-vector multiplication.
_
proc system ( value float a[], x[],
              value m, k,
              var float y[] ) =
  chan h.c[mo], u.c[mo+2], d.c[mo+2] :
  var time, cycle :
  seq
    cycle := (2 * m)
    time := (cycle * (k + 1)) + 1
    par
      -- ips cells
      par i=[0 for m]
        var reset :
        seg
          reset := ((cycle + (i - 1)) \ cycle)
          ips ( h.c[i], d.c(i+1), u.c[i+1], u.c[i+2], d.c[i+2],
                 cycle, reset, time )
       -- mux
       mux ( d.c(0), u.c(0), u.c(1), d.c(1), cycle, 0, time )
       -- sources, sinks
       so.w ( h.c, a, k, m, cycle, time )
       so.n ( d.c(0), x, cycle, time )
       si.n ( u.c(0), y, m, (time-cycle), cycle, time )
       fp.so.d( u.c(m+1), time )
       fp.si.d( d.c[m+1], time ) :
 -----
 -- Main
 --
 var float a[mo*mo], x[mo], y[mo] :
 var m, k :
 seg
   get ( m, " size of matrix A " )
   fp.get.n ( a, (m * m), " matrix A row-wise " ) .
   fp.get.n ( x, m, " vector x " )
   get ( k, " number of iterations ")
   system ( a, x, m, k, y )
   fp.put.n( y, m, " result " )
```

Α.

```
-- A.4.1
----
-- Systolic Array for matrix-matrix inner product step
-- X - A * B + C
external proc get ( var v, value s[] ) :
external proc fp.get.n ( var float v[], value n, s[] ) :
external proc fp.put.n ( value float v[], value n, s[] ) :
external proc fp.so.x ( chan xout, value float x(), value time ) :
external proc fp.so.d ( chan xout, value time ) :
external proc fp.si.x ( chan xin, var float x[], value time ) :
external proc fp.si.d ( chan xin, value time ) :
external proc fp.lk ( chan xin, xout, value time ) :
external proc ips.h ( chan ain, bin, cin, aout, bout, cout,
                      value time ) :
def no = 10, wo = 10, to = (no + wo) :
----
-- Array configuration
-- This is the hexagonnally-connected matrix multiplication array.
-- Two dummy diagonals for C, one on each side, for uniformity.
-- w is the bandwidth of A,B.
proc system ( chan ain[], bin[], cin[], xout[],
               value w, time ) =
   chan a.c[wo*(wo+1)], b.c[wo*(wo+1)], c.c[(wo+1)*(wo+1)] :
  par
     -- ips cells
     par i=[0 for w]
       par j=[0 for w]
         ips.h { a.c[(i*(w+1))+j], b.c[(i*w)+j], c.c[(i*(w+1))+j],
                 a.c[(i*(w+1))+(j+1)], b.c[(i*w)+(w+j)],
                 c.c[(i*(w+1))+(j+(w+2))], time )
     -- i/o links for A, B
     par i=[0 for w]
       par
         fp.lk ( ain[i], a.c[i*(w+1)], time )
         fp.lk ( bin[i], b.c[i], time )
         fp.si.d ( a.c[(i*(w+1))+w], time )
         fp.si.d ( b.c[(w*w)+1], time )
     -- i/o links for C, X
     par i=[0 for ((2*w)+1)]
       if
         i = 0
           par
             fp.so.d ( c.c(w], time )
              fp.si.d ( c.c[w], time )
         i <= w
           par
              fp.lk ( cin[i-1], c.c[w-i], time )
              fp.lk ( c.c(w+(i*(w+1))), xout(i-1], time )
         i < (2*v)
            par
              fp.lk ( cin[i-1], c.c[(w+1)*(i-w)], time )
              fp.lk ( c.c[(((w+1)*(w+1))-1)-(i-w)], xout[i-1], time )
          true
            par
              fp.so.d ( c.c[(w+1)*w], time )
              fp.si.d ( c.c[(w+1)*w], time ) :
  ---
 -- Main
  ---
 var float a[wo*to], b[wo*to], c[((2*wo)-1)*to], x[((2*wo)-1)*to] :
 chan a.c[2*wo], b.c[2*wo], c.c[2*((2*wo)-1)], x.c[2*((2*wo)-1)] :
 var n. w. time :
```

```
бeq
  -- getdata
 get ( n, " matrix order " )
 get ( w, " bandwidth for A, B " )
  time := (n + w)
 fp.get.n ( a, (w*time), " A sequence, upper diagonal first " )
fp.get.n ( b, (w*time), " B sequence, lower diagonal first " )
  fp.get.n ( c, (((2*w)-1)*time), " C sequence, upper diagonal first " )
  -- driver
  par
    par i=[0 for w]
      var float atemp[to], btemp[to] :
      seq
        seq j=[0 for time]
           par
             atemp[j] := a[(i*time)+j]
             btemp(j) := b[(i*time)+j]
         par
           fp.so.x ( a.c[i], atemp, time )
           fp.so.x ( b.c[i], btemp, time )
    par i=[0 for ((2*w)-1)]
      var float ctemp[to], xtemp[to] :
      seq
         seq j=[0 for time]
           ctemp[j] := c[(i*time)+j)
         par
           fp.so.x ( c.c[i], ctemp, time )
           fp.si.x ( x.c[i], xtemp, time )
         seq j=[0 for time]
           x[(i*time)+j] := xtemp[j]
     system ( a.c, b.c, c.c, x.c, w, time )
  -- putdata
```

```
fp.put.n ( x, (((2*w)-1)*time), * X sequence, upper diagonal first * )
```

4

δ

هسو

ω

I.

```
-- A.4.2
-- Systolic Pipeline for s successive squarings for a
-- (n*n) banded matrix of bandwidth w = p + q - 1, p = q.
-- area = SUM (((2**j)w - ((2**j)-1))**2), j=0,1,2...,s-1
-- time = n + SUM ((2**1)w - ((2**1)-1)), 1=0,1,2,.,s=1
-- final bandwidth = (2^{*}*s)w - ((2^{*}*s)-1),
-- channel count = SUM ((2**j)w - ((2**j)-1)), j=0,1,2...s
external proc get ( var v, value s() ) :
external proc fp.get.n ( var float v[], value n, s[] ) :
external proc fp.put.n ( value float v[], value n, s[] ) ;
external proc fp.so.x ( chan xout, value float x[], value time ) :
external proc fp.so.d ( chan xout, value time ) :
external proc fp.si.x ( chan xin, var float x(), value time ) :
external proc fp.si.d ( chan xin, value time ) :
external proc fp.lk ( chan xin, xout, value time ) :
external proc fp.dl ( chan xin, xout, value d, time ) :
external proc fp.br ( chan xin, xlout, x2out, value time ) :
external proc ips.h ( chan ain, bin, cin, aout, bout, cout,
                       value time ) :
def no=10, wi=3, so=3, wo=17, co=34, to=(no+17) :
 -- Matrix Squaring Block configuration.
 -- Two copies of A are produced, A=A and B=A transposed
 -- Delays for A(B) : q(p)-1, q(p)-2, ..., 1 for the q(p)-1 lower
 -- (upper) diagonals; here p=q=(w+1)/2.
 -- A hex-connected mmm array produces C (=A**2) = A * B.
 -- Two dummy diagonals for C, one on each side, for uniformity.
 -- s is the stage number; base is used for the mapping function
 -- of the channels.
 proc system ( chan ain[], aout[],
               value s, w, base, time ) =
   chan al.c[wo], bl.c[wo],
        a.c[wo*(wo+1)], b.c[wo*(wo+1)], c.c[(wo+1)*(wo+1)] :
   par
     -- branch
     par i=[0 for w]
       fp.br ( ain[base+i], al.c[i], bl.c((w-1)-i), time )
     -- delays
     var p :
     seg
       p := (w + 1) / 2
       par i=[0 for w]
         if
           i < p
             par
                fp.dl ( al.c[i], a.c[i*(w+1)], 0, time )
                fp.dl ( b1.c[i], b.c[i], 0, time )
            true
              par
                fp.dl ( al.c[i]; a.c[i*(w+1)], ((i-p)+1), time )
                fp.dl ( b1.c[i], b.c[i], ((i-p)+1), time )
      -- hex-array
      par i=[0 for w]
        par j=[0 for w]
          ips.h ( a.c[(i*(w+1))+j], b.c[(i*w)+j], c.c[(i*(w+1))+j],
                  a.c[(i*(w+1))+(j+1)], b.c[(i*w)+(w+j)],
                  c.c((i*(w+1))+(j+(w+2))], time )
      -- i/o links for A,B
      par i=[0 for w]
        par
          fp.si.d ( a.c[(i*(w+1))+w], time )
```

```
fp.si.d ( b.c[(w*w)+i], time )
   -- i/o links for C = A**2
   par i=[0 \text{ for } ((2*w)+1)]
     if
       1 = 0
         par
            fp.so.d ( c.c[w], time )
            fp.sid (c.c[w], time )
        i < w
          par
            fp.so.d ( c.c[w-i], time )
            fp.lk ( c.c[w+(i*(w+1))], aout[(base+w)+(i-1)], time )
        1 < (2*)
          par
            fp.so.d ( c.c((w+1)*(i-w)), time )
            fp.lk ( c.c[(((w+1)*(w+1))-1)-(i-w)],
                    aout[(base+w)+(i-1)], time )
        true
          par
            fp.so.d ( c.c[(w+1)*w], time )
            fp.si.d ( c.c[(w+1)*w], time ) :
---
-- Main
----
var float a[wi*no], c[wo*no] :
chan a.c(co) :
var n, wa, wc, s, p2, base, time r
seg
  -- getdata
  get ( n, " matrix order " )
  get (wa, "bandwidth for A : w = 2p - 1")
  fp.get.n ( a, (wa*n), " matrix A upper diagonal first " )
  get ( s, " squarings " )
  52 := 1
  base := wa
  seq i=[0 for s]
    seq
      p2 := (p2 * 2)
      base := base + ((p2 + wa) - (p2 - 1))
  wc := (p2 + wa) - (p2 - 1)
  base := base - wc
  time := n + base
  -- driver
  par
    -- input of matrix A
    par i=[0 for wa]
      var float atemp[to] :
      seq
        seq j=[0 for time]
          atemp[j] := 0.0
         seg j=[0 for n]
           atemp[j] := a[(i*n)+j]
         fp.so.x ( a.c(i], atemp, time )
     -- output of matrix C = A**2**s
     par i=[0 for wc]
      var float ctemp[to] :
       беа
         fp.si.x ( a.c[base+i], ctemp, time )
         seq j=[0 for n]
           c[(i*n)+j] := ctemp[base+j]
     -- pipeline of s blocks
     par i=[0 for s]
       var w, p2, base :
       seq
         p2 := 1
         base := wa
```

- 614 -

```
seg j=[0 for i]
       seq
         p_2 := (p_2 + 2)
         base := base + ((p2 * wa) - (p2 - 1))
     w := (p2 * wa) - (p2 - 1)
     base := base - W
     system ( a.c, a.c, i, w, base, time )
-- putdata
```

fp.put.n { c, (wc*n), " C sequence, upper diagonal first ")

```
--
```

```
-- A.4.3
--
```

--- Systolic Array Jacobi-Hotelling method.

```
-- Time Expansion.
```

-- A full mmm reusable array is used, with source-sink drivers

```
-- configured for the calculation of Successive Matrix Squares
```

```
-- after an initial Matrix Multpilication.
-- Matrix A is the Jacobi matrix obtained by the homogeneous
```

```
-- transformation of the original linear system of equations.
```

```
-- area <= (m+1)**2, where m is the order of the matrix A;
```

```
-- time = (2^{m})^{(s+2)+1}, where s, are the number of stages for
```

```
-- the matrix squaring.
```

```
external proc get ( var v, value s[] ) :
external proc fp.get.n ( var float v(), value n, s() ) :
external proc fp.put.n ( value float v[], value n, s[] ) :
external proc fp.so.x.t ( chan xout, value float x[], value t[],time ) :
external proc fp.si.x ( chan xin, var float x[], value time ) :
external proc fp.so.d ( chan xout, value time ) :
external proc fp.si.d ( chan xin, value time ) :
external proc fp.float ( value fix, var float flt ) :
external proc ips.r ( chan ein, eout, win, wout, nin, nout, sin, sout,
                      value cycle, reset, time ) :
external proc mux.r ( chan sin, sout, fin, fout,
```

```
value time ) :
```

```
def mo=10, so=10, to=((2*mo)*(so+2)), co=(mo+2):
---
```

```
-----
-- Source for row i of input matrices, west side.
-- Operates after an initial delay equal to i and every two steps.
```

```
-- For the first mpy cycle it sents the ith row of matrix A.
```

```
-- Then it sents dummy elements; fb is on.
```

```
proc so.w ( chan wout,
            value float a[],
            value i, m, cycle, time ) =
  var float temp(to), a.temp(to) :
  var fb[to] :
  seq
    -- input vectors
    par
      seq
        seq j=[0 for time]
          temp[]] := 0.0
        seq j=[0 for cycle]
          if
            (j \setminus 2) = 0
              a.temp[j] := a[(i*m)+(j/2)]
            true
              a.temp[j] := 0.0
        seq j=[0 for cycle]
          temp(j+i) := a temp(j)
      seq j={0 for time}
        fb[j] := 1
    -- source
    fp.so.x.t( wout, temp, fb, time ) :
```

```
---
-- Source for a row i of input matrices, north side.
-- Operates after an initial delay of i and every two steps
-- For the first mpy cycle it sents a column of matrix I
-- into the array.From then on dummy elements; fb is on.
```

```
proc so.n ( chan nout.
            value i, time ) -
```

σ ⊢ σ

var float temp[to] : var fb[to] : seq -- input vectors par seq seq j=[0 for time] temp[j] := 0.0 temp[(2*i)+i] := 1.0 seq j=[0 for time] fb[1] := 1 -- source fp.so.x.t(nout, temp, fb, time) : --- Sink for results on the west side -- Collects the first column in the last mpy cycle. proc si.w (chan rin, var float x{}, value i, cycle, time) = var float x.temp[to] : var delay : беа fp.si.x(rin, x.temp, time) delay := ((time + 1) - cycle) x[1] := x.temp[delay] : -- Array Configuration -- Square array of (m*m) IPS cells; west and north side have an -- additional row of m Mux's; the array is surrounded with -- 4m sources - dummy in east and south sides, and -- 4m sinks - dummy in east, north and south sides. -- Cycle is the time for one matrix-matrix multiplication. proc system (value float a[], value m, s, var float x[]) = chan r.c[mo*co], l.c[mo*co], u.c[mo*co], d.c[mo*co] : var time, cycle : seq cycle := (2 * m) time := (cycle * (s + 2)) + 1 par -- ips cells par i=[0 for m] par j={0 for m] var di, dj, reset : seq di := (i + (m + 2)) + (j + 1)dj := (j + (m + 2)) + (i + 1)reset := ((cycle + ((i + j) - 1)) \ cycle) ips.r (1.c[di+1], r.c[di+1], r.c[di], 1.c[di], d.c[dj], u.c[dj], u.c[dj+1], d.c[dj+1], cycle, reset, time) -- mux's par i=[0 for m] var wn : seq $\hat{w}_{n} := (1 + (m + 2))$ par mux.r (r.c[wn], l.c[wn], l.c[wn+1], r.c[wn+1], time) mux.r (d.c[wn], u.c[wn], u.c[wn+1], d.c[wn+1], time) -- sources, sinks par i=[0 for m] var wn, es : seg

```
wn := (i * (m + 2))
         es := wn + (m + 1)
         par
           so.w ( r.c(wn], a, i, m, cycle, time )
           so.n ( d.c[wn], i, time )
           si.w ( l.c[wn], x, i, cycle, time )
            fp.si.d( u.c(wn), time )
            fp.so.d( l.c[es], time
            fp.so.d( u.c[es], time
            fp.si.d( r.c[es], time )
            fp.si.d( d.c[es], time ) :
-- Main
```

```
var float a[mo*mo], x[mo] :
var m, s :
seq
  get ( m, " size of matrix A " )
  fp.get.n ( a, (m * m), " matrix A row-wise " )
  get ( s " iterations ")
  system ( a, m, s, x )
  fp.put.n( x, m, " result " )
```

```
-- 3.4.4
-- Systolic Array Gauss-Seidel-Hotelling method.
-- Time Expansion.
-- A full mmm reusable array is used, with source-sink drivers
-- configured for the calculation of a series of Matrix-Matrix
-- Multiplications followed by Successive Matrix Squarings.
-- Matrix A is the Jacobi matrix obtained by the homogeneous
-- transformation of the original linear system of equations.
-- area <- (m+1)**2, where m is the order of the matrix A;
-- time = (2*m)*(m+s+1)+1, where s, are the number of stages
-- for the matrix squaring.
external proc get ( var v, value s[] ) :
external proc fp.get.n ( var float v(), value n, s() ) :
external proc fp.put.n ( value float v(], value n, s(] ) :
external proc fp.so.x.t ( chan xout, value float x[], value t[], time ) :
external proc fp.si.x ( chan xin, var float x[], value time ) :
external proc fp.so.d ( chan xout, value time ) :
external proc fp.si.d ( chan xin, value time ) :
external proc fp.float ( value fix, var float flt ) :
external proc ips.r ( chan ein, eout, win, wout, nin, nout, sin, sout,
                       value cycle, reset, time ) :
external proc mux.r ( chan sin, sout, fin, fout,
                      value time ) :
def mo=10, so=10, to=((2*mo)*((mo+so)+1)), co=(mo+2):
---
-
-- Source for row i of input matrices, west side.
-- Operates after an initial delay equal to i and every two steps.
-- For j=0,1,...,m-1 mpy cycles it sents the ith row of matrix
-- I[j]+A[j], where A[j] has non-zero elements only in row j, while
-- I(j) is matrix I with row j equal to zero. Then it sents dummy
-- elements. fb is on for j=m,m+1,.. mpy cycles.
proc so.w ( chan wout,
            value float a[].
            value i, m, cycle, time ) =
  var float temp[to], a.temp[to] :
  var loc, pre, post, fb(to) :
  seq
     -- initialise
     par
      loc := i
      pre := (2 + i)
      post := (cycle - pre)
     -- input vectors
     par
      seq
         seq j=[0 for time]
           temp[j] := 0.0
         seg 1=[0 for m]
           if
             1 = 1
               sea
                 seg j=[0 for cycle]
                     (1 \setminus 2) = 0
                       a.temp[j] := a[(i*m)+(j/2)]
                     true
                       a.temp[j] := 0.0
                 seq j=[0 for cycle]
                    temp[loc+j] := a.temp[j]
                 loc := (loc + cycle)
```

true seg loc := (loc + pre) temp[loc] := 1.0loc i= (loc + post) seq seq j=[0 for time] fb[1] := 1 seq j=[0 for ((m*cycle)+(i-1))] fb[j] := 0-- source fp.so.x.t(wout, temp, fb, time) : -- Source for a row i of input matrices, north side. -- Operates after an initial delay of i and every two steps --- For the first mpy cycle it sents a column of matrix I -- into the array.From then on dummy elements; fb is on. proc so.n (chan nout, value i, time } = var float temp[to] : var fb[to] : seq -- input vectors par seq seq j=[0 for time] temp[j] := 0.0 temp[[2*1)+1] := 1.0 seq j=[0 for time] fb[j] := 1 -- source fp.so.x.t(nout, temp, fb, time) : -- Sink for results on the west side -- Collects the first column in the last mpy cycle. proc si.w (chan rin, var float x[], value i, cycle, time) = var float x.temp[to] : var delay : sea fp.si.x(rin, x.temp, time) delay := ((time + i) - cycle) x[i] := x.temp[delay] : -- Array Configuration -- Square array of (m*m) IPS cells; west and north side have an -- additional row of m Mux's; the array is surrounded with -- 4m sources - dummy in east and south sides, and -- 4m sinks - dummy in east, north and south sides. -- Cycle is the time for one matrix-matrix multiplication. proc system (value float a[), value m, s, var float x() =chan r.c[mo*co], l.c[mo*co], u.c[mo*co], d.c[mo*co] ; var time, cycle : seg cycle := (2 * m)time := (cycle * ((m + s) + 1)) + 1par -- ips cells par i={0 for m} par j=[0 for m]

- 617 -

var di, dj, reset : sea di := (i + (m + 2)) + (j + 1)dj := (j + (m + 2)) + (1 + 1)reset := ((cycle + ((i + j) - 1)) \ cycle) ips.r (1.c[di+1], r.c[di+1], r.c[di], 1.c[di], d.c[dj], u.c[dj], u.c[dj+1], d.c[dj+1], cycle, reset, time) -- mux's par i=[0 for m] var wn : seq wn := (1 * (m + 2))par mux.r (r.c[wn], l.c[wn], l.c[wn+1], r.c[wn+1], time) mux.r (d.c[wn], u.c[wn], u.c[wn+1], d.c[wn+1], time) -- sources, sinks par i=[0 for m] var wn, es : seq wn := (i + (m + 2))es := wn + (m + 1)par so.w (r.c[wn], a, i, m, cycle, time) so.n (d.c[wn], i, time) si.w (l.c[wn], x, i, cycle, time) fp.si.d(u.c(wn), time fp.so.d(l.c[es], time fp.so.d(u.c[es], time fp.si.d(r.c[es], time fp.si.d(d.c[es], time) : -- Main var float a[mo*mo], x[mo] : var m, s : sed get (m, " size of matrix A ") fp.get.n (a, (m * m), " matrix A row-wise ") get (s, "iterations) system (a, m, s, X) fp.put.n(x, m, " result ")

-- A.4.5 ÷ --- Systolic Pipeline for s iterations of the Jacobi--- Hotteling method for the solution of a linear system. -- A is the Jacobi (n*n) banded matrix of bandwidth --w = p + q - 1, p = q; y is the corresponding the vector, -- which is finally transformed to the solution vector. -- area = SUM (w(j) **2 + w(j)), j=0,1,2,...,s-1, and -- w(i) = (2**i)*w - ((2**i)-1) the bandwidth of stage j+1. -- time = n + SUM (w[i] + p[i] - 1), i = 0,1,2,...,s-1, and -p[j] = (w[j] + 1) / 2 the semiband of stage j+1. -- final bandwidth for matrix squaring = $(2^{**s})w - ((2^{**s})-1)$, -- channel count = SUM (w(j) + 1), j=0,1,2,...,s. external proc get (var v, value s[]) : external proc fp.get.n (var float v[], value n, s[]) : external proc fp.put.n (value float v[], value n, s[]) : external proc fp.so.x (chan xout, value float x(), value time) : external proc fp.so.d (chan xout, value time) : external proc fp.si.x (chan xin, var float x[], value time) : external proc fp.si.d (chan xin, value time) : external proc fp.lk (chan xin, xout, value time) : external proc fp.dl (chan xin, xout, value d, time) : external proc fp.br (chan xin, xlout, x2out, value time) : external proc ips.h (chan ain, bin, cin, aout, bout, cout, value time) : def no=10, wi=3, so=3, wo=17, co=34, to=(no+26) : ------- Matrix Squaring Block configuration. -- Two copies of A are produced, A=A and B=A transposed -- Delays for A(B) : q(p)-1, q(p)-2, ..., 1 for the q(p)-1 lower -- (upper) diagonals; here p=q=(w+1)/2. -- A hex-connected mmm array produces C (=A**2) = A * B. -- Two dummy diagonals for C, one on each side, for uniformity. proc hex (chan ain[], aout[], value w, p, time) chan al.c[wo], bl.c[wo], a.c[wo*(wo+1)], b.c[wo*(wo+1)], c.c[(wo+1)*(wo+1)] : par -- i/o links, branch delays for A,B par 1=[0 for w] par fp.br (ain[i], al.c[i], bl.c[(w-1)-i], time) iÌ i<p par fp.dl (al.c[i], a.c[i*(w+1)], 0, time) fp.dl (bl.c[i], b.c[i], 0, time) true par fp.dl (al.c[i], a.c[i*(w+1)], ((i-p)+1), time) fp.dl (bl.c[i], b.c[i], ((i-p)+1), time) fp.si.d (a.c((i*(w+1))+w), time) fp.si.d (b.c[(w*w)+i], time) -- hex-array par i=[0 for w] par j=[0 for w] ips.h (a.c[(i*(w+1))+j], b.c[(i*w)+j], c.c[(i*(w+1))+j], a.c[(i*(w+1))+(j+1)], b.c[(i*w)+(w+j)], c.c[(i*(w+1))+(j+(w+2))], time) -- i/o links, delays for C

1

δ

1-1-1

Ô

1

par i=[0 for ((2*w)+1)]

```
if
       1 = 0
         par
           fp.so.d ( c.c[w], time )
           fp.sl.d ( c.c[w], time )
       1 <- W
          par
            fp.so.d ( c.c[w-i], time )
           fp.dl ( c.c[w+(i*(w+1))], aout[i-1], (p-1), time )
        i < (2+w)
          par
            fp.so.d ( c.c[(w+1)*(i-w)], time }
            fp.dl ( c.c((((w+1)*(w+1))-1)-(i-w)), aout(i-1),
 }
                    (p-1), time )
        true
          par
            fp.so.d ( c.c((w+1)*w), time )
            fp.si.d ( c.c[(w+1)*w], time ) :
-- Mvips Block configuration
-- A mvm array performs the operation y = A^+x + y, ( x=y ).
-- Delay of (p-1) cycles for the uppermost diagonal; additional
-- delay of i cycle for the diagonals up to the main; from then
-- on 2 additional delays per diagonal. (p-1) delays for the copy
-- of y that is added.
proc lin ( chan ain[], yin, yout,
           value w, p, time ) =
  chan a.c[2*wo], x.c[(2*wo)+1], y.c[wo+2] :
  par
    -- linear array
    par i=[0 for w]
      par
        ips.h ( a.c[i], x.c[i*2], y.c[i+1],
                a.c[i+w], x.c[(i*2)+1], y.c[i+2], time )
        fp.dl ( x.c[(i*2)+1], x.c[(i*2)+2], 1, time }
    -- i/o, delays for A
    par i=[0 for w]
      par
        var del :
         seq
           Ξŧ
             i < p
               del := (p - 1) + i
             true
               del := (2 + i)
           fp.dl ( ain[i], a.c[i], del, time )
         fp.si.d ( a.c(i+w), time )
     -- i/o, branch, delays for x, y
     fp.br ( yin, y.c[0], x.c[0], time )
     fp.dl ( y.c(0], y.c(1), (p-1), time )
     fp.1k ( y.c[w+1], yout, time )
     fp.si.d ( x.c(2*w), time ) :
 -- Pipeline Block configuration
 -- It consists of one hex array producig A**2, and one linear
 -- array producing y+A*y. The output of the hex-array is delayed
 -- for (p-1)=((w+1)/2)-1 cycles for synchronisation.
 proc system ( chan a.c[], y.c[],
               value s, w, base, time ) =
   chan al.c[wo], a2.c[wo], a3.c[wo], y1.c, y2.c :
   var p :
   ьeq
     p := (w + 1)/2
     par
```

```
-- input links, branch
      par i=[0 for w]
        fp.br ( a.c[base+i], al.c[i], a2.c[i], time )
      fp.lk ( y.c[s], yl.c, time )
      -- hex, linear arrays
      hex ( al.c, a3.c, w, p, time )
      lin ( a2.c, y1.c, y2.c, w, p, time )
      -- output links
      par i=[0 for ((2*w)-1)]
        fp.1k ( a3.c[i], a.c[(base+w)+i], time )
      fp.1k ( y2.c, y.c[s+1], time ) :
⊷- Маіл
var float a[wi*no], yi[no], yo[no] :
chan a.c[co], y.c[so+1] :
var n, wa, w[so], wc, s, p2, base[so], delay, time :
seq
  -- getdata
  get ( n, " matrix order " )
get ( wa, " bandwidth for A : w = 2p - 1 " )
  fp.get.n ( a, (wa*n), " matrix A upper diagonal first " )
  fp.get.n ( yi, n, " vector y " )
  get ( s, " squarings " )
  -- setup
  p2 := 1
  w[0] := wa
  base[0] := 0
  delay := (wa + ((wa + 1) / 2)) - 1
  seg i = [1 \text{ for } (s-1)]
    seq
       p2 := (p2 + 2)
       w[i] := (p2 + wa) - (p2 - 1)
       base[i] := base[i] + w[i-1]
       delay := delay + ((w[i] + ((w[i] + 1) / 2)) - 1)
   time := n + delay
   wc := ((2*w[s-1])-1)
   -- driver
   par
     -- i/o of matrix A
     par
       par i=[0 for wa]
         var float atemp[to] :
         seq
           seq j=[0 for time]
             atemp[j] := 0.0
            seq j=[0 for n]
             atemp[j] := a[(i*n)+j]
            fp.so.x ( a.c[i], atemp, time )
       par i=[0 for wc]
         fp.si.d ( a.c[(base[s-1]+w[s-1])+i], time )
     -- i/o of vector y
     par
       var float ytemp[to] :
       seq
          seg i=[0 for time]
           vtemp[i] := 0.0
          seq i=[0 for n]
           ytemp[i] := yi[i]
          fp.so.x ( y.c[0], ytemp, time )
        var float ytemp[to] :
        sea
          fp.si.x ( y.c(s), ytemp, time )
          seq i=[0 for n]
            yo[i] := ytemp[i+delay]
      -- pipeline of s blocks
```

619 -

```
par i=[0 for s]
   system ( a.c, y.c, i, w[i], base(i], time )
   -- putdata
  fp.put.n ( yo, n, " solution vector " )
```

```
-- A.4.6
-- Systolic pipeline for Power Method
-- Area Expansion.
-- Calcualtion of the dominant eigenavalue e and the corresponding
-- eigenvector u of a (n*n) matrix C. The method has the form :
           y(i+1) = C*x(i); x(i+1) = y(i+1)/m(y(i+1)),
-- where m(x) is the first non-zero element of x; x[0]=[1 1 .. 1]
-1 = m(y[k]), u/m(u) = x[k], where k is the number of iterations.
-- Area = k*(w+1), where w = p+q-1 is the bandwidth of C.
-- Computation time = n+k*(w+p).
external proc get(var v, value s()):
external proc fp.put(value float v, value s[]):
external proc fp.get.n(var float v[], value n,s[]):
external proc fp.put.n(value float v[], value n,s[]):
external proc fp.so.x(chan xout, value float v(), value time):
external proc fp.si.x(chan xin, var float v(), value time):
external proc fp.so.d(chan xout, value time):
external proc fp.si.d(chan xin, value time):
external proc fp.dl(chan xin, xout, value d, time):
external proc fp.lk(chan xin, xout, value time):
external proc fp.br(chan xin, xlout, x2out, value time):
external proc ips.l (chan cin, xin, yin, xout, yout,
                      value time):
def no = 10, wo = \{(2*no)-1\}, ko = 10, to = \{no+(ko*(2*wo))\} :
 ---
 -- Normaliser
-- It waits for the first non-zero element of the vector and
 -- normalises the vector in respect to it. The eigenvalue is
 -- equal to the first non-zero element.
 ---
 proc nor (chan yin, yout, eout,
           value time) -
   var first.found :
   var float y[2], first :
   seq
     -- initialisation
     par
       first := 1.0
       par i=[0 for 2]
         y(1) := 0.0
       first.found := false
     -- main operation
     seq i=(0 for time)
       seq
         -- i/o
         par
           yin ? y[0]
           yout 1 y[1]
           eout | first
          -- calculation
         if
           (not first.found) and (y[0] \leftrightarrow 0.0)
               par
                first := v[0]
                 first.found := true
          y[1] := y[0] / first :
  -- Block configuration
 -- A mvm array performs the operation x'(-y) = C*x;
 -- w branching elements for C; (w+p) delays for C to next block.
 -- A normaliser cell is connected to the output of mvm array.
```

3

```
proc system (chan c.io[], x.io[], e.o[],
             value stage, w. delay, time ) -
  chan c.c[wo], x.c[(2*wo)+1], y.c[wo+2], c.1[3*wo], e.c :
 par
    -- mvm array
   par i=[0 for w]
      par
        ips.l(c.c[i], x.c[i+2], y.c[i],
             x.c[(1*2)+1], y.c[1+1], time)
        fp.dl(x.c[(i*2)+1], x.c[(i*2)+2], 1, time)
    -- brancing and delays
    Dar
      par i=[0 for w]
       par
          fp.br(c.l[i], c.l[i+w], c.c[i], time)
          fp.dl(c.l[i+w], c.l[i+(2*w)], delay, time)
    -- normaliser
    nor(y.c[w], y.c[w+1], e.c, time}
    -- i/o links
    par
      par i=[0 for w]
        par
          fp.lk(c.iof(stage*w)+i), c.l(i), time)
          fp.lk(c.l[i+(2*w)], c.io(((stage+1)*w)+i), time)
      fp.lk(x.io[stage], x.c[0], time)
      fp.lk(y.c[w+1], x.io[stage+1], time)
      fp.lk(e.c, e.o(stage), time)
      fp.so.d(y.c[0], time)
      fp.si.d(x.c[2*w], time) :
-- Source for matrix C
-- Initial delay of p-1 cycles for the uppermost diagonal;
-- additional delay of 1 cycle for the diagonals up to
-- the main; from then on 2 additional delays per diagonal.
proc source.c (chan cout[],
               value float c[],
               value n, w, p, time)=
  par j=[0 for w]
    var float c.temp[no] :
    var del :
    seq
      ΞĒ
        j < p
          del := (p - 1) + j
        true
          de1 := (2 * j)
      fp.so.d(cout[i], del)
      seg i=[0 for n]
        c.temp[i] := c[(j*n)+i]
      fp.so.x(cout[j], c.temp, n)
      fp.so.d(cout(j), (time-(del+n))) :
-- Source for vector x.
proc source.x (chan xout,
               value float x[],
               value n, time)-
  sea
    fp.so.x(xout, x, n)
    fp.so.d(xout, {time-n}) :
-- Sink for vector x.
-- Initial delay of k * (w + p) cycles.
proc sink.x (chan xin,
```

var float x[], value n, del, time) = var float x.temp[no] ; seq fp.si.d(xin, del) fp.si.x(xin, x, n) : -- Sink for eigenvalue e: one for each pipeline stage. -- It collects e from the last pipeline stage for one cycle -- after an initial delay of k * (w + p) cycles. proc sink.e (chan ein[], var float e, value stage, k, delay, time) = var float e.temp[1] : seq if stage $\langle \rangle$ (k-1) fp.si.d(ein[stage], time) true sec fp.si.d(ein[stage], delay) fp.si.x(ein[stage], e.temp, 1) fp.si.d(ein(stage), (time-(delay+1))) e := e.temp[0] : -- Sink for matrix C proc sink.c (chan cin[], value k, w, time) = par j=[0 for w] fp.si.d(cin{(k*w)+j], time) : -- Main chan c.c[ko*wo], x.c[ko+1], e.c[ko] : var float c[no*wo], xi[no], xo[no], e : var n, w, p, k, delay, time : seg -- input get(n, " size of matrix C ")
get(w, " bandwidth of matrix C ")
get(p, " size of upper semiband ")
fp.get.n(c, (w*n), " of matrix diagonals, upper diag. first ")
fp.get.n(xi, n, " of initial vector x ") get(k, " no of iterations ") delay := (w + p)time := n + (k * delay) -- pipeline par source.c(c.c, c, n, w, p, time) source.x(x.c{0}, x1, n, time) par i=[0 for k] par system(c.c, x.c, e.c, i, w, delay, time) sink.e(e.c, e, i, k, (k*delay), time) sink.c(c.c, k, w, time) sink.x(x.c(k), xo, n, (k * delay), time) -- output fp.put(e, " dominant eigenvalue ") fp.put.n(xo, n, " of corresponding eigenvector ")

.

621 -

```
-- A.4.7
-- Systolic Pipeline for a successive squarings for a
-- (n*n) banded matrix of bandwidth w = p + q - 1, p = q.
-- The output of a pipeline stage is scaled over a given
-- power of 2, so that over/under-flow is avoided.
-- area = SUM ((((2**j)w - ((2**j)-1))**2), j=0,1,2...,s-1
-- time = n + SUM ((2**j)w - ((2**j)-1)), j=0,1,2,.,s-1
-- final bandwidth = (2^{*}s)w - ((2^{*}s)-1),
-- channel count = SUM ((2**j)w - ((2**j)-1)), j=0,1,2...,s
external proc get ( var v, value s[] ) :
external proc fp.get.n ( var float v(), value n, s() ) :
external proc fp.put.n ( value float v(), value n, s() ) :
external proc fp.so.x ( chan xout, value float x(), value time ) :
external proc fp.so.d ( chan xout, value time ) :
external proc fp.si.x ( chan xin, var float x[], value time ) :
external proc fp.si.d ( chan xin, value time ) :
external proc fp.dl ( chan xin, xout, value d, time ) :
external proc fp.br ( chan xin, xlout, x2out, value time ) :
external proc ips.h ( chan ain, bin, cin, aout, bout, cout,
                      value time ) :
def no=10, wi=3, so=3, wo=17, co=34, to=(no+17) :
-- Scaling Cell : it scales its input over a given power of two.
proc sca ( chan cin, cout,
           value float scale,
           value time ) =
  var float c[2] :
  seq
    -- initialise
    par i=[0 for 2]
      c(i) := 0.0
    -- main operation
    seq i=[0 for time]
      seq
        -- 1/0
        par
          cin 7 c[0]
          cout 1 c[1]
         -- calculation
        c[1] := (c[0] / scale) :
 -- Matrix Squaring Block configuration.
 -- Two copies of A are produced, A=A and B=A transposed
 -- Delays for A(B) : q(p)-1, q(p)-2, ..., 1 for the q(p)-1 lower
 -- (upper) diagonals; here p=g=(w+1)/2.
 -- A hex-connected mmm array produces C (=A**2) = A * B.
 -- C is scaled over a power of 2.
 -- Two dummy diagonals for C, one on each side, for uniformity.
 -- s is the stage number; base is used for the mapping function
 -- of the channels.
 proc system ( chan a.io[],
               value float scale,
               value w, base, time ) =
   chan al.c(wo], bl.c(wo),
        a.c[wo*(wo+1)], b.c[wo*(wo+1)], c.c[(wo+1)*(wo+1)] :
   DAL
     -- branch
     par i=[0 for w]
       fp.br ( a.io(base+i], al.c(i), bl.c((w-1)-i), time )
     -- delays
```

```
var p:
    seq
     p := (w + 1) / 2
     par 1-[0 for w]
       if
         i<p
           par
             fp.dl ( al.c[i], a.c[i*(w+1)], 0, time )
             fp.dl ( bl.c[i], b.c[i], 0, time )
         true
            par
             fp.dl ( al.c[i], a.c[i*(w+1)], ((i-p)+1), time )
             fp.dl ( bl.c(i), b.c[i], ((i-p)+1), time )
    -- hex-array
    par i=[0 for w]
     par j=[0 \text{ for } w]
       c.c[(i*(w+1))+(j+(w+2))], time )
    -- i/o links for A,B
    par i=[0 for w]
     par
       fp.si.d ( a.c[(i*(w+1))+w], time )
       fp.si.d ( b.c[(w*w)+i], time )
    -- 1/0 links, scaling for C = A**2
    par i = [0 \text{ for } ((2*w)+1)]
     if
       i = 0
         par
           fp.so.d ( c.c[w], time )
           fp.si.d ( c.c[w], time )
       i <= w
         par
           fp.so.d ( c.c[w-i], time )
           sca ( c.c[w+{i*(w+1)}], a.io[(base+w)+(i-1)].
                 scale, time )
       1 < (2*w)
         par
           fp.so.d ( c.c[(w+1)*(i-w)], time )
           sca ( c.c[(((w+1)*(w+1))-1)-(i-w)],
                 a.io((base+w)+(i-1)), scale, time )
       true
         par
           fp.so.d ( c.c[(w+1)*w], time )
           fp.si.d ( c.c((w+1)*w), time ) :
---
-- Main
var float a[wi*no], c[wo*no] :
chan a.c(col +
var float scale[so+1] :
var n, s, w(so+1), base(so+1), time :
seg
 -- getdata
 get ( n, " matrix order " )
 get ( w[0], " bandwidth for A : w = 2p - 1 " )
 fp.get.n ( a, (w[0]*n), " matrix A upper diagonal first " )
 get ( s, " squarings " )
  fp.get.n ( scale, (s+1), " scaling factors " )
 base[0] := 0
  seq i=[1 for s]
   seq
     w(i) := ((2 + w(i-1)) - 1)
     base[i] := base[i-1] + w[i-1]
  time := n + (base[s] + s)
 -- driver
```

- 622 -

par -- input of matrix A par i=[0 for w[0]] var float atemp[no] : seq seq j={0 for n} atemp[j] := a[(i*n)+j] fp.so.x (a.c[i], atemp, n) fp.so.d (a.c(i), (time-n)) -- output of matrix C = A**2**s scaled nar i=10 for w[s]] var float ctemp[no] : seq fp.si.d (a.c[base[s]+i], (time-n)) fp.si.x (a.c[base[s]+i], ctemp, n) seg j=[0 for n] c[(i*n)+j] := ctemp[j] -- pipeline of a blocks par i=[0 for s] system (a.c, scale[i], w[i], base[i], time) -- putdata fp.put.n (c, (w[s]*n), " C sequence, upper diagonal first ")

-- A.4.8 ------ Systolic Array for Power Method. -- Time Expansion. -- Calculation of the dominant eigenvalue e and corresponding -- eigenvector u of a (m*m) matrix A.Form of the method : $y[i+1] = A^*x[i]; x[i+1] = y[i+1]/m(y[i+1]),$ -- where m(x) is the first non-zero element of x; $x(0)=(1 \ 1 \ .. \ 1)$ -- e = m(y[k]), u/m(u) = x[k], where k is the no of iterations. -- Area = (m+1) IPS cells and (m*m) memory. -- Time = (2*m)*(k+1)+1 cycles. -external proc get (var v, value s[)) : external proc fp.put (value float v, value s[]) : external proc fp.get.n (var float v[], value n, s[]) : external proc fp.put.n (value float v(), value n, s()) : external proc fp.so.x (chan xout, value float x[], value time) : external proc fp.si.x (chan xin, var float x[], value time) : external proc fp.so.d (chan xout, value time) : external proc fp.si.d (chan xin, value time) : def mo=10, ko=10, to=(((2*mo)*(ko+1))+1) : ---- Inner Product Step Cell : Multiply - Accumulate -- When the last accumulation occurs the result is sent as feedback -- and the accumulator is reset for next cycle of calculations. proc ips (chan win, nin, nout, sin, sout, value cycle, reset, time) = var float w[2], n[2], s[2], acc : seq -- initialise par acc := 0.0 par i=[0 for 2] par w[i] := 0.0 n(i) := 0.0s[i] := 0.0 -- main operation seq i=[0 for time] seq -- i/o par win 7 w[0] nin 7 ní01 sin 7 s[0] nout ! n[1] sout 1 s[1] -- calculation acc := acc + (w[1] + n[0]) par w[1] := w[0]n[1] := s[0] s[1] := n[0]-- control if (i \ cycle) = reset seq n[1] := accacc := 0.0 : -- Multiplexer -- For each mpy cycle it normalises the feedback vector.

1

σ

с С

-- The initial vector enters the array through s.

```
-- Output: the normalised vector and the eigenvalue.
proc mux ( chan sin, sout, fin, fout, eout,
           value cycle, reset, time } -
  var float s, f[2], first :
  var first.found :
  seq
    initialise
    par
      par i=[0 for 2]
        f[i] := 0.0
      s := 0.0
      first := 1.0
      first.found := false
    -- main operation
    seq i=[0 for time]
      seq
         -_ i/o
         par
           sin 7 s
           fin 7 f[0]
           sout I f[1]
           fout 1 f[1]
           eout 1 first
         -- control
         if
           (i \ cycle) = reset
             par
               first.found := false
               first := 1.0
         if
           (not first.found) and (f[0] <> 0.0)
             par
               first.found := true
                first := f[0]
         f[1] := (f[0] + s) / first :
 -- Source for initial vector: operates every two steps.
 -- For the first mpy cycle it sents the initial vector, and
 -- then it sends dummy elements.
 proc so.n ( chan nout,
              value float x[],
              value cycle, time ) =
   var float temp[2*mo] :
   seq
     seg j=[0 for cycle]
          (j \setminus 2) = 0
            temp[j] := x[j/2]
          true
            temp[j] := 0.0
      fp.so.x( nout, temp, cycle )
      fp.so.d( nout, (time-cycle) ) :
  -- Source for a row i of matrix A.
  -- Operates after an initial delay of i and every two steps,
  -- for k mpy cycles; then it sends dummy elements.
  proc so.w ( chan wout[],
              value float a[],
              value k, m, cycle, time ) =
    par i=[0 for m]
      var float temp[2*mo] :
      seq
```

```
seg j=[0 for cycle]
        ίf
          (j \setminus 2) = 0
            temp[j] := a[(i*m)+(j/2)]
          true
            temp[j] := 0.0
      fp.so.d( wout[1], 1)
      seq j [0 for k]
        fp.so.x( wout[i], temp, cycle )
      fp.so.d( wout(i), (time - ((k*cycle) + i)) ) :
-- Sink for results on the west side
-- Collects the eigenvalue in the first step of the last
-- mpy cycle, and the eigenvector in the last mpy cycle
-- every two steps.
proc si.n ( chan nin, ein,
            var float x[], e,
            value m, delay, cycle, time ) =
  par
    var float x.temp[2*mo] :
    seg
      fp.si.d( nin, delay )
      fp.si.x( nin, x.temp, cycle )
      seq j=[0 for cycle]
        if
          (j \setminus 2) = 0
            x[j/2] := x.temp[j]
    var float e.temp[1] :
     seq
       fp.si.d( ein, delay )
       fp.si.x( ein, e.temp, 1 )
       fp.si.d( ein, (cycle-1) )
       e := e.temp[0] :
-- Array Configuration
-- Linear array of m IPS cells; north side has a Mux cell.
-- m west sources for matrix A; north source-sink for
-- initial and final vector; dummy source-sink in south.
-- Cycle is the time for one matrix-matrix multiplication.
proc system ( value float a[], xi[],
               value m, k,
               var float xo[], e ) =
   chan h.c(mo], u.c(mo+2], d.c(mo+2), e.c :
   var time, cycle :
   seq
     cycle := (2 * m)
     time := (cycle * (k + 1)) + 1
     par
       --- ips cells
       par i=[0 for m]
         var reset :
         seq
            reset := ((cycle + (i - 1)) \setminus cycle)
           ips ( h.c[i], d.c[i+1], u.c[i+1], u.c[i+2], d.c[i+2],
                 cycle, reset, time )
       -- mux
       mux { d.c[0], u.c[0], u.c[1], d.c[1], e.c, cycle, 0, time }
       -- sources, sinks
       so.w ( h.c, a, k, m, cycle, time )
       so.n ( d.c[0], xi, cycle, time )
       si.n { u.c[0], e.c, xo, e, m, (time-cycle), cycle, time )
        fp.so.d( u.c(m+1), time )
        fp.si.d( d.c[m+1], time ) :
```

624 -

-- Main

var float a[mo*mo], xi[mo], xo[mo], e : var m, k z

seq

get (m, " size of matrix A ") fp.get.n (a, (m * m), " matrix A row-wise ") fp.get.n (xi, m, " initial vector ") get (k, " number of iterations ") system (a, xi, m, k, xo, e)

fp.put(e, " eigenvalue ")

fp.put.n(xo, m, " eigenvector ")

----- A.4.9

-- Systolic Array for the Matrix Squaring Method.

-- Time Expansion.

-- A full mmm reusable array is used, with source-sink drivers

-- configured for the calculation of a series of Successive

-- Matrix Squarings followed by a series of mmm's (usually 1).

-- The result of a matrix squaring is scaled over a power of 2.

-- area = m**2, where m is the order of the matrix;

-- time = (2*m)*(s+p+2)+1, where s,p is the number of squarings -- and mmm's respectively.

external proc get (var v. value s()) :

external proc fp.get.n (var float v(), value n, s()) :

external proc fp.put.n (value float v[], value n, s[]) :

external proc fp.so.x.t (chan xout, value float x(), value t(),time) :

external proc fp.si.x (chan xin, var float x[], value time) :

external proc fp.so.d (chan xout, value time) :

external proc fp.si.d (chan xin, value time) :

external proc fp.float (value fix, var float flt) ;

external proc ips.r (chan ein, eout, win, wout, nin, nout, sin, sout, value cycle, reset, time) :

t

δ

N

ຫັ

t

def mo=10, so=10, po=10, to=(((2*mo)*(so+(po+2)))+1), co=(mo+2): * ----

-- Multiplexer - Scaling Cell. -- It adds the source input to the array feedback if fb -- is true; otherwise it accepts the source input.

-- The final array input is scaled over a given factor. -----

proc mux.s (chan sin, sout, fin, fout, value float scale[], value cycle, reset, time) = var float s(2), f(2):

var j, fb :

seq -- initialise

par par i=[0 for 2]

par

1 :- 0

-- main operation

seq i=[0 for time]

```
par
```

s[1] := f[1]

sin 7 s[0]: fb

fin ? f[0]

sout ! s[1]

fout [f[1] -- control ìf (i \ cycle) = reset j := (j + 1)

```
ί£
  fb = 1
    seq
     f[1] := ({f[0] + s[0]} / scale[j])
```

```
s[i] := 0.0
    f[i] := 0.0
fb := 0
```

seq -- i/o

> true seq

```
f(1) := f(0)
              s[1] := (f(0) / scale[j]) :
-- Source for row i of input matrices, morth side.
-- Operates after an initial delay equal to i and every two steps.
-- For j=0 mpy cycle it sents the ith column of I. Then it sents
-- dummy elements; fb is on.
___
proc so.n ( chan nout,
            value i, cycle, time ) -
  var float temp[to] :
  var fbitol :
  seq
    -- input vectors
 · par
      seq
        seq j=[0 for time]
          temp[j] := 0.0
        temp[i+(2*i)] := 1.0
      seg i=[0 for time]
        fb[j] := 1
    -- source
    fp.so.x.t( nout, temp, fb, time ) :
-- Source for a row i of input matrices, west side.
-- Operates after an initial delay of i and every two steps
-- For j=0 and s+1,s+2,...,s+p mpy cycles it pumps a row of A.
-- Otherwise it sends dummy elements; fb is on until s+1 mpy.
proc so.w ( chan wout,
             value float a[],
             value i, s, p, m, cycle, time ) =
  var float a.temp[to], temp[to] :
  var fb(to) :
  seq
     -- input vectors
    par
      seq
         seg ]=[0 for time]
           temp[1] := 0.0
         seq j=[0 for cycle]
           ίf΄
             (1 \setminus 2) = 0
               a.temp[j] := a[(i*m)+(j/2)]
             true
               a.temp[j] := 0.0
         seq j=[0 for cycle]
           temp[i+j] := a.temp[j]
         seg l=((s+1) for p)
           seg j=[0 for cycle]
             temp[(i+(l*cycle))+j] := a.temp[j]
       sea
         seq j=[0 for time]
           fb[i] := 0
         seq j=[0 for (i+((s+1)*cycle))]
           fb[j] := 1
     -- source
     fp.so.x.t( wout, temp, fb, time ) :
 -- Sink for results on the west side
 -- Collects the first column of the result in the last two
 -- mpy cycles.
 proc si.w ( chan rin,
             var float x1[], x2[],
             value i, m, cycle, time ) -
```

```
var float x.temp[to] :
 var delay :
 seg
   fp.si.x( rin, x.temp, time )
   delay := ((time + i) - (2 * cycle))
   xl[i] := x.temp[delay]
   x2[i] := x.temp[delay+cycle] :
-- Array Configuration
-- Square array of (m*m) IPS cells; west and north side have an
-- additional row of m Mux's; the array is surrounded with
-- 4m sources - dummy in east and south sides, and
-- 4m sinks - dummy in east, north and south sides.
-- Cycle is the time for one matrix-matrix multiplication.
proc system ( value float a[], scale[],
              value m, s, p,
              var float x1[], x2[]  =
  chan r.c[mo*co], l.c[mo*co], u.c[mo*co], d.c[mo*co] :
  var time, cycle :
  seq
    cvcle := (2 * m)
    time := (cycle * (s + (p + 2))) + 1
    par
      -- ips cells
      par l=[0 for m]
        par j=[0 for m]
          var di, dj, reset :
          seq
            di := (i + (m + 2)) + (i + 1)
            dj := (j + (m + 2)) + (1 + 1)
             reset := ((cycle + ((i + j) - 1)) \ cycle)
             ips.r ( l.c[di+1], r.c[di+1], r.c[di], l.c[di],
                     d.c[dj], u.c[dj], u.c[dj+1], d.c[dj+1],
                     cycle, reset, time }
      -- mux's
      par i=[0 for m]
        var wn, reset :
         seq.
          wn := (i + (m + 2))
           reset := ((cycle + i) \setminus cycle)
          par
             mux.s ( r.c[wn], l.c[wn], l.c[wn+1], r.c[wn+1],
                     scale, cycle, reset, time )
             mux.s ( d.c[wn], u.c[wn), u.c[wn+1], d.c[wn+1],
                     scale, cycle, reset, time )
       -- sources, sinks
      par i=[0 \text{ for } m]
         var wn, es :
         seq
           wn := (i * (m + 2))
           es := wn + (m + 1)
           par
             so.n ( d.c[wn], i, cycle, time )
             so.w ( r.c[wn], a, i, s, p, m, cycle, time )
             si.w ( l.c[wn], x1, x2, 1, m, cycle, time )
             fp.si.d( u.c(wn), time )
             fp.so.d( l.c[es], time
             fp.so.d( u.c[es], time
             fp.si.d( r.c(es), time
             fp.si.d( d.c(es), time ) :
-- Main
var float a[mo*mo], x1[mo], x2[mo], scale[(so+po)+4] :
var m, s, p :
```

.

626 .

seq

get (m, " size of matrix A ") fp.get.n (a, (m * m), " matrix A row-wise ") get ('s, " matrix squaring steps ")
get (p, " multiplication steps ") fp.get.n (scale, ((s+p)+4), " scaling factors ") system (a, scale, m, s, p, x1, x2) fp.put.n(x1, m, " result-1 ")
fp.put.n(x2, m, " result-2 ")

-- A.4.10 -----

-- Systolic Array for Matrix Exponential.

-- Time Expansion. -- A full mmm reusable array is used, with source-sink drivers -- configured for the calculation of a Matrix Exponential, which is -- analysed to the calculation of a Matrix Polynomial followed by --- a series of Successive Matrix Squarings. These two different -- computations are performed on the same array by changing the -- feedback-multiplexing operations accordingly. The source drivers -- are assumed to have some memory-calculation capabilities. -- area - m**2, where m is the order of the matrix; -- time - (2*m)* (k+s+2)+M-1where k, are the number of stages for -- the matrix polynomial, and the matrix squaring respectively. ---external proc get (var v, value s[]) : external proc fp.get.n (var float v(], value n, s(]) : external proc fp.put.n (value float v(), value n, s()) : external proc fp.so.x.t (chan xout, value float x[], value t[], time) : external proc fp.si.x (chan xin, var float x[], value time) : external proc fp.so.d (chan xout, value time) : external proc fp.si.d (chan xin, value time) : external proc fp.float (value fix, var float flt) : external proc ips.r (chan ein, eout, win, wout, nin, nout, sin, sout, value cycle, reset, time) : external proc mux.r (chan sin, sout, fin, fout, value time) : def mo=10, ko=10, so=10, to=((2*mo)*((ko+so)+2)), co=(mo+2): ---~--- Source for row i of input matrices, west side. -- Operates after an initial delay equal to i and every two steps. -- For j = 0,1,...k-1,k mpy cycles it sents the ith row of -- (1 / (k - j))) * I. Then it sents dummy elements; fb is on. proc so.w (chan wout, value i, k, cycle, time) = var float temp{to} : var fact, loc, pre, post, fb[to] : seq -- initialise par seq fact := 1seg j=[1 for k] fact := (fact * j) loc := i pre := (2 * i) post := (cycle - pre) -- input vectors par seg seg j=[0 for time] temp[j] := 0.0seq j=[0 for (k+1)] var float ffact : seq loc := (loc + pre) fp.float (fact, ffact) temp[loc] := (1.0 / ffact) loc := (loc + post) if 1 < k

fact := (fact / (k - j))

```
seg j=[0 for time]
       fb(j) := 1
    -- source
    fp.so.x.t( wout, temp, fb, time ) :
-- Source for a row i of input matrices, north side.
-- Operates after an initial delay of i and every two steps
-- For j=0,1,...,k-1 mpy cycles it pumps a column of matrix
-- (A / 2**s) into the array. For the kth mpy it sends matrix
-- I; from then on dummy elements; fb is on after the kth mpy.
----
proc so.n ( chan nout,
            value float a[].
            value i, k, s, m, cycle, time ) -
  var float a.temp[to], temp[to], div :
  var pre, fb[to] :
  seq
    -- initialise
    par
      pre := (2 * i)
      seq
        div := 1.0
        seg 1=[1 for s]
          div = (div + 2.0)
    -- input vectors
    par
      seq
         seq j=[0 for time]
          temp[j] := 0.0
         seq j=[0 for cycle]
          if
            (j \ 2) = 0
               a.temp[j] := (a[((j/2)*m)+i) / div)
             true
               a.temp[j] := 0.0
         seg j=[0 for k]
           seq 1=[0 for cycle]
             temp((j*cycle)+(l+i)) := a.temp[1]
         temp[(k*cycle)+(pre+i)] := 1.0
       seq
         seq j=[0 for time]
           fb(j) := 1
         seg j=[0 for (((k+1)*cycle)+(i-1))]
           fb[j] := 0
     -- source
     fp.so.x.t( nout, temp, fb, time ) :
 -- Sink for results on the west side
 -- Collects results in the last mpy cycle every two steps
 proc si.w ( chan rin,
             var float x[],
             value i, m, cycle, time ) =
   var float x.temp[to] :
   var delay :
   seq
     fp.si.x( rin, x.temp, time )
     delay := ( time - (cycle -1) ) - ((m-1) - i)
     seq j=[delay for cycle]
       -1€
         ((j - delay) \setminus 2) = 0
           x[(i*m)+((j-delay)/2)] := x.temp[j] :
 -- Array Configuration
 -- Square array of (m*m) IPS cells; west and north side have an
  -- additional row of m Mux's; the array is surrounded with
```

-- 4m sources - dummy in east and south sides, and -- 4m sinks - dummy in east, north and south sides. -- Cycle is the time for one matrix-matrix multiplication. proc system (value float a[], value m, k, s, var float x() > = chan r.c[mo*co], l.c[mo*co], u.c[mo*co], d.c[mo*co] : var time, cycle : seq cycle := (2 * m) time := (cycle * (k + s + 2)) + (m-1) par -- ips cells par i=[0 for m] par j=(0 for m) var di, dj, reset : seq di := (i + (m + 2)) + (j + 1)dj = (j + (m + 2)) + (1 + 1)reset := ((cycle + ((i + j) - 1)) \ cycle} ips.r (1.c[di+1], r.c[di+1], r.c[di], 1.c[di], d.c(dj), u.c(dj), u.c(dj+1), d.c(dj+1), cycle, reset, time) -- mux's par i=[0 for m] var wn : seq wn := (i + (m + 2))par mux.r (r.c[wn], l.c[wn], l.c[wn+1], r.c[wn+1], time) mux.r (d.c[wn], u.c[wn], u.c[wn+1], d.c[wn+1], time) -- sources, sinks par i=[0 for m] var wh, es : seq $w_{1} := (i + (m + 2))$ es := wn + (m + 1)par so.w (r.c[wn], i, k, cycle, time) so.n (d.c[wn], a, i, k, s, m, cycle, time) si.w (l.c[wn], x, i, m, cycle, time) fp.si.d(u.c(wn), time) fp.so.d(l.c[es], time fp.so.d(u.c[es], time fp.si.d(r.c(es), time) fp.si.d(d.c[es], time) : ------ Main ---var float a[mo*mo], x[to*mo] : var m, k, 6 : seq get (m, " size of matrix A ") fp.get.n (a, (m * m), " matrix A row-wise ") get (k, " number of Taylor series terms ")
get (s, " scaling-squaring factor ") system (a, m, k, s, x) fp.put.n(x, (m * m), " result ")

- 628 -

-- 8.4.11 ___ -- Systolic Pipeline for calculation of matrix polynomial --p(A) = b0*I + b1*A + b2*A**2 + ... + bs*A**s, where A is-- (n*n) banded matrix of bandwidth w = p + q - 1, p = q, -- and b0, b1, ..., bs given matrices.Horner's scheme used. -- ips area = 1 + SUM (w * (j*w - (j-1))), j=0,1,2,..,s-1 -- total area = s + ((s-1)*w-(s-2))**2, -- time = n + s + ((s-1)*w - (s-2)), -- final bandwidth - s*w - (s-1), -- channel count for result = SUM (j*w - (j-1)), j=0,1,...,s. -- The area in each stage is equal to the max area required; -- only part of it is used and the rest is replaced by delays; -- optimum time = n + w + SUM(j*w - (j-1)), j=1,2,...,s-1. external proc get { var v, value s[]) : external proc fp.get.n (var float v[], value n, s[]) : external proc fp.put.n (value float v(], value n, s(]) : external proc fp.so.x (chan xout, value float x{}, value time) : external proc fp.so.d (chan xout, value time) : external proc fp.si.x (chan xin, var float x[], value time) : external proc fp.si.d (chan xin, value time) : external proc fp.lk (chan xin, xout, value time) : external proc fp.dl (chan xin, xout, value d, time) : external proc fp.br (chan xin, xlout, x2out, value time) : external proc ips.h (chan ain, bin, cin, aout, bout, cout, value time) : def no=10, so=5, wi=3, wo=11, co=36, to=(no+(so*wo)) : ----- Matrix Polynomial Block configuration. -- A hex-connected mmm array produces C = X + A * B, with -- X = bs*I, where s is the stage number. A travels along the -- pipeline and a copy of it branches in each stage.B is the -- result of the previous stage.wa = 2pa-1, win=2pin-1 are the -- bandwidths of A, B; w = 2p-1 is the max bandwidth allowed -- by the hex-array. A and B are aligned in the center of the -- input channels of the hex-array; similarly only wa+win-1 -- outputs are significant. Two dummy diagonals for C, one on -- each side, for uniformity base is used for the mapping -- function of the channels carrying the result. proc system (chan ain[], cin[], aout[], cout[], value float x[], value s, wa, win, w, base, time) = chan al.c[2*wo], a.c[wo*(wo+1)], b.c[wo*(wo+1)], c.c((wo+1)*(wo+1)] : Dar -- i/o, branch, delays for A par i=[0 for wa] par fp.br { ain[(s*wa)+i], al.c[i], al.c[i+wa], time } fp.dl (al.c[i], aout[((s+1)*wa)+i], w, time) -- hex-array par i=[0 for w] par j=[0 for w] ips.h (a.c[(i*(w+1))+j], b.c[(i*w)+j], c.c[(i*(w+1))+j], a.c[(i*(w+1))+(j+1)], b.c[(i*w)+(w+j)], c.c[(i*(w+1))+(j+(w+2))], time) -- i/o. input delays for A.B var da, din, pa, pin : 6ea da := (w - wa) / 2pa := (wa + 1) / 2din $\tau = (w - win) / 2$

pin := (win + 1) / 2par i=[0 for w] par if (i >= da) and (i < (wa + da))if $i \in \{pa + da\}$ fp.dl (al.c[(i+wa)-da], a.c[i*(w+1)], 0, time) true fp.dl (al.c((i+wa)-da], a.c(i*(w+1)), ((i-da)-(pa-1)), time) true fp.so.d (a.c[i*(w+1)], time) if $(i \ge din)$ and (i < (win + din))if i < (pin + din)fp.dl (cin(base+(i-din)), b.c(i), 0, time) true fp.dl (cin[base+(i-din)], b.c[i], ((i-din)-(pin-1)), time) true fp.so.d (b.c[i], time) fp.si.d (a.c[(i*(w+1))+w], time) fp.si.d (b.c[(w*w)+i], time) -- 1/o for X.C var wout, dout : seg wout := (wa + win) - 1dout := (((2 * w) + 1) - wout) / 2par i=[0 for ((2*w)+1)]par if i<w fp.so.d (c.c[w-i], time) 1 - - fp.so.x (c.c[w-i], x, time) true fp.so.d { c.c((w+1)*(i-w)}, time) ١f i < dout fp.si.d { c.c[w+(i*(w+1))], time) 1 (w fp.lk { c.c[w+(i*(w+1))], cout[(base+win)+(i-dout)], time) i < (wout + dout) fp.lk (c.c[(((w+1)*(w+1))-1)-(i-w)), cout[(base+win)+(i-dout)], time) true fp.si.d (c.c[(((w+1)*(w+1))-1)-(i-w)], time) : -- Main var float a(wi*no), c(wo*to), x(so+1) : chan a.c[wi*(so+1)], c.c[wo*so] : var n, s, wa, w, wc, base, delay, time : seg -- getdata get (n, " matrix order ") get (wa, " bandwidth for A : w = 2p - 1 ") fp.get.n { a, (wa*n), " matrix A upper diagonal first ") get (s, " stages ") fp.get.n (x, (s+1), " polynomial coeffs ") base := 0 seq 1=[0 for s]

629

Т

```
base := base + ((i * wa) - (i - 1))
w := ((s - 1) + wa) - (s - 2)
i€
  w < wa
    w :- wa
wc := (s + wa) - (s - 1)
delay := (6 * W)
time := n + delay
-- driver
Dac
  -- i/o of matrix A
  par i=[0 for wa]
    par
      var float atemp[to] :
      seq
        seg j=[0 for time]
          atemp[j] := 0.0
        seq j=[0 for n]
         atemp[j] := a[(i*n)+j]
        fp.so.x ( a.c(i), atemp, time )
      fp.si.d (a.c[(s*wa)+i], time )
  -- i/o for C=p(A)
  par
    var float xtemp[to] :
     seq
      seg j=[0 for time]
        xtemp(j] := x[s]
       fp.so.x ( c.c[0], xtemp, time )
     par i=[0 for wc]
       var float ctemp[to] :
       seg
        fp.si.x ( c.c(base+i), ctemp, time )
         seg j={0 for n}
          c[(i*n)+j] := ctemp[delay+j]
   -- pipeline of s blocks
   par i=[0 for s]
     var float xtemp[to] :
     var win, base :
     seq
       seg j=[0 for time]
        xtemp[j] := x[s-(i+1)]
       base := 0
       seq j={0 for i}
         base := base + ((j * wa) - (j - 1))
       win := ((i * wa) - (i - 1))
       system ( a.c., c.c., a.c., c.c.,
                xtemp, i, wa, win, w, base, time )
 -- outdata
 fp.put.n ( c, (wc*n), " C sequence, upper diagonal first " )
```

.

```
-- A.5.1
-- Soft-Systolic simulation for an Optical Processor
-- performing Matrix-Vector Multiplication ( Ax = y )
-- for a banded (n*n) matrix with band w = p + q - 1,
-- using Inner Product.
-- Computation time = n + w - 1; emitter and modulator
-- size = w; detector size = 1.
-- i/o routines
external proc get ( var v, value s[] ) :
external proc get.n ( var v[], value n, s[] ) :
external proc put.n ( value v(), n, s() ) :
-- optical-systolic routines
external proc emit ( chan datain, dataout, beamout[],
                     value start, num, step, time)
external proc modl ( chan datain, dataout, beamin[],beamout[].
                     value start, num, step, time):
external proc shda ( chan datain, dataout, beamin[],
                     value start, num, step, time):
external proc driver.em ( chan xout, value x[], time ) :
external proc outp.em ( chan xin, value time ) :
external proc inp.shda ( chan xout, value time ) :
external proc outp.shda ( chan xin, var x[], value time ) :
---
-- max matrix size, max bandwidth, max time.
def no = 10, wo = 10, to = (no + wo) :
-- optical processor configuration.
proc system ( var y[],
              value a[], x[], w, time )=
  -- data communication.
  chan a.c[2*wo], x.c[wo+1], y.c[2], em.c[wo], md.c[wo] :
  -- help occam get the types right.
  var temp :
  seq
    temp := x[0]
    temp := y(0)
    par
       -- w-pixel emitter for vector x
      par
        driver.em ( x.c[0], x, time )
        par i=[0 for w]
          emit ( x.c[i], x.c[i+1], em.c, i, 1,1,time }
        outp.em ( x.c[w], time )
       -- w-pixel modulator for matrix A
       par i=[0 for w]
        --- diagonal of matrix A
        var temp.a[to] :
        seq
           par j=[0 for time]
             temp.a[j] := a[(i*time)+j]
           par
             driver.em ( a.c[i], temp.a, time )
             modl ( a.c[i], a.c[i+w], em.c , md.c,i,1,i,time )
             outp.em ( a.c[i+w], time )
       -- 1-pixel shift detector array for result vector y.
       par
         inp.shda ( y.c[0], time )
         shda ( y.c(0), y.c(1), md.c, 0, w,s,time )
         outp.shda ( y.c[1], y, time ) :
 -- main program.
 -- input and output data for calculation Ax - y.
```

```
var a[wo*to], x[to], y[to],
-- actual matrix size; bandwidth; time.
    n, w, time :
seq
    get { n, " size of matrix A " }
    get { w, " bandwidth of matrix A " }
    time := (n + w) - 1
    get.n { a, (w*time}, " matrix A data seq: upper diag first " }
    get.n { x, time, " vector x data seq " }
    system { y, a, x, w, time }
    put.n { y, time, " vector y data seq " }
```

Т

```
-- A.5.2
-- Soft-Systolic simulation for an Optical Processor
-- performing Matrix-Vector Multiplication ( Ax - y )
-- for a banded (n*n) matrix with band w = p + q - 1,
-- using Outer Product.
-- Computation time = n + p; emitter size = 1; modulator
-- and detector size = w.
-- i/o routines.
external proc get ( var v, value s[] ) :
external proc get.n ( var v(), value n, s() ) ;
external proc put.n ( value v[], n, s[] ) :
-- optical-systolic routines.
external proc emit ( chan datain, dataout, beamout[],
                      value start, num, stop, time):
external proc modl ( chan datain, dataout, beamin[],beamout[],
                      value start, num, step, time ):
external proc shda ( chan datain, dataout, beamin[],
                      value start, num, step, time);
external proc driver.em ( chan xout, value x[], time ) :
external proc outp.em ( chan xin, value time ) :
external proc inp.shda ( chan xout, value time ) :
external proc outp.shda ( chan xin, var x[], value time ) :
---
-- max matrix size, max bandwidth, max time.
def no = 10, wo = 10, to = (no + wo) :
----
-- optical processor configuration.
proc system ( var y[],
              value a[], x[], w, time )=
  -- data communication.
  chan a.c[2*wo], x.c[2], y.c[wo+1], em.c[wo], md.c[wo] :
  -- help occam get the types right.
  var temp :
  $eq
    temp := x[0]
    temp := y(0)
    par
      -- 1-pixel emitter for vector x
      par
        driver.em ( x.c(0), x, time )
        emit ( x.c[0], x.c[1], em.c, 0, w,4,time )
        outp.em ( x.c[1], time )
      -- w-pixel modulator for matrix A
      par i=[0 \text{ for } w]
        -- diagonal of matrix A
        var temp.a(to) :
        seg
          par j=[0 for time]
            temp.a[j] := a[(i*time)+j]
          par
            driver.em ( a.c[i], temp.a, time )
            modl ( a.c[i], a.c[i+w], em.c , md.c, i,1,i,time )
            outp.em ( a.c[i+w], time )
      -- w-pixel shift detector array for result vector y.
      par
        inp.shda ( y.c[0], time )
        par i=[0 \text{ for } w]
          shda ( y.c[i], y.c[i+1], md.c, i, 1,1,time )
        outp.shda ( y.c[w], y, time ) ;
-- main program.
```

-- input and output data for calculation Ax = y. -var a[wo*to], x[to], y[to], -- actual matrix size; bandwidth; upper semiband; time. n, w, p, time : seq get (n, " size of matrix A ") get (w, " bandwidth of matrix A ") get (p, " upper semiband of matrix A ") time := (n + p) get.n (a, (w*time), " matrix A data seq: lowest diag first ") get.n (x, time, " vector x data seq ") system (y, a, x, w, time) put.n (y, time, " vector y data seq ")

```
-- A.S.3
-- Soft-Systolic simulation for an Optical Processor
-- performing Matrix-Matrix Multiplication of two banded
-- (n*n) matrices A, B with bands w(a) = w(b) = w,
-- w = p + q - 1, producing matrix C with band w(c) =
-2 + w = 1, p(c) = 2 + p - 1, q(c) = 2 + q - 1
-- using Inner Product.
-- Computation time = n + p; emitter size = w; modulator
-- size = (w + w); detector size = (2 + w - 1).
-- i/o routines.
external proc get ( var v, value s[] ) :
external proc get.n ( var v[], value n, s[] ) :
external proc put.n ( value v(), n, s() ) :
-- optical-systolic routines.
external proc emit ( chan datain, dataout, beamout[],
                     value start, num, step, time ) :
external proc modl ( chan datain, dataout, beamin[], beamout[],
                     value start, num, step, time ) :
external proc shda ( chan datain, dataout, beamin[],
                     value start, num, step, time ) :
external proc driver.em ( chan xout, value x[], time ) :
external proc outp.em ( chan xin, value time ) :
external proc inp.shda ( chan xout, value time ) :
external proc outp.shda ( chan xin, var x(), value time ) :
-- max matrix size, max bandwidth, max time.
def no = 10, wo = 10, to = (no + wo) :
-- optical processor configuration.
proc system ( var c[].
              value a[], b[], w, time )=
  -- data communication.
   chan a.c[2*wo], b.c[wo*(wo+1)], c.c[2*((2*wo)-1)],
        em.c[wo*wo], md.c(wo*wo) :
   -- help occam get the types right.
   var temp.
   -- bandwidth of resulting matrix C
      WC :
   seq
    temp := a[0]
     temp := b[0]
     temp := c[0]
     wc = ((2 + w) - 1)
     par
       -- w-pixel emitter for matrix A
       par i=[0 for w]
         -- diagonal of matrix A
         var temp.a[to] :
         seq
           par j-[0 for time]
             temp.a[j] := a[(i*time)+j]
           par
             driver.em ( a.c[i], temp.a, time )
             emit ( a.c[i], a.c[i+w], em.c, (i*w), w, 1, time )
             outp.em ( a.c[i+w], time )
       -- w*w-pixel modulator for matrix B
       par i=[0 for w]
         -- diagonal of matrix B
         var temp.b[to] :
         seq
           par j={0 for time}
```

```
temp.b[j] := b[(i*time)+j]
          par
           driver.em ( b.c[i], temp.b, time )
           par j=[0 for w]
             mod1 { b.c[(i*w)+j], b.c[(i*w)+(j+w)], em.c, md.c,
                     ((i*w)+j), 1, 1, time )
            outp.em ( b.c((w*w)+i), time )
     -- (2*w-1)-pixel shift detector array for result matrix C
     par i=[0 for wc]
        -- diagonal of matrix C
        var temp.c[to] :
        seq
          par
            inp.shda ( c.c[i], time )
            ₫Ĕ
              ićw
                shda ( c.c[i], c.c[i+wc], md.c, i, (i+1), (w-1), time )
              true
                shda { c.c[i], c.c[i+wc], md.c, ((i*w)-((w-1)*(w-1))),
                       (wc-i), (w-1), time )
            outp.shda ( c.c(i+wc), temp.c, time )
          par j=[0 for time]
            c[(i*time)+j] := temp.c[j] :
  main : p-1 cycles delay for the input of A so that B reaches
---
          the appropriate position.
---
-- input and output data for calculation AB = C.
var a[wo*to], b[wo*to], c[((2*wo)-1)*to],
-- actual matrix size; bandwidths; upper semiband, time.
    n, w, p, time :
seq
 get ( n, " size of problem " )
get ( w, " bandwidth of input matrices ")
  get ( p, " upper semiband " )
  time := (n + p)
  get.n ( a, (witime), " matrix A data seg " )
  get.n ( b, (w*time), " matrix B data seq " )
  system ( c, a, b, w, time )
  put.n ( c, (((2*w)-1)*time), " matrix C data seq " )
```

Т

-- A.5.4 ------- Soft-Systolic simulation for an Optical Processor -- performing Matrix-Matrix Multiplication of two banded \rightarrow (n*n) matrices A, B with bands w[a] = w[b] = w, -- w = p + q - 1, producing matrix C with band w[c] = -2 + w - 1, p[c] = 2 + p - 1, q[c] = 2 + q - 1-- using Outer Product. -- Computation time = n + w - 1; emitter size = w; modulator -- size = w; detector size = (w * w). -- i/o routines. external proc get (var v, value s[]) : external proc get.n (var v[], value n, s[]) : external proc put.n (value v[], n, s[]) : -- optical-systolic routines. external proc emit (chan datain, dataout, beamout[], value start, num, step, time) : external proc modl (chan datain, dataout, beamin[], beamout[], value start, num, step, time) : external proc shda (chan datain, dataout, beamin[), value start, num, step, time) : external proc driver.em (chan xout, value x[], time) : external proc outp.em (chan xin, value time) : external proc inp.shda (chan xout, value time) : external proc outp.shda (chan xin, var x[), value time) : -- max matrix size, max bandwidth, max time. def no = 10, wo = 10, to = (no + wo) : -- optical processor configuration. proc system (var c[], value a[], b[], w, time)= -- data communication. chan a.c[2*wo], b.c[2*wo], c.c[(wo+1)*(wo+1)], em.c[wo*wo], md.c[wo*wo] : -- help occam get the types right. var temp : pea temp := a[0] temp := b[0] temp := c[0]par -- w-pixel emitter for matrix A par i=[0 for w] -- diagonal of matrix A var temp.a[to] : sea par j=[0 for time] temp.a[j] := a[(i*time)+j]par driver.em (a.c[i], temp.a, time) emit (a.c[i], a.c[i+w], em.c, (w*i), w, 1, time) outp.em (a.c[i+w], time) -- w-pixel modulator for matrix B par i=[0 for w] -- diagonal of matrix B var temp.b[to] : seq par j=[0 for time] temp.b[j] := b[(i*time)+j]par driver.em (b.c[i], temp.b, time)

```
modl ( b.c[i], b.c[i+w], em.c, md.c, i, w, w, time )
            outp.em ( b.c[i+w], time )
      -- w*w-pixel shift detector array for result matrix C
      par
        par i={0 for w}
          par j=[0 for w]
            shda ( c.c[((i+1)*(w+1))+j], c.c[(i*(w+1))+(j+1)], md.c,
                   ((i*w)+j), 1, 1, time )
        par i=[0 \text{ for } ((2*w)+1)]
          if
            i <= w
              inp.shda ( c.c[i*(w+1)], time )
            true
              inp.shda ( c.c[i+(w*w)], time )
        par i=[0 for ((2*w)+1)]
          -- diagonal of matrix C
          var temp.c[to] :
          sea
            ί£
              i (= w
                outp.shda ( c.c[i], temp.c, time )
              true
                outp.shda ( c.c[i+((i-w)*w)], temp.c, time )
            par j=[0 for time]
              c[(i*time)+j] := temp.c[j] :
  main : two dummy diagonals for C; the uppermost
          and the lowermost.
-- input and output data for calculation AB = C.
var a[wo*to], b[wo*to], c[((2*wo)+1)*to],
-- actual matrix size; bandwidths; time.
    n, w, time :
seq
  get ( n, " size of problem " )
  get ( w, " bandwidth of input matrices ")
  time := ((n + w) - 1)
  get.n ( a, (w*time), " matrix A data seg " )
  get.n ( b, (w*time), " matrix B data seq " )
  system ( c, a, b, w, time )
  put.n ( c, (((2*w)+1)*time), " matrix C data seg " )
```

--

--

```
--
-- A.5.5
-- Soft-Systolic simulation for an Optical Processor
-- performing Digital Multiplication of two numbers a, b
-- with wordlength w, producing a number c with wordlength
-- wc = 2 * w = 1, (DMAC algorithm).
-- Emitter and modulator size = w; detector size = wc.
-- i/o routines.
external proc get ( var v, value s() ) :
external proc get.n ( var v[], value n, s[] ) :
external proc put.n ( value v[], n, s[] ) :
-- optical-systolic routines.
external proc emit ( chan datain, dataout, beamout[],
                     value start, num, step, time ) :
external proc modl ( chan datain, dataout, beamin[], beamout[],
                     value start, num, step, time ) :
external proc shda ( chan datain, dataout, beamin[],
                     value start, num, step, time ) :
external proc driver.em ( chan xout, value x(), time ) :
external proc outp.em ( chan xin, value time ) :
external proc inp.shda ( chan xout, value time ) :
external proc outp.shda ( chan xin, var x[], value time ) :
 -- max wordlength, time.
 def wo = 10, to = 2 :
 -- optical processor configuration.
proc system ( var c[],
               value a[], b[], w, time )=
  -- data communication.
  chan a.c[2*wo], b.c[2*wo], c.c[(wo+1)*(wo+1)],
        em.c[wo*wo], md.c[wo*wo] :
   -- help occam get the types right.
   var temp,
   -- wordlwngth of result number c
       WC :
   seq
     temp := a[0]
     temp := b[0]
     temp := c[0]
     wc := ((2 + w) - 1)
     par
       -- w-pixel emitter for number a
       par i=[0 for w]
         -- one bit of number a
         var temp.a[to] :
         seg
           par j=[0 for time]
             temp.a[j] := a[(i*time)+j]
           par
             driver.em ( a.c[i], temp.a, time )
              emit ( a.c[i], a.c[i+w], em.c, (w*i), w, 1, time )
              outp.em ( a.c[i+w], time )
       -- w-pixel modulator for number b
       par i=[0 for w]
         -- one bit of number b
         var temp.b[to] :
          pea
           par j=[0 for time]
              temp.b[j] := b[(i*time)+j]
            par
              driver.em ( b.c[i], temp.b, time )
              modl ( b.c(i), b.c(i+w), em.c, md.c, i, w, w, time )
```

```
outp.em ( b.c[i+w], time )
     -- (2*w-1)-pixel shift detector array for result number c
      par i=[0 for wc]
        -- one bit of number C
        var temp.c[to] :
        seq
          par
            inp.shda ( c.c[i], time )
            if
              ゴくw
                shda ( c.c[i], c.c[i+wc], md.c, i, (i+1), (w-1), time )
              true
                 shda { c.c[i], c.c[i+wc], md.c, ((i*w)-((w-1)*(w-1))),
                        (wc-i), (w-1), time )
            outp.shda ( c.c[i+wc], temp.c, time )
          par j=[0 for time]
            c[(i*time)+j] := temp.c[j] :
-- Main
-- input and output data for calculation ab - c
var a[wo*to], b[wo*to], c[((2*wo)-1)*to],
-- actual wordlength
    w :
seq
  get ( w, " wordlength of input numbers ")
 get.n ( a, (w*to), " number a bit seg " )
get.n ( b, (w*to), " number b bit seg " )
  system ( c, a, b, w, to )
  put.n ( c, (((2*w)-1)*to), " number c bit seg " )
```

0

```
-- A.6.1
---
-- Soft-Systolic Simulation Library
external proc str.to.screen(value s[]):
external proc num.to.screen(value n):
external proc num.from.keyboard(var n):
external proc fp.num.to.screen(value float n):
external proc fp.num.from.keyboard(var float n):
-- Get/Put fixed-/floating-point scalars/vectors from/to the screen
-- get put
                  fp
                                         n
library proc fp.get(var float v,value s[])=
  sea
    str.to.screen("*c *n Input ")
    str.to.screen($)
    fp.num.from.keyboard(v)
    fp.num.to.screen(v) :
library proc fp.get.n(var float v[],value n, s[])=
  seq
    str.to.screen("*c *n Input stream ")
    str.to.screen(s)
    seg i=[0 for n]
      seq
        fp.num.from.keyboard(v[i])
        fp.num.to.screen(v[i])
        str.to.screen(" ") :
library proc get(var v,value s[])=
  seq
    str.to.screen("*c *n Input ")
    str.to.screen(s)
    num.from.keyboard(v)
    num.to.screen(v) :
library proc get.n(var v[],value n, s[])=
  seq
    str.to.screen("*c *n Input stream ")
    str.to.screen(s)
     seg i=[0 for n]
      seq
        num.from.kevboard(v[i])
        num.to.screen(v[i])
        str.to.screen(" ") :
library proc fp.put (value float v,value s[])=
   $ 0 Q
     str.to.screen("*c *n Output " )
     str.to.screen(s)
     fp.num.to.screen(v) :
 library proc fp.put.n (value float v[],value n, s[])=
   seq
     str.to.screen("*c *n Output stream " )
     str.to.screen(s)
     seg i=[0 for n]
       seq
         fp.num.to.screen(v[i])
         str.to.screen(" ") :
 library proc put (value v,s[))=
   seq
```

```
str.to.screen("*c *n Output " )
    str.to.screen(s)
    num.to.screen(v) :
library proc put.n (value v[],n,s[])=
  seq
    str.to.screen("*c *n Output stream " )
    str.to.screen(s)
    seq i=[0 for n]
      sea
        num.to.screen(v[1])
        str.to.screen(" ") :
-- Source/Sink of fixed-/floating-point vectors of dummy/significant
                                                   A.
                                                        ¥
         si
                         fp
-- 50
-- elements with/without tag
            t
---
library proc so.x (chan xout, value x[], time) =
  seq tol0 for time}
    xout | x[t] :
library proc fp.so.x (chan xout, value float x(), value time) =
  seq t-[0 for time]
    xout ! x[t] :
library proc so.d (chan xout, value time) =
  seq t=[0 for time]
    xout 1 0 :
library proc fp.so.d (chan xout, value time) =
  seq t=[0 for time]
    xout 1 0.0 :
library proc si.x (chan xin, var x[], value time) =
  seq t=[0 for time]
    xin 7 x[t] :
library proc fp.si.x (chan xin, var float x(), value time) =
  seq t=[0 for time]
    xin 7 x[t] :
library proc si.d (chan xin, value time) =
  seq t=[0 for time]
    xin ? any :
library proc fp.si.d (chan xin, value time) =
   seq t=[0 for time]
    xin ? any :
 library proc so.x.t (chan xout, value x[], tag[], time} =
   seg t=[0 for time]
     xout 1 x[t]; tag[t] :
 library proc fp.so.x.t (chan xout, value float x[], value tag[], time) =
   seq t=[0 for time]
    xout ! x[t]; tag[t] :
library proc so.d.t (chan xout, value time) =
   seg t=[0 for time]
     xout 1 0; 0 :
 library proc fp.so.d.t (chan xout, value time) =
   seq t=[0 for time]
     xout 1 0.0; 0 :
```

σ

ω

σ

```
library proc sing t (chan xin, var x[], tag[], value time) =
  seq t=[0 for time]
    xin 7 x[t]; taq[t] :
library proc fp.si.x.t (chan xin, var float x[], var tag[], value time) =
  seq t=[0 for time]
    xin 7 x(t); tag(t) :
library proc si.d.t (chan xin, value time) =
  var r(2):
  seq t=[0 for time]
    xin 7 r(0); r(1);
library proc fp.si.d.t (chan xin, value time) =
  var float r :
  VAT S :
  seg t=[0 for time]
    xin ? r; s :
-- Fixed-/Floating-point conversions
library proc fp.float (value fix, var float flt) =
  seq
    flt := fix :
library proc fp.fix (value float flt, var fix) =
  seq
    fix := flt :
___
-- Delay/ for fixed-/floating-point variables with/without tag
-- dl
                     fp
                                               ÷.
def do = 20 :
library proc dl (chan xin, xout,
                  value d, time )=
  var x[do] :
  Бeq
    par i=[0 for (d+1)]
      x[1] := 0
     seq i=[0 for time]
      seq
         seq
           xin 7 x[0]
           xout 1 x d
         seq j=[0 for d]
          \bar{x}(\bar{d}-j) := x((d-j)-1) :
library proc fp.dl (chan xin, xout,
                     value d, time )=
  var float x[do] :
  seq
     par i=[0 for (d+1)]
      x[i] := 0.0
     seg i={0 for time}
       seq
         seq
           xin 7 x[0]
           xout [ x[d]
         seq j=[0 for d]
           x[d-j] := x[(d-j)-1] :
 library proc dl.t (chan xin, xout,
                     value d, time )=
  var x(do), t(do) :
   sea
     par i=[0 for (d+1)]
```

,

```
par
       x[1] := 0
       tii := 0
    seq i=[0 for time]
     seq
        seq
          xin 7 x[0]; t[0]
          xout 1 x(d); t(d)
        seq j=[0 for d]
          par
            x(d-1) := x((d-1)-1)
            t[d-j] := t[(d-j)-1] :
library proc fp.dl.t (chan xin, xout,
                      value d, time )=
  var float x[do] :
  var t(do) :
  seq
    par i=[0 for (d+1)]
      par
        x[1] := 0.0
        t(i) := 0
    seq i={0 for time}
      seq
        seq
          xin 7 x[0]; t[0]
          xout l x[d]; t[d]
        seq j=[0 for d]
          par
            x[d-j] := x[(d-j)-1]
            t[d-j] := t[(d-j)-1] :
-- Branching for fixed-/floating-point variables with/without tag
                        fp
                                                 t
-- br
library proc br (chan xin, xout1, xout2,
                 value time} =
  var x :
  seq
    x := 0
    seq i=[0 for time]
       seq
        xin ? x
        par
          xoutl 1 x
          xout2 1 x :
library proc fp.br (chan xin, xout1, xout2,
                     value time) -
  var float x :
   seq
    x := 0.0
    seg i=[0 for time]
       seq
         xin 7 x
         par
          xoutl 1 x
          xout2 | x :
 library proc br.t (chan xin, xout1, xout2,
                    value time) =
   var x, t :
   seq
     par
       x := 0
       t := 0
```

- 637

```
seg i=[0 for time]
      pea
        xin 7 x: t
        par
          xoutl 1 x; t
          xout2 1 x: t :
library proc fp.br.t (chan xin, xout1, xout2,
                      value time) =
  var float x :
  vart:
  seg
    par
      x := 0.0
      t := 0
    seq i=[0 for time]
      seq
        xin ? x; t
        par
          xoutl 1 X; t
          xout2 1 x; t :
-- Links for fixed-/floating-point variables with/without tag
-- 1k
                    fo
                                              ÷.
library proc lk (chan xin, xout, value time) =
  var x :
  seq
    x := 0
    seq t=[0 for time]
      seq
        xin 7 x
        xout I x :
library proc fp.lk (chan xin, xout, value time) =
  var float x :
  seq
    \bar{x} := 0.0
    seg t=[0 for time]
      sea
        xin ? x
        xout ! x :
library proc lk.t (chan xin, xout, value time) =
  var x, tag :
  seq
    par
      x := 0.0
       tag := 0
     seg t=[0 for time]
       seq
         xin 7 x; taq
         xout 1 x; tag :
 library proc fp.lk.t (chan xin, xout, value time) -
   var float x :
   var tag :
   seq
     par
       x := 0.0
       tag := 0
     seq t=[0 for time]
       seq
         xin 7 x; tag
         xout 1 x; tag :
 ---
```

-- Inner Product Step cell for a Linear array library proc ips.l (chan cin, xin, yin, xout, yout, value time)= var float c, x[2], y[2] : seq --- initialisation par c := 0.0par i=[0 for 2] par x[i] := 0.0v[i] := 0.0 -- main operation seq i=[0 for time] seg -- 1/o par cin 7 c xin ? x[0] yin 7 y[0] xout [x[1] yout I y[1] -- calculation par x[1] := x[0]y(1) := y(0) + (x(0) + c) :--- Inner Product Step Cell for the Hex-connected array library proc ips.h (chan ain, bin, cin, aout, bout, cout, . value time) = var float a[2], b[2], c[2] : seq -- initialise par i=[0 for 2] par a[i] := 0.0 b[i] := 0.0c[i] := 0.0 -- main operation seq i=[0 for time] seg -- 1/0 par ain 7 a[0] bin 7 b[0] cin ? c[0] aout 1 a[1] bout | b[1] cout | c(1) -- calculation par c[1] := c[0] + (a[0] + b[0])a(1) := a(0)b[1] := b[0] :-- Building Blocks for the reusable matrix multiplication array ---- Inner Product Step Cell : Multiply - Accumulate -- When the last accumulation occurs the result is sent as feedback -- and the accumulator is reset for next cycle of calculations. library proc ips.r (chan ein, eout, win, wout, nin, nout, sin, sout, value cycle, reset, time) = var float e[2], w[2], n[2], s[2], acc :

t

seg - initialise seg par i=[0 for 2] par e[i] := 0.0 w[i] := 0.0 n(i) := 0.0s[i] := 0.0 acc := 0.0 -- main operation seq i=[0 for time] seq -- i/o par ein ? e[0] win 7 w[0] nin 7 n[0] sin 7 s(0) eout 1 e[1] wout | w[1] nout [n[1] sout i s[1] -- calculation par acc := acc + (w[0] * n[0]) e[1] := w[0] w[1] := e[0]n(1) := s[0] s(1) := n(0)-- control if (i \ cycle) = reset seq par w[1] := acc n[1] := acc acc := 0.0 : ----- Multiplexer -- It adds the source input to the array feedback if fb -- is true; otherwise it accepts the source input. library proc mux.r { chan sin, sout, fin, fout, value time) = var float s[2], f[2]: var fb : seq initialise par par i=[0 for 2] par s[i] := 0.0 f[i] := 0.0 fb := 0-- main operation seg i=[0 for time] seg -- 1/o par sin ? s[0]; fb fin ? f[0] sout 1 s(1) fout 1 f[1] -- control par

if
 fb = 1
 f[1] := (f[0] + s[0])
 true
 f[1] := s[0]
 s[1] := f[0] :

```
-- A.6.2
-
-- Soft-systolic simulation library of optical
-- systolic algorithms.
--
-- Definition of a pixel of an emitter : a beam with
-- intensity proportional to the input data item is
-- emitted towards the modulators as specified by the
-- topology of the Optical Procesor. The data item moves
-- systolically to the next pixel.
library proc emit ( chan datain, dataout, beamout[],
                    value start, num, step, time ) =
  var data[2] :
  seq
    Initialisation
    par i=[0 for 2]
      data[i] := 0
    -- main operation
    seg i=[0 for time]
      seq
        par
           datain ? data[0]
           dataout | data[1]
         par
           par j=[0 for num]
            beamout[start+(j*step)] 1 data[1]
           data[1] := data[0] :
-- Definition of a pixel of a modulator : a beam from an
-- emitter is modulated in proportion to the input data item
-- i.e a multiplication occurs. The data item moves
-- systolically to the next pixel.
library proc modl ( chan datain, dataout, beamin{}, beamout{},
                     value start, num, step, time ) =
   -- max number of incident beams
   def bo = 10 :
   var data[2], beam[bo] :
   seq
     -- initialisation
     par
       par i=[0 for 2]
         data[i] := 0
       par i=[0 for num]
         beam[i] := 0
     -- main operation
     seq i=[0 for time]
       seq
         par
           datain 7 data[0]
           dataout | data[1]
           par j=[0 for num]
             beamin(start+(j*step)] ? beam[j]
         par j=[0 for num]
           beam[j] := (data[0] * beam[j] )
         par
           par j=[0 for num]
              beamout[start+(j*step)] 1 beam[j]
            data[1] := data[0] :
 -- Definition of a pixel of a (shift) detector array : the
 -- incident light beams are transformed to electic charges
 -- and accumulated onto a "bin" ; i.e two additions occur :
```

-- one of the incident beams and one of the accumulated charges.

library proc shda (chan datain, dataout, beamin[], value start, num, step, time)= -- max number of incident beams. def bo = 10: var data[2], sum, beam[bo] : seq -- initialisation par par i=[0 for 2] data[i] := 0 par i=[0 for num] beam[1] := 0 -- main operation seq i=[0 for time] seq par datain ? data[0] dataout ! data[1] par j=[0 for num] beamin[start+(j*step)] ? beam[j] sum := 0 par j=(0 for num) sum := sum + beam(j) data[1] := (data[0] + sum) : -- Driver for emitter, modulator. library proc driver.em (chan xout, value x[], time)= seq i=[0 for time] xout 1 x[i] : -- Dummy output for emitter, modulator. library proc outp.em (chan xin, value time)= seq i=[0 for time] xin 7 any : -- Input for (shift) detector array. library proc inp.shda (chan xout, value time)= seg i=[0 for time] xout 1 0 : -- Output of a (shift) detector array. library proc outp.shda (chan xin, var x[], value time)= seg i=[0 for time] xin 7 x[i] :

-- The result moves systolically to the next pixel.

640 -

. .

• • • _____