

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



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

 $\mathcal{L} = \mathcal{L} \cup \mathcal{L}$ ' .-.  $360s$ c  $N0 - D \times 8/0S$  $\cdot\cdot\cdot$  .  $\cdot\cdot\cdot$ **LOUGHBOROUGH** *:* \ UNIVERSITY OF TECHNOLOGY LIBRARY AUTHOR/FILING TITLE  $MARCARITIS,$  KG ' ' **ACCESSION/COPY NO.** ~· ~-voL~f.io~--~---- -c-~i.~~~~{/:-1:-*\_\_\_ , \_\_\_\_\_\_\_\_\_\_\_* -· .  $3.0 - 107$  1989  $-5.111$   $+997$  $-6$  JUL 1990  $\mathbb{R}^3$  . •  $-6$  JHL  $1990$  $-2$  Jul. 1993 - 5 JUL 1991 '  $18~0$   $0.144$ 001 6645 02  $\setminus$ 

 $\label{eq:2.1} \frac{1}{\sqrt{2}}\sum_{i=1}^n\frac{1}{\sqrt{2\pi}}\sum_{i=1}^n\frac{1}{\sqrt{2\pi}}\sum_{i=1}^n\frac{1}{\sqrt{2\pi}}\sum_{i=1}^n\frac{1}{\sqrt{2\pi}}\sum_{i=1}^n\frac{1}{\sqrt{2\pi}}\sum_{i=1}^n\frac{1}{\sqrt{2\pi}}\sum_{i=1}^n\frac{1}{\sqrt{2\pi}}\sum_{i=1}^n\frac{1}{\sqrt{2\pi}}\sum_{i=1}^n\frac{1}{\sqrt{2\pi}}\sum_{i=1}^n\frac{$  $\mathcal{A}^{\text{max}}_{\text{max}}$  and  $\mathcal{A}^{\text{max}}_{\text{max}}$  $\mathcal{L}(\mathcal{A})$  and  $\mathcal{L}(\mathcal{A})$  .  $\frac{1}{2} \sum_{i=1}^{n} \frac{1}{i} \sum_{j=1}^{n} \frac{1}{j} \sum_{j=1}^{n$  $\label{eq:2.1} \frac{1}{\sqrt{2\pi}}\sum_{i=1}^n\frac{1}{\sqrt{2\pi}}\sum_{i=1}^n\frac{1}{\sqrt{2\pi}}\sum_{i=1}^n\frac{1}{\sqrt{2\pi}}\sum_{i=1}^n\frac{1}{\sqrt{2\pi}}\sum_{i=1}^n\frac{1}{\sqrt{2\pi}}\sum_{i=1}^n\frac{1}{\sqrt{2\pi}}\sum_{i=1}^n\frac{1}{\sqrt{2\pi}}\sum_{i=1}^n\frac{1}{\sqrt{2\pi}}\sum_{i=1}^n\frac{1}{\sqrt{2\pi}}\sum_{i=1}^n\$  $\mathcal{O}(\mathcal{O}_\mathcal{O})$  $\mathcal{L}^{\text{max}}_{\text{max}}$  and  $\mathcal{L}^{\text{max}}_{\text{max}}$  $\mathcal{L}^{\text{max}}_{\text{max}}$  .

 $\sim 0.5$ 

 $\mathcal{F}_{\mathcal{G}}$ 

#### **A STUDY**

**0 F S Y S T 0 L I C A L G 0** R I **T H M S F 0 .R V L S I PROCESSOR ARRAYS A N D 0 P T I C A L COMPUTING** 

By

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

#### V 0 L **U M E** I I

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.

(§) by K.G.Margaritis, 1987.



 $\frac{1}{\sqrt{2}}\sum_{i=1}^{n} \frac{1}{\sqrt{2}}\left(\frac{1}{\sqrt{2}}\right)^2\left(\frac{1}{\sqrt{2}}\right)^2.$ 

 $\label{eq:2.1} \begin{split} \mathcal{L}_{\text{max}}(\mathbf{r}) & = \mathcal{L}_{\text{max}}(\mathbf{r}) \mathcal{L}_{\text{max}}(\mathbf{r})$ 

 $\label{eq:2.1} \frac{d\mathbf{r}}{d\mathbf{r}} = \frac{1}{2}\sum_{i=1}^n \frac{d\mathbf{r}}{d\mathbf{r}} \left[ \frac{d\mathbf{r}}{d\mathbf{r}} \right] \left[ \frac{d\mathbf{r}}{d$ 

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 starts with some basic definitions and terminology for sys- tolic 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 Chapter 5 presents systolic algorithms for the<br>clution and the updating of the solution of efficient solution and the updating of the solution 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 expan-<br>sion, in iterative solution of linear systems of equations, sion, 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 andjor in time. The systolic implementation of successive matrix-matrix multiplications implementation of successive matrix-matrix multiplications<br>is discussed and then a group of algorithms based on matrix 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 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**

**0 F** S Y S **T 0 L** I C **A L** G 0 **R** I T **H M S F 0 R V L** 5 I **P R 0 C E 5 S 0 R 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

### V 0 L U M E I

 $\sim$ 



 $\sim$   $\sim$ 

### CHAPTER 1

#### INTRODUCTION



### C H A P T E R 2

 $\sim 10^7$ 

 $\sim 10^{-10}$ 

 $\sim$   $\sim$ 

# BASIC MATHEMATICAL DEFINITIONS





# C H A P T E R 3

# SYSTOLIC ALGORITHMS AND ARCHITECTURES



 $\bar{\gamma}$ 

 $\bar{\mathcal{A}}$ 



# CHAPTER 4

# SYSTOLIC SOLUTION OF POLYNOMIAL EQUATIONS



### - viii-

# C H A P T E R 5

### SYSTOLIC LU DECOMPOSITION



# VOLUME II



# C H A P T E R 6

# SYSTOLIC MATRIX VECTOR MULTIPLICATION PIPELINES





# C H A P T E R 7

# SYSTOLIC ALGORITHMS USING MATRIX POWERS



# C H A P T E R 8

# OPTICAL SYSTOLIC ALGORITHMS



# C H A P T E R 9

### CONCLUSIONS



# L I S T 0 F F I G U R E S

 $\mathcal{A}^{\text{max}}_{\text{max}}$ 

 $\sim$ 

 $\mathcal{A}$ 

 $\ddot{\phantom{a}}$ 

 $\ddot{\phantom{1}}$ 



 $\sim$ 



7

 $\bar{z}$ 



 $\bar{\beta}$ 



 $\label{eq:2.1} \mathcal{L}(\mathcal{L}^{\text{max}}_{\mathcal{L}}(\mathcal{L}^{\text{max}}_{\mathcal{L}})) \leq \mathcal{L}(\mathcal{L}^{\text{max}}_{\mathcal{L}}(\mathcal{L}^{\text{max}}_{\mathcal{L}}))$ 

 $\label{eq:2.1} \frac{1}{\sqrt{2\pi}}\int_{0}^{\infty}\frac{1}{\sqrt{2\pi}}\left(\frac{1}{\sqrt{2\pi}}\right)^{2\alpha} \frac{1}{\sqrt{2\pi}}\frac{1}{\sqrt{2\pi}}\int_{0}^{\infty}\frac{1}{\sqrt{2\pi}}\frac{1}{\sqrt{2\pi}}\frac{1}{\sqrt{2\pi}}\frac{1}{\sqrt{2\pi}}\frac{1}{\sqrt{2\pi}}\frac{1}{\sqrt{2\pi}}\frac{1}{\sqrt{2\pi}}\frac{1}{\sqrt{2\pi}}\frac{1}{\sqrt{2\pi}}\frac{1}{\sqrt{2\pi}}\frac{$ 

 $\mathcal{A}^{\text{max}}_{\text{max}}$ 

 $\mathcal{L}^{\text{max}}_{\text{max}}$  and  $\mathcal{L}^{\text{max}}_{\text{max}}$ 

 $- xiv -$ 



 $\sim 10^{-1}$ 

 $\frac{1}{2}$ 

 $\mathbb{E}_{\mathcal{L}}$ 

 $\label{eq:2.1} \mathcal{L}(\mathcal{L}^{\text{max}}_{\mathcal{L}}(\mathcal{L}^{\text{max}}_{\mathcal{L}})) \leq \mathcal{L}(\mathcal{L}^{\text{max}}_{\mathcal{L}}(\mathcal{L}^{\text{max}}_{\mathcal{L}}))$ 



 $\mathcal{L}^{\text{max}}_{\text{max}}$  and  $\mathcal{L}^{\text{max}}_{\text{max}}$ 

 $\mathcal{A}_{\mathcal{A}}$ 



 $\ddot{\phantom{0}}$ 

 $\hat{\boldsymbol{\beta}}$ 

 $\mathcal{L}^{\text{max}}_{\text{max}}$  , where  $\mathcal{L}^{\text{max}}_{\text{max}}$ 

 $\mathcal{L}^{\text{max}}_{\text{max}}$  and  $\mathcal{L}^{\text{max}}_{\text{max}}$ 

 $\mathcal{L}^{\text{max}}_{\text{max}}$  and  $\mathcal{L}^{\text{max}}_{\text{max}}$ 

 $\sim$ 

# L I S T 0 F T A B L E S



 $\cdot$ 

Page

# **L** I S T 0 F P R 0 G R A M S

 $\label{eq:2} \mathcal{L}(\mathcal{L}^{\text{max}}_{\mathcal{L}}(\mathcal{L}^{\text{max}}_{\mathcal{L}}(\mathcal{L}^{\text{max}}_{\mathcal{L}^{\text{max}}_{\mathcal{L}}(\mathcal{L}^{\text{max}}_{\mathcal{L}^{\text{max}}_{\mathcal{L}^{\text{max}}_{\mathcal{L}^{\text{max}}_{\mathcal{L}^{\text{max}}_{\mathcal{L}^{\text{max}}_{\mathcal{L}^{\text{max}}_{\mathcal{L}^{\text{max}}_{\mathcal{L}^{\text{max}}_{\mathcal{L}^{\text{max}}_{\mathcal{L}^$ 

 $\sim 10^6$ 

 $\mathcal{L}_{\text{max}}$ 

 $\sim 40$ 



 $\mathcal{L}^{\text{max}}_{\text{max}}$ 



 $\mathcal{L}^{\text{max}}_{\text{max}}$  and  $\mathcal{L}^{\text{max}}_{\text{max}}$ 

#### **C H** A P **T E R 6**

### SYSTOLIC MATRIX VECTOR MULTIPLICATION PIPELINES

#### 6.1 INTRODUCTION

The matrix vector multiplication (mvm) computation is probably the single most useful operation in signal process-. ing, 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

<sup>\*</sup> 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  $(6.2.1)$ ; x moves to the left, while the diagonals of matrix A are accessed through vertical channels; the entries of 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 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 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  $x$  and  $y$  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  $y$ . 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.

 $- 323 -$ 



Fig.6.2.4. Dataflow for unidirectional mvm  $(y$  is delayed).
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 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 dataf1ow 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.

The unidirectional mvm systolic designs proposed herein allow for compact data sequences, and therefore optimal computation time; furthermore fault-tolerance tech~ niques 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),[2951 has the same characteristics with the mvm arrays discussed herein.

 $-325 -$ 



Fig.6.2.5. Dataflow for unidirectional mvm ( $\times$  is delayed).

 $\bar{\bar{z}}$ 

 $\hat{\mathbf{r}}$ 

 $\bar{\mathcal{A}}$ 

 $\hat{\boldsymbol{\beta}}$ 



Fig.6.2.6. Systolic array for unidirectional mvm.

## 6.3 IMPROVED SYSTOLIC DESIGNS FOR THE ITERATIVE SOLUTION OF LINEAR SYSTEMS \*

Given a linear system of equations,  $Ax = b$ , which, without loss of generality has been so ordered that  $a_{1,1} \neq 0$ , for all i. The system can be rewritten as

$$
\mathbf{x} = \mathbf{M}\mathbf{x} + \mathbf{g} \tag{6.3.1}
$$

where M is the Jacobi matrix of A, with

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

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

(ii) Jacobi overrelaxation method (JOR)

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

where is  $\omega$  the overrelaxation factor.

(iii) Gauss-Seidel method (GS)  $x^{(k+1)} = Lx^{(k+1)} + Ux^{(k)} + g$ 

(6.3.5)

<sup>\*</sup> 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)

$$
\underline{x}^{(k+1)} = \omega(\underline{\mathbf{L}}\underline{x}^{(k+1)} + \underline{\mathbf{U}}\underline{x}^{(k)} + \underline{q}) + (1-\omega)\underline{x}^{(k)}
$$
(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

$$
-\frac{a_{i1}}{a_{1i}}, -\frac{a_{i1}}{a_{1i}}, i \neq j \text{ and } \frac{b_{i}}{a_{1i}}, \frac{b_{i}}{a_{1i}} \qquad (6.3.7)
$$

are calculated repetitively in each step of the iterative



4

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

*w w*  0

 $\mathbf{L}$ 



Fig.6.3.2. Pipeline block for JOR method.

 $\frac{231}{1}$  $\mathbf{L}$ 

 $\pmb{\downarrow}$ 



Fig.6.3.3. Pipeline block for GS method.

*w w*  ....

 $\mathbf{I}$ 



Fig.6.3.4. Pipeline block for SOR method.



$$
\mathbf{^{a}_{out}}
$$

$$
x1_{in}
$$
\n
$$
x2_{out}
$$
\n
$$
x3_{out}
$$
\n
$$
x4_{in}
$$
\n
$$
x2_{out}
$$
\n
$$
x3_{out}
$$
\n
$$
x4_{in}
$$
\n
$$
x2_{in}
$$
\n
$$
x3_{out}
$$
\n
$$
x2_{in}
$$
\n
$$
x3_{out}
$$
\n
$$
x4_{out}
$$

$$
x^{2} \text{in}
$$
\n
$$
x^{2} \text{in}
$$



 $x_0$ <sub>out</sub>:  $x_1$ <sub>in</sub>  $x^2$ <sub>out</sub>: =  $x^2$ <sub>in</sub>/a<sub>in</sub>  $a_{\text{out}}$  $a_{\text{in}}$ 

$$
x_{out} := x_{in}
$$
  
\n
$$
x_{out} = x_{in} / a_{in} + x_{in}
$$
  
\n
$$
(x := x_{in} d_{in} d_{out} = 2p-1
$$
 cycles)  
\n
$$
a_{out} := a_{in}
$$
  
\n
$$
a_{out} := a_{in}
$$
  
\n
$$
a_{out} := a_{in}
$$

 $x_{\text{out}}^i$ :  $(x_{\text{in}}^i+x_{\text{in}}^i+b_{\text{in}}^i/a_{\text{in}}^i)$  $\mathbf{a}_{\text{out}} \colon = \mathbf{a}_{\text{in}}$  $b_{\text{out}} := b_{\text{in}}$ 

 $x1$ <sub>out</sub>:= $(x1_{in}+x2_{in}+b_{in})/a_{in}+x\omega_{in}$ <br>( $x:=x3_{in}$  delayed 2(p-1) cycles)  $a_{\text{out}} := a_{\text{in}}$  $b_{\text{out}}$ := $b_{\text{in}}$ 



 $-334 -$ 



Table 6.3.1. Area-time requirements for systolic pipelines in Fig.6.3.1-5.

 $\mathbf{I}$  : w w U1  $\mathbf{I}$ 

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 prepro cessing. 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

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

$$
\underline{x}^{(k+1)} = \underline{u} L \underline{x}^{(k+1)} + ((1 - u) L + u) \underline{x}^{(k)} + u \underline{g}.
$$
 (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  $\omega/a_{\text{i}i}$ ; 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 -(w/a<sub>ii</sub>)a<sub>ij</sub>, except for the cell corresponding to b that only calculates  $(\omega/a_{i,i})b_i$ . 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.

3<br>ما

 $\mathbf{I}$ 



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

 $\mathbf{I}$ w w 0>  $\mathbf{I}$ 

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



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

 $\lambda$ 

 $\mathbf{r}$ 



 $\Delta$ 

Fig.6.3.8. Preprocessor for J, JOR methods.

 $\blacksquare$ 

 $\blacksquare$ 



 $\sim$ 

 $\sim 10^{11}$ 

 $\sim 10^{-1}$ 

Fig.6.3.9. Preprocessor for GS, SOR methods.

 $\mathbf{I}$ 

 $\mathbf{I}$ 



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

 $\bullet$ 

 $\sim 10^{-1}$ 

*w*  ... w  $\mathbf{r}$ 

 $\mathbf{I}$ 



 $\sim$ 



 $\Delta$ 

 $\blacksquare$ 344  $\mathbf{r}$ 

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 are required for the result to be produced; on the other 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 SYSTOLIC NETWORKS FOR ITERATIVE MEHODS USING CYCLIC 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)

$$
-\frac{d^2x(z)}{dz^2} + \sigma(z)x(z) = \phi(z)
$$
 (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  $A\underline{x}=\underline{b}$ , where A is a tridiagonal, irreducible, diagonal dominant matrix, with

$$
a_{ij}^{0}
$$
,  $a_{ij}^{0}$ ,  $ij$ ,  $i=1,2,...,n$ . (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

$$
A = \begin{bmatrix} D_1 & A_1 \\ \hline & & \\ A_2 & & D_2 \end{bmatrix} \qquad M = \begin{bmatrix} 0 & F_1 \\ \hline & & \\ F_2 & & \end{bmatrix} \qquad (6.4.3)
$$

Thus, the Jacobi matrix produced by the coefficient matrix A of the discrete approximation to the solution of a general second order ordinary differential eqution 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









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









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

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  $\mathbb{F}_2^+$  is upper triangular bidiagonal matrix, of size  $(\lceil (n+1)/2 \rceil x \lceil (n+1)/2 \rceil)$ ; for n odd the addition of a dummy column (row) is necessary.

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

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

Thus, method J (6.3.3) can be written as

$$
\frac{x_1^{(k+1)} = F_1 x_2^{(k)} + g_1}{x_2^{(k+1)} = F_2 x_1^{(k)} + g_2}
$$
\n(6.4.5)

i.e. a pair of decoupled equations is created, where  $\underline{x}_1^{(k+1)}$ depends on only  $x_2^{(k)}$  and vice versa:  $-1$ 



Since  $x_1, x_2$  have both  $\lceil (n+1)/2 \rceil$  entries, the two systems can

$$
\frac{x_1^{(k+1)} = \omega (F_1 \frac{x_2^{(k)} + g_1 + (1 - \omega) x_1^{(k)}}{2} + \frac{x_2^{(k+1)} = \omega (F_2 \frac{x_1^{(k)} + g_2 + (1 - \omega) x_2^{(k)}}{2} + \frac{x_2^{(k+1)} + g_2^{(k+1)} + \frac{x_2^{(k+1)} + g_2^{(k+1)}}{2} + \frac{x_2^{(k+1)} + g_2^{(k+1)} + \frac{x_2^{(k+1)} + g_2^{(k+1)} + g_2^{(k+1)}}{2} + \frac{x_2^{(k+1)} + g_2^{(k+1)} + \frac{x_2^{(k+1)} + g_2^{(k+1)} + g_2^{(k+1)} + \frac{x_2^{(k+1)} + g
$$



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

defined as  
\n
$$
\sum_{1}^{(k+1)} = F_1 \sum_{2}^{(k)} + g_1
$$
\n
$$
\sum_{2}^{(k+1)} = F_2 \sum_{1}^{(k+1)} + g_2
$$
\n(6.4.9)

Alternatively, since

$$
\begin{bmatrix}\n0 & F_1 \\
F_2 & 0\n\end{bmatrix}^2 = \begin{bmatrix}\nF_1F_2 & 0 \\
0 & F_2F_1\n\end{bmatrix}
$$
\n(6.4.10)

the solution of (6.4.6) is equivalent to the solution of the uncoupled equations

$$
\begin{aligned}\n\frac{x_1^{(k+1)} - F_1 F_2 x_1^{(k)} + (F_1 g_2 + g_1)}{x_2^{(k+1)} - F_2 F_1 x_2^{(k)} + (F_2 g_1 + g_2)}\n\end{aligned}
$$
\n(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:

$$
\sum_{\underline{x}_1} = F_1 \underline{x}_2 + \underline{g}_1
$$
\n
$$
\sum_{\underline{x}_2} = F_2 F_1 \underline{x}_2 + (F_2 \underline{g}_1 + \underline{g}_2)
$$
\n(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  $\texttt{F}_2\texttt{F}_1$  and  $(\texttt{F}_2\texttt{G}_1\texttt{+}\texttt{G}_2)$  are. formed. These two calculations can be combined by squaring the augmented matrix

$$
\begin{bmatrix} 1 & \frac{\mathbf{0}^{\mathrm{T}}}{2} \\ \frac{\mathbf{0}}{\mathbf{0}} & \mathbf{M} \end{bmatrix}^{2} = \begin{bmatrix} 1 & \frac{\mathbf{0}^{\mathrm{T}}}{2} \\ \frac{\mathbf{0}^{\mathrm{+M}}\mathbf{0}}{2} & \mathbf{M}^{2} \end{bmatrix},
$$
\n(6.4.15)

with  $M^2$  as in (6.4.10) and  $q+Mq$  analysed as

$$
\begin{bmatrix} g_1 \\ -g_2 \end{bmatrix} + \begin{bmatrix} 0 & F_1 \\ -F_2 & 0 \end{bmatrix} \begin{bmatrix} g_1 \\ -g_2 \end{bmatrix} = \begin{bmatrix} F_1 g_2 + g_1 \\ -F_2 g_1 + g_2 \end{bmatrix} .
$$
 (6.4.16)

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

$$
\frac{x_1^{(k+1)} = \omega (F_1 \underline{x}_2^{(k)} + g_1) + (1-\omega) \underline{x}_1^{(k)}
$$
  
\n
$$
\underline{x}_2^{(k+1)} = \omega (F_2 \underline{x}_1^{(k+1)} + g_2) + (1-\omega) \underline{x}_2^{(k)}
$$
 (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  $x_1^{(k)}$  and  $x_2^{(k)}$  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=S. 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-

 $-352 -$ 



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



 $\bullet$ 

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

 $\mathbf{r}$ ይ<br>ይ<br>ል  $\mathbf{f}$ 



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

w<br>u  $\mathbf{F}$ 

 $\mathbf{I}$ 

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 pipeline accepts input from stage k of both pipelines; furthermore, the computation is more complex since  $(1-\omega)\underline{x}_1$  or  $(1-\omega)\underline{x}_2$  should be added to the result of the mvips operation. This is accomplished by means of an additional IPS cell performing the calculation  $\omega q_1 + (1-\omega)\underline{x}_1$  or  $\omega q_2 + (1-\omega)\underline{x}_2$ .

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 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 <code>imposes</code> 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  $\mathtt{F_1}, \mathtt{F_2}, \mathtt{g_1}, \mathtt{g_2}$  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-w) 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 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  $F_2F_1+g_2$ . 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  $\omega$  it produces  $F_1,F_2, g_1, g_2.$  The designs are similar to that in Fig.6.3.1l;what differs is the arrangement of the data streams and reformatting delays so that the output sequence format of the postprocessor is identical to that required by the next block of the system. For designs 3 and 4, the next block is a combination of mmm and mvm arrays that performs the calculations in (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<br>solution of 2-cyclic systems.

 $\mathbf{I}$ 359  $\mathbf{I}$ 

# 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  $p \geq 2$ , if by a permutation of rows and columns it can be partitioned into the form:



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



(6.5.2)

where

$$
M_{1,p} = -A_{11}^{-1}A_{1,p}
$$
  
\n
$$
M_{r,r-1} = -A_{rr}^{-1}A_{r,r-1}, r=2,...,p
$$
\n(6.5.3)
Matrix M is said to be weakly cyclic of index p; if all  $A_{i,i}$ are diagonal matrices then (6.5.3) can be rewritten as:

$$
m_{ij} = -a_{ij}/a_{ji}
$$
 (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  $A_{i,i}$  are diagonal matrices and therefore the elements of the corresponding Jacobi matrix is given by  $(6.5.4)$ . The bandwidth of the submatrices  $A_{i,j}$  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  $A\underline{x}=\underline{b}$ , for  $A$  a p-cyclic matrix, takes the form:



 $p=2$ ,  $s=3$ 



p=2, s=S

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

$$
\underline{x}_{1}^{(k+1)} = M_{1,p} \cdot \underline{x}_{p}^{(k)} + \underline{q}_{1}
$$
\n
$$
\underline{x}_{i}^{(k+1)} = M_{i,i-1} \cdot \underline{x}_{i-1}^{(k)} + \underline{q}_{i} , i=2,...,p
$$
\n(6.5.7)

Similarly, the JOR method (6.3.4) has the form

$$
\underline{x}_{1}^{(k+1)} = \omega(M_{1,p} \cdot \underline{x}_{p}^{(k)} + \underline{q}_{1}) + (1-\omega) \underline{x}_{1}^{(k)},
$$
\n
$$
\underline{x}_{i}^{(k+1)} = \omega(M_{1,i-1} \cdot \underline{x}_{i-1}^{(k)} + \underline{q}_{i}) + (1-\omega) \underline{x}_{i}^{(k)}, \quad i=2,...,p \qquad (6.5.8)
$$

The GS method (6.3.5) is defined as

$$
\underline{x}_1^{(k+1)} = M_{1, p} \underline{x}_p^{(k)} + \underline{q}_1
$$
\n
$$
\underline{x}_i^{(k+1)} = M_{i, i-1} \underline{x}_{i-1}^{(k+1)} + \underline{q}_i , i = 2, ..., p.
$$
\n(6.5.9)

 $-364 -$ 

Finally the SOR method (6.3.6) is modified as follows

$$
\underline{x}_{1}^{(k+1)} = \omega(M_{1, p} \underline{x}_{p}^{(k)} + \underline{q}_{1}) + (1-\omega) \underline{x}_{1}^{(k)}
$$
\n
$$
\underline{x}_{i}^{(k+1)} = \omega(M_{i, i-1} \underline{x}_{i-1}^{(k+1)} + \underline{q}_{i}) + (1-\omega) \underline{x}_{i}^{(k)}, i=2,...,p \qquad (6.5.10)
$$

computational networks for the four methods are given in Fig.6.5.2,3. For the J,JOR methods the subvectors  $\underline{x}_1^{(k+1)}$ , $\underline{x}_2^{(k+1)}$ , ..., $\underline{x}_p^{(k+1)}$  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  $M^D$  is completely reducible, i.e. it has the form









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







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



 $\mathcal{L}_{\text{max}}$ 

 $\sim$ 

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

**Contractor** 

 $\sim 10^{-1}$ 

- 369 -

where

$$
C_{1} = M_{1,p} M_{p,p-1} M_{p-1,p-2} \cdots M_{2,1}
$$
  
\n
$$
C_{2} = M_{2,1} M_{1,p} M_{p,p-1} \cdots M_{3,2}
$$
  
\n...  
\n
$$
C_{p} = M_{p,p-1} M_{p-1,p-2} M_{p-2,p-3} \cdots M_{1,p}
$$
  
\n(6.5.12)

Thus, equation (6.5.6) is equivalent to



Notice that  $h$  can be computed simultaneously with  $M^P$  if the calculation



(6.5.14)

is performed instead of  $M^P$ . Thus,

$$
\underline{h}_{1} = M_{1,p}(M_{P,P-1}(\dots M_{32}g_{2}+g_{3})+\dots) + g_{p}) + g_{1}
$$
\n
$$
\underline{h}_{2} = M_{2,1}(M_{1,p}(\dots (M_{43}g_{3}+g_{4})+\dots) + g_{1}) + g_{2}
$$
\n
$$
\underline{h}_{p} = M_{p,p-1}(M_{p-1,p-2}(\dots (M_{21}g_{1}+g_{2})+\dots) + g_{p-1}) + g_{p}
$$
\n(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  $h_p$  are calculated and the system

$$
\underline{x}_p = C_{p-p} + \underline{h}_p \tag{6.5.16}
$$

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

$$
\frac{x_1}{i} = M_{1, p} \frac{x_2}{p} + \mathcal{Q}_1
$$
\n
$$
\frac{x_1}{i} = M_{i, i-1} \frac{x_1}{i-1} + \mathcal{Q}_i, \quad i = 2, ..., p-1.
$$
\n(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  $x_{i-1}$  $i=2,3, \ldots, p-1$ . is used for the computation of  $\underline{x_i}$ , for

## 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 Ax=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

$$
\begin{bmatrix} x \\ x \\ x_B \\ \vdots \end{bmatrix} = \begin{bmatrix} 0 & M_{RB} \\ M_{BR} & 0 \\ \vdots \end{bmatrix} \begin{bmatrix} x \\ x_B \\ x_B \\ \vdots \end{bmatrix} + \begin{bmatrix} g \\ g \\ g \\ g \end{bmatrix}
$$

(6.5.18)

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

$$
\frac{x_R^{(k+1)}}{A_B} = M_{RB} \frac{x_B^{(k)}}{A_B} + \frac{q_R}{A_B}
$$
\n
$$
\frac{x_R^{(k+1)}}{B_B} = M_{BR} \frac{x_R^{(k+1)}}{A_B} + \frac{q_R}{A_B}
$$
\n(6.5.19)

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

$$
\begin{bmatrix} x \ x \end{bmatrix} = \begin{bmatrix} 0 & M_{RB} & M_{RG} \\ M_{BR} & 0 & M_{BG} \\ M_{GR} & M_{GB} & 0 \end{bmatrix} + \begin{bmatrix} x \ y \ z \end{bmatrix}
$$
(6.5.20)

and

~~ <sup>0</sup>~ MRG MR;;l ~1 r~l <sup>~</sup>MBR <sup>0</sup>~ MBO ~ .'Is = <sup>+</sup> ~ MGR M 0 MGO ~ L~J GB MOR MOB MOG 0 (6.5.21)

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

$$
\frac{x_{R}^{(k+1)}}{x_{B}} = M_{RB} \frac{x_{B}^{(k)}}{x_{B}} + M_{RG} \frac{x_{G}^{(k)}}{x_{G}} + g_{R}
$$
\n
$$
\frac{x_{B}^{(k+1)}}{x_{B}} = M_{BR} \frac{x_{R}^{(k+1)}}{x_{B}} + M_{BG} \frac{x_{G}^{(k)}}{x_{B}} + g_{B}
$$
\n
$$
\frac{x_{G}^{(k+1)}}{x_{G}} = M_{GR} \frac{x_{R}^{(k+1)}}{x_{B}} + M_{GB} \frac{x_{B}^{(k+1)}}{x_{B}} + g_{G}
$$
\n(6.5.22)

and

$$
\frac{x_{R}^{(k+1)}}{x_{B}} = M_{RB} \frac{x_{B}^{(k)}}{x_{B}} + M_{RG} \frac{x_{G}^{(k)}}{x_{G}} + M_{RO} \frac{x_{O}}{x_{O}} + g_{R}
$$
\n
$$
\frac{x_{B}^{(k+1)}}{x_{B}} = M_{BR} \frac{x_{R}^{(k+1)}}{x_{B}} + M_{BG} \frac{x_{G}^{(k)}}{x_{G}} + M_{BO} \frac{x_{O}}{x_{O}} + g_{B}
$$
\n
$$
\frac{x_{G}^{(k+1)}}{x_{G}} = M_{GR} \frac{x_{R}^{(k+1)}}{x_{B}} + M_{GB} \frac{x_{B}^{(k+1)}}{x_{B}} + M_{GG} \frac{x_{C}^{(k+1)}}{x_{G}} + g_{O}
$$
\n
$$
\frac{x_{O}}{x_{O}} = M_{OR} \frac{x_{R}^{(k+1)}}{x_{B}} + M_{OB} \frac{x_{C}^{(k+1)}}{x_{B}} + M_{OG} \frac{x_{C}^{(k+1)}}{x_{G}} + g_{O}
$$
\n
$$
(6.5.23)
$$



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

 $R$  7 x x x B 4 x x x x x B 8 3 X X X X X B 6 X X X X X R 9 X X X X B 8 X X X X X





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





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 -









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

 $\hat{\boldsymbol{\beta}}$ 

 $\ddot{\phantom{a}}$ 

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 the submatrices  $M_r$ . The total computation time is ((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 of the GS method using natural and multi-color ordering, for the matrices in Fig.6.5.4-6.



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

 $\mathbf{r}$ 

## 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.

**979**  $\mathbf{I}$ 

 $\mathbf{I}$ 

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  $Ax=y$ , where A is a full (nxn) matrix, consists of n IPS processors and a multiplexer cell. Each one of the



(a) Time-expansion

 $\sim 10$ 



 $\frac{x}{-k}$ <sup>-Ax</sup><sub>k-1</sub>'

 $k=1,2, \ldots$ 

(b) Area-expansion

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

 $-381 -$ 



Fig.6.6.3. Re-usable mvm array.

 $\blacksquare$ 

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 y; 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,  $Ax+y$ , if the multiplexer is augmented with a memory module holding vector y, and an adder. Thus, the mvips computation for a dense (nxn) matrix can be performed in 2n+l IPS cycles, and the area required is n+l 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+l) 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].

## C H A P T E R 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 word-

level 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  $\texttt{A}^\texttt{2^S}$  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), 1.e. :

$$
A^{2} = A*A; \quad A^{4} = A^{2}*A^{2}; \quad A^{8} = A^{4}*A^{4}
$$
 (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 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.



 $A_k = A_{k-1}^2$ ,  $k=1,2, .$  $\ddot{\phantom{a}}$ 

 $\mathcal{L}_{\text{max}}$  and  $\mathcal{L}_{\text{max}}$  and  $\mathcal{L}_{\text{max}}$ 



Fig.7.2.1. Time and area expansion for matrix squaring computation.

 $\sim$   $\alpha$ 

 $\sim$ 

 $\bar{\mathbf{A}}$ 



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

 $\mathcal{L}_{\text{max}}$  and  $\mathcal{L}_{\text{max}}$  and  $\mathcal{L}_{\text{max}}$ 

¥



Fig.7.2.2(b). Banded matrix squaring on a unidirectional<br>hex-array  $w_A=3$ ,  $p_A=q_A=2$ .

 $\overline{\mathbf{A}}$ 

 $-392 -$ 

(w=3) of size n=5. The array consists of  $\mathrm{w}^2$  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  $n>>w_{n}$ , and  $w_{n}$ , $w_{n}$ 2 the bandwidths of matrices A,  $A^{2}$  we have  $w_n^2 = 2w_n - 1$ , and in general

$$
w_{A^{2}}i = 2^{i}w_{A}^{-(2^{i}-1)}, i=0,1,\ldots,s
$$
 (7.2.2)

This also means that in order to calculate  $\texttt{A}^\texttt{2}$  an  $\texttt{array}$  of  $\rm{w}^2$  .  $_2$ (s-1) IPS cells is required. It is obvious then that suc-A cessive 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  $w_{\text{max}}^2$ , then the maximum number of squaring steps s is given by the relation

$$
2^{s-1}w_A^{-(2^{s-1}-1)} \leq w_{\text{max}} \tag{7.2.3}
$$

The squaring of a matrix with unequal semibands is illustrated in Fig.7.2.3, for  $w_n = 4$ ,  $p_n = 3$  and  $q_n = 2$ . 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

$$
\begin{cases}\n\frac{1}{2} \mathbf{a}^{2i} = \frac{1}{2} \mathbf{a}^{2i} + \frac{1}{2} \mathbf{a}^{2i} - 1, \quad i = 0, 1, \ldots, s \\
\frac{1}{2} \mathbf{b}^{2i} = \frac{1}{2} \mathbf{b}^{2i} - \frac{1}{2} \mathbf{b}^{2i} \\
\frac{1}{2} \mathbf{b}^{2i} = \frac{1}{2} \mathbf{b}^{2i} - \frac{1}{2} \mathbf{b}^{2i} \\
\frac{1}{2} \mathbf{b}^{2i} = \frac{1}{2} \mathbf{b}^{2i} - \frac{1}{2} \mathbf{b}^{2i} \\
\frac{1}{2} \mathbf{b}^{2i} = \frac{1}{2} \mathbf{b}^{2i} - \frac{1}{2} \mathbf{b}^{2i} \\
\frac{1}{2} \mathbf{b}^{2i} = \frac{1}{2} \mathbf{b}^{2i} - \frac{1}{2} \mathbf{b}^{2i} \\
\frac{1}{2} \mathbf{b}^{2i} = \frac{1}{2} \mathbf{b}^{2i} - \frac{1}{2} \mathbf{b}^{2i} \\
\frac{1}{2} \mathbf{b}^{2i} = \frac{1}{2} \mathbf{b}^{2i} - \frac{1}{2} \mathbf{b}^{2i} \\
\frac{1}{2} \mathbf{b}^{2i} = \frac{1}{2} \mathbf{b}^{2i} - \frac{1}{2} \mathbf{b}^{2i} \\
\frac{1}{2} \mathbf{b}^{2i} = \frac{1}{2} \mathbf{b}^{2i} - \frac{1}{2} \mathbf{b}^{2i} \\
\frac{1}{2} \mathbf{b}^{2i} = \frac{1}{2} \mathbf{b}^{2i} - \frac{1}{2} \mathbf{b}^{2i} \\
\frac{1}{2} \mathbf{b}^{2i} = \frac{1}{2} \mathbf{b}^{2i} - \frac{1}{2} \mathbf{b}^{2i} \\
\frac{1}{2} \mathbf{b}^{2i} = \frac{1}{2} \mathbf{b}^{2i} - \frac{1}{2} \mathbf{b}^{2i} \\
\frac{1}{2} \mathbf{b}^{2i} = \frac{1}{2} \mathbf{b}^{2
$$

A<sup>-</sup> Thus, for the computation of A<sup>25</sup> 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  $w'_{A}$  =  $2max(p_{A},q_{A})-1$ , and the array consists of  $w_A^2$  IPS cells. The total computation time for the calculation of  $A^2$  in both Fig.7.2.2,3 is

$$
n + \sum_{i=0}^{s-1} ((2^{i}w'_{A} - (2^{i}-1))
$$
 (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,  $k=1, 2, ..., p-1$  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<sup>25</sup> 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<br>bandwidth  $w=5$ ,  $p=q=3$ .

397  $\mathbf{1}$ 

 $\mathbf{I}$ 

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<sup>25</sup> a total area of

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

IPS cells is required, and the total time is

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

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





 $-399 -$ 

 $\ddot{\phantom{0}}$ 



 $\sim$ 

 $\sim 10^{11}$  km  $^{-1}$ 

 $\sim$ 

 $\mathcal{L}_{\mathrm{eff}}$ 

 $\sim$ 

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

 $\sim$ 

 $\sim$ 

 $\pmb{\mathsf{I}}$ 400  $\mathbf{I}$ 



Fig.7.2.7. Re-usable matrix multiplication array, n=3.

 $\sim 10^{11}$  km

 $\frac{1}{2}$ 

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  $\overline{\text{n}^2}$  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 (l=on, O=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-1IPS cycles.



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

# 7.3 SYSTOLIC ITERATIVE SOLUTIONS OF LINEAR SYSTEMS USING MATRIX POWERS\*

Consider the. system of n linear equations Ax•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], [1481 :

$$
\begin{bmatrix} 1 & \mathbf{0}^T \\ \frac{\mathbf{b}}{\mathbf{a}} & -\mathbf{a} \end{bmatrix} \cdot \begin{bmatrix} 1 \\ \frac{\mathbf{c}}{\mathbf{a}} \end{bmatrix} = \mathbf{0}
$$
 (7.3.1)

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

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

where

$$
\mathbf{y}^{\mathrm{T}} = [1 | \mathbf{x}^{\mathrm{T}}] \tag{7.3.3}
$$

and

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

with y and 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

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

with

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

where  $q_i$ ,  $I_i$ ,  $M_i$  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,  $R^2$ ,  $R^4$  , ...,  $R^2$  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  $r^{2^K}$  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

 $\begin{bmatrix} 1 & \frac{\mathbf{0}^{\mathrm{T}}}{2} \\ \mathbf{y} & \mathbf{c} \end{bmatrix}^2 = \begin{bmatrix} 1 & \frac{\mathbf{0}^{\mathrm{T}}}{2} \\ \frac{\mathbf{y} + \mathbf{c} \mathbf{y}}{2} & \frac{\mathbf{c}^2}{2} \end{bmatrix}$  $(7.3.8)$ 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  $\underline{x} = \underline{y} + c\underline{y} + c^2 (\underline{y} + c\underline{y}) + c^4 (\underline{y} + c\underline{y} + c^2 (\underline{y} + c\underline{y})) + \dots$ 

 $= \chi + cy + c^2 \chi + \dots$  (7.3.9)  $= \chi + C(\chi + C(\chi + \ldots +C\chi) \ldots)$ .

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 used to implement the same iterative methods, if full 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

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

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<br>step for a banded matrix  $C$ ,  $w=5$ ,  $p=q=3$ .

 $\mathbf{L}$ **410**  $\blacksquare$ 

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  $A$ ,  $\underline{b}$ , (see Fig.6.3.11).

# 7.4 SYSTOLIC DESIGNS FOR EIGENVALUE-EIGENVECTOR COMPUTATION USING MATRIX POWERS

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

$$
|\lambda_1| > |\lambda_2| \ge |\lambda_3| \ge \dots \ge |\lambda_n| \tag{7.4.1}
$$

Now let A operate repeatedly on a vector  $u$ , which is a</u> linear combination of the eigenvectors of A, (section 2.4):

$$
\underline{u} = c_1 \underline{x}_1 + c_2 \underline{x}_2 + \dots + c_n \underline{x}_n \tag{7.4.2}
$$

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

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

For k sufficiently large, we have,

$$
A^k \underline{u} = \lambda_1^k c_1 \underline{x}_1, \quad c_1 \neq 0 \tag{7.4.4}
$$

Thus, the dominant eigenvalue  $\lambda_1$  is calculated as

$$
\lambda_1 = (A^k \underline{u})_i / (A^{k-1} \underline{u})_i, \quad i=1,2,\ldots,n
$$
 (7.4.5)

while  $A^{k}u$  is proportional to the corresponding eigenvector  $x_1$ , providing no overflow occurs. The convergence rate is determined by

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

where  $\varepsilon$  is a specified tolerance. The Power Method algorithm with a normalising strategy included can be formulated as follows [298] : Given  $\chi_0$ , an arbitrary vector, and let the

sequences  $z_k$ ,  $y_k$  be defined by the equations,  $z_{i+1}$  = A<sub> $y_1$ </sub>  $Y_{j+1} = Z_{j+1}/\mu(Z_{j+1}), \quad j=0,1,\ldots,k-1$ with (7.4.7)

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

For k sufficiently large,

$$
\Sigma_k \stackrel{\approx}{=} \frac{x_1}{\mu} \frac{(x_1)}{x_1} \tag{7.4.9}
$$

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  $\lambda_1$  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  $\mu(\underline{z}_{\dot 1})$ 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  $z_j$  is either too big or too small. On the other hand, (7.4.8) imposes a considerable delay in the computations, since  $\underline{z}_{i+1}$  cannot be computed before all elements of  $\underline{z}_i$  are known.

The convergence rate in  $(7.4.6)$  can be improved, if, instead of forming the sequence  $A^{k}y$ , a sequence of matrix powers is constructed directly [298]. The Matrix Squaring method has the advantage that when  $A^j$  is computed,  $A^2$ <sup>j</sup> is obtained in one mmm. Hence the sequence  $A, A^2, A^4, \ldots, A^{2^S}$ one is easily built up. For sufficiently large s [ 56] :

$$
A^{2^{S}} = \lambda_1^{2^{S}} c_1 \underline{x}_1 \underline{x}_1^{T} . \qquad (7.4.11)
$$

Each column of  $a^{2^S}$  is proportional to  $\underline{x}_1$ , each row is proportional to  $x_1^T$ . To obtain the eigenvalue  $\lambda_1$  any column of  $A^{\text{2}}$  can be multiplied by A. Then (7.4.5) can be between the original column of  $\text{a}^2^\texttt{S}$ and that of  $aa^2$  . applied

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

$$
|\lambda_2/\lambda_1|^2 \leq \varepsilon \tag{7.4.12}
$$

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  $\lambda_2/\lambda_1$ , but, a simple estimation can be derived [1831

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

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

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

where

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

as is shown in [148],[298]. Thus, the dominant eigenvalue is obtained by a series of r square root extractions;  $x_1$  is collected as before. The rate of convergence is now determined by

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

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):

 $Y_{j+1} = Z_{j+1}/\mu(Z_j)$ 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 a pipeline as in Fig.7.4.1, in n+k(w+p) cycles, 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.

 $\ddot{\phantom{1}}$ 



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

 $\mathbf{I}$ ....  $\frac{8}{10}$  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

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

cycles and with area requirements of

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

 $- 422 -$ 





 $- 423 -$ 

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+l.

The final mmm, i.e.  $AA^2$  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  $n^2$  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 followed by  $k$  degenerate mmm's between B=A $^{2^{\Gamma}}$  and [ $\underline{x_i}$ |0], squaring steps are i=O,l, ... ,k. Thus only n+l 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

$$
\frac{dx}{dt} = A(t) \underline{x}_0
$$
 (7.5.1)

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

$$
\underline{x} = e^{\underline{A}} \underline{x}_0, \qquad (7.5.2)
$$

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  $e^{\hat{A}}$  in (7.5.3) can be the basis for a simple algorithm calculating the exponential of a matrix using Taylor series approximation techniques. Thus,

$$
e^{A} = T_{k}(A) = \sum_{i=0}^{k} (\frac{1}{i!} A^{i}).
$$
 (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  $\frac{||A||}{2^j} \le 1$ . (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 c, and magnitude of  $\left\vert \left\vert \mathsf{A}\right\vert \right\vert_1$ , Table 7.5.1 summarizes the optimum (k,j) associated with  $[\texttt{T}_k(\texttt{A}/2^{\texttt{J}})]^{\texttt{Z}^{\texttt{J}}}$ , [210].

### 7.5.1 SYSTOLIC DESIGNS

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

$$
S_0 = X_k
$$
  
\n
$$
S_i = AS_{i-1} + X_{k-i} , for i=1,2,...,k,
$$
 (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,



Table 7.5.1. Optimum  $(k, j)$  for given  $\varepsilon$  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  $2n(k+j+2)+n-1IPS$  cycles. A soft-systolic simulation program in OCCAM is given inA.4.10.

A pipeline design calculating the exponential of a 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<sup>i</sup>-1), where w is the bandwidth of matrix A. This is because the bandwidth of  $A/Z^{\frac{1}{J}}$  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



 $s_k = s_{k-1} + ax_{k-1}$  $k = 1, 2, \ldots$  $\mathbb{R}^2$  $i = 0, 1, ..., k$ 



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



 $x_i = \frac{1}{i!} I$ ,  $R_i = T_i (A/2^j) - X_{k-i}$ ,  $S_k = T_k (A/2^j)$ ,  $S_k^2 = e^A$ 

 $\label{eq:2.1} \frac{1}{\sqrt{2\pi}}\int_{0}^{\infty}\frac{1}{\sqrt{2\pi}}\left(\frac{1}{\sqrt{2\pi}}\right)^{2\alpha} \frac{1}{\sqrt{2\pi}}\frac{1}{\sqrt{2\pi}}\int_{0}^{\infty}\frac{1}{\sqrt{2\pi}}\frac{1}{\sqrt{2\pi}}\frac{1}{\sqrt{2\pi}}\frac{1}{\sqrt{2\pi}}\frac{1}{\sqrt{2\pi}}\frac{1}{\sqrt{2\pi}}\frac{1}{\sqrt{2\pi}}\frac{1}{\sqrt{2\pi}}\frac{1}{\sqrt{2\pi}}\frac{1}{\sqrt{2\pi}}\frac{$ 

Fig.7.5.2. Iterative array configuration.

 $\Delta \phi$  and  $\Delta \phi$  are  $\Delta \phi$ 

 $- 430 -$ 

 $\label{eq:2.1} \frac{1}{\sqrt{2}}\int_{\mathbb{R}^3}\frac{1}{\sqrt{2}}\left(\frac{1}{\sqrt{2}}\right)^2\frac{1}{\sqrt{2}}\left(\frac{1}{\sqrt{2}}\right)^2\frac{1}{\sqrt{2}}\left(\frac{1}{\sqrt{2}}\right)^2\frac{1}{\sqrt{2}}\left(\frac{1}{\sqrt{2}}\right)^2.$ 



 $\sim 10^{11}$  km

 $\hat{\mathcal{A}}$ 

the contract of the contract of the contract of

 $\mathcal{L}^{\text{max}}_{\text{max}}$  , where  $\mathcal{L}^{\text{max}}_{\text{max}}$ 

 $\label{eq:2.1} \frac{1}{\sqrt{2}}\int_{\mathbb{R}^3}\frac{1}{\sqrt{2}}\left(\frac{1}{\sqrt{2}}\right)^2\frac{1}{\sqrt{2}}\left(\frac{1}{\sqrt{2}}\right)^2\frac{1}{\sqrt{2}}\left(\frac{1}{\sqrt{2}}\right)^2\frac{1}{\sqrt{2}}\left(\frac{1}{\sqrt{2}}\right)^2\frac{1}{\sqrt{2}}\left(\frac{1}{\sqrt{2}}\right)^2\frac{1}{\sqrt{2}}\left(\frac{1}{\sqrt{2}}\right)^2\frac{1}{\sqrt{2}}\left(\frac{1}{\sqrt{2}}\right)^2\frac{1}{\sqrt{$ 

 $\mathcal{F}^{\text{max}}_{\text{max}}$ Fig.7.5.3. Pipeline configuration for  $k=2$ ,  $j=1$ ,  $w=3$ .

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 conelude 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

$$
n+w+\sum_{i=1}^{k-1} (iw-(i-1)) + \sum_{i=0}^{j-1} ((2^jw'-(2^i-1))
$$
 (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,  $w_A=4$ ,  $p_A=3$ ,  $q_A=2$ ,  $-3$ ,  $p_B = 2$ ,  $q_B = 2$ .



Fig.7.5.4(b). Banded matrix multiplication,  $w_n=3$ ,  $p_n=2$ ,  $q_n=2$ ,  $w_B=4, p_B=3, q_B=2.$ 

- **434** -

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  $w_{\text{max}}^2$  then, we have

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

A soft-systolic simulation program for the matrix polynomial pipeline is given in  $A.A.11.A$  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  $Ax=b$ , 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

 $A^{-1} = B(I-M)^{-1} = B(I+M+M^{2}+M^{3}+\dots)$  $(7.6.2)$ with  $\vert \vert M \vert \vert \langle 1 \vert [111] \rangle$ . 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  $S_k$ :

$$
S_k = I + M + (I + M(I + ... M(I + M1)...))
$$
\n(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:

$$
S_2i+1 = S_2i + S_2iM^2 \t i=0,1,2,\ldots,1
$$
 (7.6.5)

(iii) Another procedure based on successive squarings can be derived from (7.6.5):

$$
S_k = (I + M)(I + M^2)(I + M^4) \dots (I + M^{2^1})
$$
 (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:

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

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  $tr(A^{\dot{1}})$ ,  $i=1,2,...,k$ , 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

$$
P_0 = A, Q_0 = I
$$
  
\n
$$
P_{k+1} = \frac{1}{2} (P_k + Q_k^{-1})
$$
  
\n
$$
Q_{k+1} = \frac{1}{2} (Q_k + P_k^{-1}), k = 0, 1, 2, ...
$$
\n(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):

 $Z_{\mathbf{k}} = \mathbf{I} - \mathbf{T}_{\mathbf{k}} \mathbf{A} \mathbf{T}_{\mathbf{k}}$ 

 $T_{k+1} = T_k(I + \frac{1}{2}k), k = 0, 1, 2, ...$ (7.6.9) where  $\texttt{T}_{0}$  is an initial approximation of  $\texttt{A}^{-1/2}.$  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

 $cos(2A) = 2cos(A)^{2}-I$ 

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

$$
S_k = 2S_{k-1}C_{k-1}
$$
  
\n
$$
C_k = 2C_{k-1}^2 - 1, k = 1, 2, ..., j
$$
 (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 approximat ion

$$
log(A) = (A-1) - \frac{(A-1)^2}{2} + \frac{(A-1)^3}{3} - \dots
$$
 (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  $||A|| < 1$ .

## C H A P T E R 8

## OPTICAL SYSTOLIC ALGORITHMS

#### 8.1 INTRODUCTION

The optical implementation of systolic algorithms was originally introduced in [58), where asystolic 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 systolie 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 ( $w_n=w_{p}=3$ ) 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.B.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 the R+F matrix multiplication algorithm are explained with the help of Fig.B.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.B.2.3,4 consists of two 2-d transducer arrays, each with  $w_{\text{A}}w_{\text{B}}$  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 of the mmm algorithm as illustrated in Fig.B.2.2; i.e. the optical implementation follows the same dataflow principles as the hex-connected VLSI systolic array, although the data 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.

**050** 



Fig.8.2.4. Matrix multiplication optical processor.

151

 $\mathbf{I}$ 

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  $\frac{1}{2}$  c  $\frac{1}{2}$  c  $\frac{1}{55}$  and c<sub>11</sub> 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  $w_{A}w_{B}$  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=S) 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

<sup>\*</sup> 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  $a_{i,j}$  in a certain pixel of the shift register array in a certain time unit the modification is  $a_{i,i} = a_{i,i} + c$ , where c is a quantity proportional to the intensity of the incident light beam collected by 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.

89\$  $\mathbf{L}$ 



 $\sim$ 

 $\sim 10^{-11}$ 



## DETECTORS SHIFT REGISTERS

Fig.8.3.2. Details of optical processor.

65b  $\blacksquare$ 

 $\mathbf{r}$ 

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 to the transducer planes. Since the entries of L 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 more. The operations performed by each part of the optical system are also indicated in the same figure: note that only one pixel operates as a pure !PS processor. In the general case of a banded matrix with bandwidth  $w = p+q-1$  we have (p-l)x(q-1) pixels performing !PS computations; p-1 cells perfoming negations, i.e. producing  $-u_{i,j}$ ; q-1 cells performing simple multiplications, i.e. producing the elements of matrix L, and finally one cell producing  $u_{i,i}^{-1}$ . 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.

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 !PS 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<sub>55</sub> enters the detector array. In the third step,  $a_{21}$  and  $a_{12}$  are available and  $1_{21}$ = $a_{21}$ 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  $1_{21}$  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.  $\bar{z}$ 

 $\bar{z}$ 



Fig.8.3.4. (continued)

 $\hat{\boldsymbol{\gamma}}$ 

 $\hat{\vec{r}}$ 

are produced;  $1_{45}$  and  $-u_{54}$  are calculated and the transducer plane produce light beam with intensity  $1_{21}(-u_{12})$  which is added to  $a_{22}$  to produce  $a_{22} = a_{22}+1_{21}(-u_{12})$ .

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  $Ax = b$  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 b and the output of the shift register array (indicated as  $y)$ .



Fig.8.3.5. Optical processor for triangular system solution.

... "' Ul

 $\mathbf{L}$ 

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.8.3.7. 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 B. An alternative way is to see matrix C as a summation of matrices formed by outer products between columns of A and rows of B [ 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  $F_2$  that is added upon  $F_1$ . Thus, after n steps,  $C = F_1 + F_2 + ... + F_n^2$ .

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<sub>B</sub>, where w<sub>A</sub> and w<sub>B</sub> 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

<sup>\*</sup> 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.



Fig.8.4.2. Banded matrix multiplication using outer products.

 $\mathbf{I}$ 472

 $\mathbf{I}$ 

the computation are located within an 'active window' of size  $w_{a}w_{a}$ . 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



 $\bar{\beta}$ 



 $\mathbf{r}$ 

474

 $\mathbf{I}$ 

 $a_{31}$  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  $\mathsf{n}^2$  pixels. The detector array is now augmented with a  $\,$  plane  $\,$ of shift registers so that the resulting 'active window' of matrix c can be moved along the detector array as indicated in Fig.8.4.2. Thus, the elements of matrix C are not stored 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.S. Each full

 $- 475 -$ 









1: shift in new row-column vectors<br>' 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<br>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, B 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 as full mmm computations, can be readily applied on the outer product processor.

In comparison with the outer product processor discussed in ( 13), the proposed system reduces significantly the hardware requirements especially when  $n>>w_{\text{A}}$ ,  $w_{\text{B}}$ ; furthermore the system is size-independent, i.e it works for any n given a maximum bandwidth; finally, no unloading cycles are

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 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)

 $\begin{bmatrix} 1 & 0 & 0 \ 1 & 1 & 0 \ e_1 & 1 & 0 \ e_2 & 0 & 1 \end{bmatrix} + \begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} & a_{13}^{(1)} \ a_{21}^{(1)} & a_{22}^{(1)} & a_{23}^{(1)} \ a_{31}^{(1)} & a_{32}^{(1)} & a_{33}^{(1)} \end{bmatrix} = \begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} & a_{13}^{(1)} \ 0 & a_{22}^{(2)} & a_{23}^{(2)} \ 0 & a_{32}^{$  $A^{(1)}$  $_{a}^{(2)}$  $E_{1}$  $e_1 = -\frac{a_{21}^{(1)}}{a_{11}^{(1)}}$ ,  $e_2 = -\frac{a_{31}^{(1)}}{a_{11}^{(1)}}$ . where  $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & e_3 & 1 \end{bmatrix} + \begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} & a_{13}^{(1)} \\ 0 & a_{22}^{(2)} & a_{23}^{(2)} \\ 0 & a_{32}^{(2)} & a_{33}^{(2)} \end{bmatrix} = \begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} & a_{13}^{(1)} \\ 0 & a_{22}^{(2)} & a_{23}^{(2)} \\ 0 & 0 & a_{33}^{(3)} \end{bmatrix}$  $_{A}$ (2)  $_{\rm A}$  (3)  $E_{2}$ where  $e_3 = -\frac{a_{32}^{(2)}}{a_{33}^{(2)}}$  $\begin{bmatrix} 1 & 0 & 0 \\ x_{21} & 1 & 0 \\ x_{31} & x_{32} & 1 \end{bmatrix}$  =  $\begin{bmatrix} 1 & 0 & 0 \\ -e_1 & 1 & 0 \\ -e_2 & -e_3 & 1 \end{bmatrix}$ L  $\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}$  =  $\begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} & a_{13} \\ 0 & a_{22}^{(2)} & a_{23}^{(2)} \\ 0 & 0 & a_{33}^{(2)} \end{bmatrix}$ Ŭ

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  $\mathtt{A^{(2)}}$  and adding the resulting-matrix to  $\mathtt{A^{(2)}}$  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  $E_{i}^{*}$ , (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  $a_{22}^{(1)}$  $-2^{21}$  $\mathbf{o}$  $a_{21}$  $-2$ <sub>31</sub>  $\mathbf{1}$  $\frac{1}{2}$ a $\frac{1}{32}$  $a_{35}$  $\begin{bmatrix} a_{31} \\ a_{32} \end{bmatrix}$  $a_{32}$   $a_{33}$  $\circ$  $\blacksquare$  $a_{42} a_{43}$  $44 \begin{array}{c} a_4 \ a_5 \end{array}$  $\mathbf{1}$  $a_{42}$  $\mathbf{1}$  $\begin{bmatrix} 1 \\ + \\ + \\ + \\ + \\ + \\ 22 \end{bmatrix}$  $| 0$  $\frac{1}{2}$  $a_{34} a_{35}$  $\begin{array}{c|c}\n\diagup & a_{42} & a_{42}\n\end{array}$  $a_{45}$  $\sqrt{10}$  $\mathbf{L}$  $a_{54}$  $\rm{^{a}_{53}}$  $^{\tt a}$ 55

Fig.8.4.7. Banded LU decomposition using outer products.

 $\mathbf{I}$ 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.B.4.7 the superscripts indicate the number of modifications each element undergoes.

The outer product calculation between the first column  $E_1^*$  and the first row of A<sup>(1)</sup> (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 light beams that travel in the directions shown by the arrows. For example, the beam with intensity proportional to e<sub>2</sub> passes through the transducer pixel which modulates its intensity in proportion to  $a_{11}^{(1)}$ , yielding a beam with intensity proportional to  $e_2a_{11}^{(1)}$ . 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:  $E_{1}^{*}(1)_{+A}(1)_{=A}(2)$ .

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

- **483** -





1-d input transducers

2-d detector.array

Fig.8.4.8. LU decomposition using outer product.





J.

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

calculating the first column of matrices  $E_i^*$  and L (see Fig.8.4.6)<sup>\*</sup> An outer product is then calculated between the first column of  $E_i^*$  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<sub>11</sub> a<sub>12</sub> a<sub>13</sub>] form the first row of U and [a<sub>11</sub> a<sub>21</sub> a<sub>31</sub>]<sup>T</sup> are involved in the calculation of the first column for L,  $E_1^{\pi}$ . The second phase calculates the coefficients for  $L, E_1$ . The third phase performs the outer product and the addition

<sup>\*</sup> The computation of the columns of  $E_{J}^{*}$  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.  $a_{22}^{(1)}$  =  $a_{32}^{(1)} = a_{32} - l_{31} a_{12}$ 

Now the processor is ready to shift out another row of U and another column for L and to restart the calculations. 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 subtractiondivision and the data transfers. Notice 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.

## 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. As is shown in Fig.8.5.1 the GE algorithm can be performed in n-1 major steps, each step involving an 'outer product plus addition' computation. For example, in the first step of GE  $[a_{11} \ a_{21} \ a_{31}]^T$  determines the multipliers  $\begin{bmatrix} 1 & e_{21} & e_{31} \end{bmatrix}^T$ ; 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 'active window' of size  $(q-1)x(p-1)$  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.B.S.l. 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.B.S.l 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.B.S.l. 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 b.

In both the algorithms described for the solution of linear systems there are two 'active windows': one for 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.B.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.B.5.3.

The GJ algorithm can be used for the inversion of 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.B.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



τ

τ

т



 $\mathbf{x}$ 

 $\mathbf{x}$ 

 $\mathbf{x}$ 



 $\mathbf{x}$ 

 $\mathbf{x}$ 

 $\mathbf{x}$ 







 $\pmb{\times}$ 

 $\mathbf{x}$  $\mathbf{x}$ 

n-1 modification steps





Fig.8.5.2. Gauss Elimination for A,  $\underline{b}$ .

 $\bar{\mathbf{x}}$  $\pmb{\times}$  $\pmb{\times}$  $\pmb{\mathsf{x}}$  $\pmb{\mathsf{x}}$ |x  $\pmb{\times}$ |x  $\pmb{\mathsf{x}}$ ×.  $\pmb{\mathsf{x}}$  $\boldsymbol{\mathsf{x}}$ 











 $\pmb{\mathsf{x}}$  $\pmb{\times}$  $\pmb{\mathsf{x}}$  $\mathbf{x}$  $\mathbf{x}$  $\mathbf{x}$ 



n-1 modification steps



division

Fig.8.5.3. Back substitution for A,  $\underline{b}$ .

 $-496 -$ 

 $\mathbf{x}$  $\mathbf{x}$  $\mathbf{x}$  $\mathbf{x}$  $\pmb{\mathsf{x}}$ þ, Ŀ  $\boldsymbol{\mathsf{x}}$ ىر  $\mathbf{x}$  $\bar{\mathbf{x}}$  $\bar{\mathbf{x}}$  $\bar{\mathbf{x}}$  $\pmb{\mathsf{x}}$  $\pmb{\mathsf{x}}$  $\mathbf{x}$  $\mathbf{x}$  $\mathbf{x}$ 















n-1 modifications





division

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

 $-497 -$ 

















n-1 modifications





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 b' 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.8.5.6. Optical processor for Gauss Elimination.

 $\ddot{\phantom{0}}$ 

 $\sim 10^{-11}$ 

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-l,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 b, 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 b 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 l,q-1 pixels. 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 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 b 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.B.S.B. The outer product processor for the modification of b 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.

 $\sim 10^6$ 

 $\hat{\mathcal{L}}$ 

 $\sim$   $\sim$ 

# 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 Convolution (DMAC) method. This algorithm is outlined, 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, so that 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 processer, however, forces a pipelined computation to be performed at the speed of the slowest component, i.e. the digital electronic processing elements. Thus, an interesting 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

<sup>\*</sup> This technique may be the cause of potential problem, in power budgets, if the bondwidth 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 mvm 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 for some simple optical systolic algorithms. Two types of banded mvm are presented (A.S.l and A.5.2), one using outer product concepts and the other using simple space integration (inner product). These architectures are 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.

#### C H A P T E R 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 is an introduction to specific topics: solution of polynomial 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 subsequnent chapters. Next there is an overview of some 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 the soft-systolic simulation method, using OCCAM. Finally, 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 the framework initially proposed so that it can accommodate the different classes of systolic algorithms: hard (traditional, direct VLSI implementation); hybrid (programmable, 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

in detail: Graeffe and Bernoulli. In the study of these methods transformational techniques are exemplified: retiming technique; bidirectional array transformed to architecture; area-time expansion. Further, a general ring systolic 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 quindiagonal 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 fault-tolerance. The same technique can be extended to 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 with the updating technique, and other factorization 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 the design of a programmable systolic array performing all 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 normalisation problems. However, the advances in technology make possible the consideration of these algorithms that 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 systolic algorithms, mainly based on Gauss Elimination. Finally, some of the problems of the optical implementation of systolic algorithms are briefly discussed: first the 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 given inputs. Thus, systolic architectures are considered as networks 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 algorithms 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 outline a class of corresponding algorithms. The algorithms 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, or as soft-systolic structures performable on a generalpurpose 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 processors. The Systolic Ring may process many polynomials in 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 multi-coloring methods, as well as in the computation of and

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 in the case of iterative structures based on mvm or mmm arrays, while the storage of intermediate results is useful in the two-stage recurrences for the computation of matrix 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 optical processor discussed in [13], since it requires less area and is problem size-independent. A number of Outer Product Processors can operate in parallel to perform multiple outer products using the same coefficient marix, as is the case in Gauss elimination algorithms. Further, a feedback 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 architectufes, 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 of certain differential equations; they correspond to 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 number of areas of computing, in addition to the ones 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

- **524** -

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 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.

# R E F E R E N C E S

- 1. Abdel Kader, A.A., "OCSAMO: A Systolic Array for matrix operations", in Proc. CONPAR 86, Springer-verlag, 1986, pp. 319-328.
- 2. Abushagur, M.A.G., caulfield, H.J., "Speed and convernsushugus, nintes, orderal computers", Optical Engineering,  $1987(26)$ , pp.  $22-27$ .
- 3. Adams, L., "Iterative algorithms for large sparse linear system on parallel computers", Ph.D. thesis, Virginia University, 1982.
- 4. Alves Marques, I., cunha, A., "Clocking of cults", in "VLSI Architecture", Randell, (eds.), Prentice-Hall, 1983, pp. 165-178. VLSI cir-B., et al.
- 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.
- 6. Ang, P.H., Morf, M., "Concurrent Array Processor for Fast Eigenvalue Computations", Proc. IEEE ICASSP 1984, pp. 34A.2.1-34A.2.4.
- 7. Annaratone, M., et al., "Extending the CMU Warp Machine with a Boundary Processor", Proc. SPIE, RTSP VIII, 1985(564), pp. 56-65.
- 8. 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.
- 10. Arrathoon, R., "Digital Optical Computing: possibili-<br>ties and pitfalls", Proc. SPIE, RTSP VIII, 1985(564), pp. 108-118.  $1985(564)$ ,
- 11. Arrathoon, R. et al., "Digital Convolution and Correlation with Electrooptic Bragg Processors", Proc. SPIE,

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

- 12. 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.
- 13. Athale, R.A., Outer-Product 931-941. Lee,  $J.N.$ , Concepts", "Optical Processing using Proc. IEEE, 1984(72), pp.
- 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 Analysis", John Wiley, 1978. Introduction to Numerical
- 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.
- 18. Barth, B., Martin, R.S., Wilkinson, J.H., "Calculation of the Eigenvalues of a Symmetric Tridiagonal Matrix by<br>the Method of Bisection", Numer. Math., 1967(9), pp. 386-393.
- 19. Bekakos, M.P, puters and Thesis, Dept. of Computer Studies, LUT, 1986. "A study of Algorithms for Parallel VLSI Systolic Processor Arrays", Com-Ph.D
- 20. Bekakos, M.P., Evans, D.J., "The Exposure and Exploita- tion of Parallelism in Fifth Generation Computer Sys- tems", 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", Parallel Computing, 1988, in press.
- 22. Bellman, R., "Introduction McGraw-Hill, 1960. to Matrix Analysis",
- 23. Bennet, J.M., "Triangular Factors of Modified Matrices", Numer. Math., 1965(7), pp. 217-221.
- 24. Bertossi, A.A., Bonuccelli, M.A., "A VLSI Implementations of the Simplex Algorithm", IEEE, Trans. on Computers, 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 sys- tolic 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.
- 30. 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", SIAM J. on Scientific and <u>Statistical</u> Computing, 1984(5), pp. 95-T04.
- 35. Brenner, K.H., Lohmann, A.W., "The Digital Optical Computing Program at Erlangen", Springer-Verlag, 1986, pp. 69-75. Proc. CONPAR '86,
- 36. Brent, R.P., Kung, H.T., Time GCD Computation", 1984(C-33), pp. 731-736. "Systolic Arrays IEEE, Trans. on for Linear Computers,
- 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", 1n "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", Proc. Cambridge Philosophical Society,  $1924(22)$ , 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.
- 44. 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, 0., "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 Systollc 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 us1ng 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 optisystolic processing architectures", Optical Engineering, 1985(24), pp. 59-64.
- 50. Casasent, D., "Acoustooptic Linear Algebra Processors: Architectures, Algorithms and Applications", Proc. IEEE, 1984(72), pp. 831-849.
- 51. Casasent, D., Ghosh, A., "Optical Linear Algebra",

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 Number of 102-108. Nonlinear Matrix Equations using. a Fixed steps", Proc. SPIE, RTSP VII, 1984(495), pp.
- 54. Casasent, D., Taylor, B.K., Processor", Proc. SPIE, RTSP 149. "Optical Finite Element VIII, 1985(564), pp. 139-
- 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", Optics Communications, 1981(40), pp. 86-90.
- 59. Chandy, K.M., Misra, J., "Systolic algorithms as programs'', Distributed Computing, 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'', International J. of Computer Mathematics, 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", Quart. J. of Mechanical and Applied  $\frac{\text{Maths}}{\text{Maths}}$ , 1948(1), pp. 145-148.
- 66. Conrad, M., "On design principles for a Molecular Computer", Communications of the ACM, 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", Intern. J. Computer Math, 1988, in press.
- 69. Csanky, L., "Fast Parallel Matrix Inversion Algo-<br>rithms", SIAM J. of Computing, 1976(5), pp. 6i8-623.
- 70. Culik, II,K., Gruska, J., Salomaa, A., Trellis Automata, Part I and II", Intern. Math., 1984(15), pp. 195-212; 1984(16), pp. "Systolic  $\frac{J}{3-22}$ .
- 71. Culik, II,K., Yu, S., "Fault-tolerant schemes for some systolic systems", Intern. J. Computer Math., 1987(22), pp. 13-42.
- 72. Dahlquist, G., Bjork, A., "Numerical Methods",<br>Prentice-Hall, 1974.
- 73. Danielsson, P.E., "Serial/Parallel Convolvers", IEEE, Trans. on Computers, 1984(C-33), pp. 652-667.
- 74. Davis, G.J., "Numerical Solution of a Quadratic Matrix Equation", SIAM, J. Sci. Stat. Comput., 1981(2), pp. 164-175.
- 75. Davis, J.A., Jones, K.D., Lilly, R.A., "Improved system for binary multiplication by optical convolution", Optical 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, (eds.), North-Holland, 1986, pp. 295-316. m., et al.
- 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 tures for VLSI signal processing", architecin "VLSI

 $\alpha$ 

 $\alpha$  .

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.
- 80. 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.
- 82. 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<br>for Computing the Singular Value Decomposition using for Computing the Singular Value Decomposition using<br>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<br>Engineering, 1986(25), pp. 38-43. Engineering,  $1986(25)$ , pp. 38-43.
- 86. Dunway, D.K., "Calculation of the zeros of a real polynomial through factorization using Euclid's algorithm", SIAM J. Numer. Anal., 1974(11), pp. 1087-1104.
- 87. Eichman, G., "Systolic Arrays for Eigenvalue Computa- tion", Proc. SPIE, RTSP VIII, 1985(564), pp. 39-44.
- 88. El-Amawy, A., Porter, W.A., Aravena, J.L., "Array<br>architectures for iterative matrix calculations", <u>Proc</u>.<br><u>IEEE</u>, 1987(134-E), pp. 149-154.
- 89. Ersoy, o., "Semisystolic Array Implementation of Circular, Skew Circular and Linear Convolutions", IEEE, 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

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'<br>Algorithm Systolization", Intern. J. Computer Math.,  $1986(20)$ , 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", Intern. J. Computer Math., 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. Sl8.1-Sl8.17.
- 98. Evans, D.J., Margaritis, K.G., Bekakos, M.P., "A systolic and holographic pyramidical soft-systolic designs for successive matrix powers", Parallel Computing, 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<br>cations, 1988. cell planes to map a R+F algorithm using<br>geometry", to appear in Optics Communi-
- 100. Evans, D.J., Megson, G.M., "Romberg integration using systolic Arrays", Parallel Computing, 1986(3), pp. 289-304.
- 101. Evans, D.J., Shanehchi, J., Rick, C.C., "A Modified Bisection Algorithm for the Determination of the Eigen-<br>values of Symmetric Tridiagonal Matrix", <u>Numer</u>. <u>Math</u>.,<br>1982(38), pp. 417-419. values of Symmetric Tridiagonal Matrix", <u>Numer. Math</u>.,<br>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, Processor Arrays", 34), pp. 734-740. H.T., "Synchronizing Large VLSI IEEE, Trans. on Computers, 1985(C-
- 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 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, by systolic arrays", pp. 19-26. H.T., "Matrix triangularization Proc. SPIE, RTSP IV, 1981(298),
- 114. Gerald, C.F., "Applied Numerical Analysis", Addison-Wesley, 1970.
- 115. Gill, P.E., et al., "Methods for modifying Matrix Factorizations", Math. Comput., 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 designs", Proc. pp.140-149. unifying framework for systolic AWOC '86, Springer-Verlag, 1986,
- **119. Guerra, C., Kanade, T., 11A** systolic algorithm for Algorithms and Architec- (eds.), North-Holland, stereo matching", in "VLSI: tures", Bertolazzi, P., et al. 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., " $AT^2 = O(Nlog^4 N)$ , 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", IEEE Computer, 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<br>Orthogonal Equivalence Transformations and their Appli-Orthogonal Equivalence Transformations and their Appli- cations", Proc. of Conf. on Advanced Research in VLSI, M.I.T. I 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 puting the square root of a matrix", IEEE, Automatic Control, 1978(AC-23), pp. 494-495. of com-Trans. on

----------
- 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</u>. <u>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 eigen-<br>value 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", IEEE, Trans. on Com-Huang, K.H., Abraham, J.A., "Algorithm based fault<br>tolerance for\_matrix\_operations",\_<u>IEEE,\_Trans.\_on\_Com--</u><br>puters,\_1984(C-33),\_pp.\_518-528.
- 135. Hsu, F.H., et al., "LINC: the link and interconnection chip", Technical CMU, 1984. Report, Dept. of Computer Science,
- 136. Hsu, I.N., et al., "The VLSI Implementation of a Reed-Solomon Encoder using Berlekamp's Bit-Serial Multiplier Algorithm", IEEE, Trans. on Computers, 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 for VLSI Arithmetic Systems", IEEE, Trans. ers,  $1982(C-31)$ , pp.  $1215-1224$ . Algorithms on Comput-
- 139. Ibarra, O.H., Kim, S.M., Palis, M.A., "Designing Sys-<br>tolic Algorithms using Sequential Machines", IEEE, Trans. on Computers, 1986(C-35), pp. 531-542. IEEE,
- 140. Inmos Ltd., "OCCAM: user's manual", Prentice-Hall, 1985.
- 141. Inmos Ltd., "Trahsputer: 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, tolic 13-21. I.C.F., "Singular Value Decomposition with Sys-Arrays", Proc. SPIE, RTSP VII, 1984(495), pp.
- 144. Jagadish, H.V., et al., "A study of Pipelining in Computing Arrays", IEEE, Trans. on Computers, 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 Jenkins, M., Traub, J., "A three-stage algorithm for<br>real polynomials using quadratic iteration", <u>SIAM</u> J.<br><u>Numer</u>. <u>Anal</u>., 1970(7), pp. 545-566.
- "First Course in Numerical Methods", 148. Jennings, W.,<br>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,  $198\bar{6}(7\bar{4})$ , pp. 732-741.
- 154. Koren, I., Enhancement through redundancy in VLSI and WSI Multiprocessor Systems", <u>Proc</u>. IEEE, 1986(74), pp. 699-711. Pradhan, D.K., "Yield and Performance
- 155. Kunde, M., "A General Approach to sorting on 3 dimensionally Mesh-connected Array", Proc. CONPAR '86, Springer-Verlag, 1986, pp. 329-337.
- 156. Kunde, M., et al., "The Instruction Systolic Array and kunde, H., et al., The Instruction Byscolic Aria, and Proc. Parallel Computing '85, North-Holland, 1986, pp. 491-498.
- 157. Krishnakumar, A.S., Morf, M., "A tree the symmetric eigenproblem", Proc.  $1983(441)$ , pp.  $77-\overline{83}$ . architecture for SPIE, RTSP VI,
- 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?", IEEE Computer, 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,  $1984(1)$ ,  $\overline{p}p.$  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", Proc. IEEE, 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", IEEE, Trans. 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, Image Processing", Image Processing", 1984, pp. 133-155. J.T., "VLSI Wavefront Arrays for in "VLSI for Pattern Recognition and Fu, K.S. (ed.), Springer-Verlag,
- 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", IEEE, Computer, 1985(18), pp. 42-52.
- 178. Lang, H.W., et al., "Systolic Sorting on a Mesh-<br>connected Network", IEEE, Trans. on Computers, 1985(C-<br>34), pp. 652-658. connected Network", IEEE, Trans. on Computers, 1985(C-
- 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", IEEE, Trans. on Computers, Leighton, F.T., Leiserson, C.E.,<br>tion of systolic arra<mark>ys"</mark>, <u>IEEE</u><br>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",  $66 - 77.$ IEEE, Trans. on Computers, 1985(C-34), pp.
- 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 86-11, School of 1986. techniques", Technical Report, EE-CEG-Electrical Engineering, Cornell Univ.,
- 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", Report, 887, Centre for Theoretical Computer Dept. of Computer Sci., Univ. of Leeds, 1987. Technical Science,
- 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.

lEE, 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<br>Matrix Algorithms", Proc. Parallel Computing '85, North-Holland, 1986, pp. 171-180.
- 202. Melhem, R., "Irregular Wavefronts in Data-driven, Data-dependent Computati6ns", 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. df 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, duct on orthogonally appear in IEEE, Trans. M., "Complexity of matrix proconnected systolic arrays", (to on Computers, 1987).
- 206. Midwinter, J.E., "Light electronics, myth or reality?", Proc. lEE, 1985(132-J), pp. 371-383.
- 207. Moldovan, D.I., "On the Analysis and Synthesis of VLSI algorithms", IEEE, Trans. on Computers, 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", IEEE, Trans. on Computers, 1986(C-35), pp. **1-12.**
- 210. Moler, c., Van Loan, c., pute the exponential  $1978(20)$ , pp.  $801-838$ . "Nineteen dubious ways to comof a matrix", <u>SIAM</u> Review,
- 211. Moraga, c., "On a case arrays", <u>INTEGRATION</u>,  $243 - 253$ . of symbiosis between systolic the <sup>"</sup>VLSI journal, 1984(2), pp.
- 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 Nuclear Systems", John Wiley, 1977. and
- 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., of Unary functions", Proc. Arithmetic, 1985, pp. 82-85. "High-speed computation 7th Symposium on Computer
- 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", SIAM J. Sci. Stat. Comput., 1985(5), pp. 620-632.

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

- **544** -

and other reorderings", IEEE, Trans. on Computers, 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., Analysis for Large Dept. of Mathematics and Computer Science, Clarkson Univ., 1986. "VLSI Structures and Iterative Scale Computation", Ph.D. Thesis,
- 225. Parlett, B.N., "Laguerre's method applied to the Matrix<br>Eigenvalue problem", Maths. Comput. 1964(18), pp. Eigenvalue problem", Maths. Comput. 464-485.
- 226. Parlett, **tions <sup>11</sup> ,**  Kailath, 106-120. B.N., "Remarks on Matrix Eigenvalue Computa- in "VLSI and Modern Signal Processing", T., et al. (eds.), Prentice-Hall, 1985, pp.
- 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 Algo rithms and Processor Arrays", Springer-Verlag, 1986, pp. 165-172. Proc. CONPAR '86,
- 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", Proc. IEEE, 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.), Northand Architectures", Cosnard, M., et al. (eds.), North-<br>Holland, 1986, pp. 259-268.
- 238. Ramakrishnan, I.V., Fussell, D.S., Silbersachatz, A., "Mapping Homogeneous Graphs on Linear Arrays", IEEE, Trans. on Computers,  $1986(\tilde{C}-35)$ , 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- Processing", Proc. IEEE ICASSP, 1985, pp. 26.9.1-<br>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", Proc. IEEE, 1984(72), pp. 820-830.
- 242. Robert, Y., "Block LU Decomposition of a Band Matrix on a Systolic Array", Int. J. Comput. Math., 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)", Computing, 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, systolic Journal, 1986(5), pp. 196-204. P.M., "Experience with OCCAM for simulating and wavefront arrays", Software Engineering
- 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., **array", in**  Cosnard, M., 281-292. "A comparison-based instruction systolic "Parallel Algorithms and Architectures", et al. (eds.), North-Holland, 1986, pp.
- 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 arrays", in "Computing methods in Applied Sciences and Engineering, VI", Glowinski, R., et al. (eds.), North-Holland, 1984, pp. 285-295. eigenvalues of symmetric matrices using systolic ngineering, 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<sup>"</sup>, BIT, 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., updating the pp. 4 51-466. Tang, W.P., "On Systolic Arrays for cholesky Factorization", BIT, 1986(26),
- 264. Seitz, C.L., "Concurrent, VLSI Architectures", IEEE, Trans. on Computers, 1984(C-33), pp. 1247-1265.
- 265. Serbin, S.M., Blalock, S.A., "An Algorithm for Computing the Matrix Cosine", SIAM J. Sci. Stat. Comput., 1980(1), pp. 198-204.
- 266. Shapiro, Parallel Weizmann E., "Systolic Programming: a paradigm of Processing", Technical Report, CS84-16, Institute of Science, Rehovot, Israel, 1984.
- 267. Shepherd, T., McWhirter, J., "A linearly constrained least-squares Int. workshop on Systolic Arrays, Univ. of Oxford, 1986, pp. 151-159. systolic array for optimisation", Proc.
- 268. Singer, B., Spilerman, S., social processes by Markov Sociology, 1976(28), pp. 1-55. "The representation of Models", American J. of
- 269. Snyder, L., "Introduction to the configurable highly  $\longrightarrow$  parallel computer", IEEE, Computer, 1982(15), pp. 47-
- 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", *IEEE*, Trans. on Computers, 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 Systo1ic 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", Applied Optics, 1986(25), pp. 3033-3046.
- 280. Tee, G.J., "An application of p-cyclic Matrices for Tee, G.J., "An application of p-cyclic<br>solving periodic parabolic problems",  $1964(6)$ , pp.  $142-159$ . Numer. Math.,
- 281. Tenorio, systems Parallel M.F.M., Moldovan, D.I., "Mapping production into Multiprocessors", Proc. Int. Conf. on 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 Sys- tolic 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", Proc. IEEE, 1984(131-F), pp. 623-631.
- 288. Van Loan, C., "A Note on the Evaluation of Matrix Polynomials", IEEE, Trans. on Automatic Control, 1979(AC-24), pp.  $32\overline{0-321}$ .
- 289. Varga, R.S., "Matrix Iterative Analysis", Prentice-Hall, 1962.
- 290. Verber, C.M., cal optical 942-953. "Integrated-optical approaches to numeriprocessing", Proc. IEEE, 1984(72), pp.
- 291. Verber, C.M., "Integrated optical architectures for matrix multiplication", Optical Engineering, 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 Springer-Verlag, 1982. in computer Science, 127,
- 294. ward, J.S., McCanny, J.V., McWhirter, J.G., "A Systolic Implementation of the Winograd Fourier Transform Algo-<br>rithm", 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 Processes", N.P.L. Notes on H.M.S.O., London, 1963. Errors in Algebraic Applied Science, No. 32,
- 298. Wilkinson, J.H., "The Algebraic Eigenvalue Problem", Claredon Press, Oxford, 1965.
- 299. Yalamanchili, s., Aggarwal, J.K., "Reconfiguration strategies for parallel archi tectures", IEEE, Computer, 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", INTEGRATION, the VLSI journal, 1984(2), pp. 213-225.

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

 $\sim$ 

 $\mathcal{A}^{\mathcal{A}}$ 

 $\label{eq:2.1} \mathcal{L}(\mathbf{z}^{(i)}_{\mathbf{z}}) = \mathcal{L}(\mathbf{z}^{(i)}_{\mathbf{z}}) = \mathcal{L}(\mathbf{z}^{(i)}_{\mathbf{z}})$ 

 $\mathcal{L}_{\mathcal{A}}$ 

 $\mathcal{A}^{\mathcal{A}}$ 

 $\mathcal{L}_{\mathcal{A}}$ 

 $\mathcal{L}^{\mathcal{L}}$ 

 $\sim$  5  $^{-1}$ 

 $\mathcal{L}_{\text{max}}$  ,  $\mathcal{L}_{\text{max}}$ 

 $\sim$ 

 $\label{eq:2.1} \frac{1}{\sqrt{2}}\sum_{i=1}^n\frac{1}{\sqrt{2}}\sum_{i=1}^n\frac{1}{\sqrt{2}}\sum_{i=1}^n\frac{1}{\sqrt{2}}\sum_{i=1}^n\frac{1}{\sqrt{2}}\sum_{i=1}^n\frac{1}{\sqrt{2}}\sum_{i=1}^n\frac{1}{\sqrt{2}}\sum_{i=1}^n\frac{1}{\sqrt{2}}\sum_{i=1}^n\frac{1}{\sqrt{2}}\sum_{i=1}^n\frac{1}{\sqrt{2}}\sum_{i=1}^n\frac{1}{\sqrt{2}}\sum_{i=1}^n\frac$ 

 $\sim$ 

 $\mathcal{L}_{\rm{max}}$ 

# A P P E N D I X

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 c!e 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.

Constructs: 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  $\overline{P}_1$ <br> $P_2$ <br> $P_3$ 

where the component processes  $p_1$ ,  $p_2$ ,  $p_3$ , . . . 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  $P_1 P_2 P_3$ 

and in contrast to seq, here all the component processes  $P_1$ ,  $P_2$ ,  $P_3$ , . . . 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  $p_1$  condition 2  $P_2$ .

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

```
alt 
      input 1 
      \begin{array}{c} p_1 \\ p_2 \\ p_3 \end{array}P_2
```
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.

Repetition: 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.

Replication: A replicator is used with a constructor to replicate the component process a number of times. With 'seq' a standard for loop

seq i=[O 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=[O for n] 
  P_i
```
is created such that n similar processes  $p_0, p_1, \ldots, p_{n-1}$  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 to a following process by a colon(:) at the last line of the declaration. The process follows on the next line at the same level of indentation as the keyword declaration. For 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(O], 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, c_1, c_2 :<br>par
    \begin{array}{c} \texttt{par} \quad 1 & 2 \\ \texttt{buffer}(c_1, c) \\ \texttt{buffer}(c_2, c) \end{array}buffer(c_1^1c_2)
```
declares two buffer processes executed concurrently, buffer is the named process with formal channel parameters in and out. In the following process  $c_1, c_2$ , are actual parameters and on execution the 'while' loop will replace the procedure calls and  $c, c_1, c_2$  will replace in and out channels. The size of a vector is not specified in the formal parameters of a 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

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.

Program 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:

Constructs: 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.

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

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

Declarations: 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.

Comments: 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 implements 'proto-OCCAM'. A more sophisticated version OCCAM-2 is now available providing Real, Integer, and 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.

# II. LOUGHBOROUGH IMPLEMENTATION OF OCCAM

Help for running the occan compiler

A source 'occam' file (OCCAM 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 ".ooc' 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 occan compiler and built in linker.

Bach previously compiled occan program is specified in the command line in the form  $" \cdot \circ" \cdot \circ \cdot$ 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 -1 option is used to generate new versions of library file objects.

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

#### $\pi$ :

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.

#### $\sim$  :

occam keep it -o saverun

Compile 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 occan specification. LUT extensions (see file 'occanwersion') currently include :-

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

 $-c$ :

 $1.001 - c$ 

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

-G : occam -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.

 $\sim$  .

 $a.out -a$ 

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

 $-F$  and  $-M$  :

 $occam - P$  num-occ.

'-P' Includes the floating point library routines to provide a simple real mathematical library routines to provide mathematical library routines.

 $-I$ :

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

Pull 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 magnonic 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

- $-5$ The next flag(s) are system flags  $\sim$  switch flag mode.
- Run the program with Cursor addressable options enabled.  $\sim$ The library routines 'clear-screen' and 'goto-x.y' need this flag set. If used for the compiler must also give the  $-x$  option.
- Produce object/run object for Execution tracing. The resulting object  $\bullet$ file is then run with the '-e' option. This utility is described in 'tracerinfo'.
- Porce full occam semantic cbeck on use of variables.  $-1$ 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.
- Run the resulting object file if compilation succeeded.  $-\mathbf{g}$ The program Goes immediately it is ready to.
- Print out this 'Help' information.  $\rightarrow$
- $-1$ Force an Interrupt impediately before start of execution immediately displays the debug help menu. This enables break and trace points to be setup prior to anything being executed.
- Compile but do not link the occam source. Needed when  $-1$ using multiple occam source Library files.
- Check that every channel Match properly on execution, channels can  $-12$ have only one input and one output process during execution.
- Produce an Object program with name given by the non-switch  $\sim$ argument following this switch. Enables you to choose an object file name other than 'a.out'.
- Run the program without outputting some non occam program produced ۳Q 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.
- 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 compliations  $-$  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 Floating 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.

 $\mathbf{u}$ 

 $\sigma$ 

N

- -M Include the Mathematical library and floating point routines.
- -O Produce optimized object. May improve run time by 20%.
- -R Use Randonized 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 alower 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.

## **Systen Flags**

- $-1$ Switch back to expecting 'user' mode flag options. This means you can replace -G%v by -%v%G.
- Enable Analysis of the usage of channels this facility is still  $-\mathbf{a}$ under test.
- Check the source occam for syntax errors, but do Not produce any ÷n. object data from it.
- Print out the program in the form just after it was Transformed.  $-<sup>2</sup>$ Not generallt useful as the program has changed so much.
- Give Verbose information at each stage when running the compiler - $-9<sup>1</sup>$ 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 occan link information.

- $-A$ Produce the object code ('C' or Assembler) in a permanent file so that it can be inspected.
- Produce 'C' rather than assembler output from the occam compiler  $\mathbf{C}$ 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 I (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 occan-'C' interface Library, the two files ending '-c.c' and  $-1$ '.occ' are linked together, the occan can refer directly to the 'C' routines.
- Run the compiler showing the steps it would execute but without -41 actually doing anything - like '-n' in the UNIX 'make' command. Useful when options start getting complicated. A No operation facility.
- -9 Undocumented feature under test.
- $\sim$ Do not apply some Simplifying transformations on the program. These currently remove constructs with no processes in then and redundant SEQ and PAR headers. These save a small amount of space and time at run and compile time and there is little noint in turning off this option.
- Print out the procedures that have been defined in the link files  $-\mathbf{X}$ but has not been referenced - detects eXtra procedures defined across files but not used.  $-Y$
- Produce the linker assembler output in a permanent file rather than in a temporary file on '/tmp'. Enables the output from the linker to be debugged.
- Get the linker to print out all the definitions it is told about. -7.

# 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 :-

 $\mathbf{r}$ 

<u>un</u>

 $\alpha$ 

**W** 

 $\mathbf{L}$ 

PROC str.to.screen (VALUE s []) =  $550$   $n = [1$  for a  $[BTTS$  0]] screen i 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 ("Xd",n):' statement.

EXTERNAL FROC str.to.chan (CHAN o.VALUE s []) :

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 FROC 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 FROC 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' nackage 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 FROC 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 messare.

## 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'.

## **EXTERNAL 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 'CIRL-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 FROC close.pipe (CHAN io.chan) :

Cease connection of the channel with its currently active command.

EXTERNAL IROC 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 microseconds. Timers count downwards and are not especially accurate.

 $\mathbf{r}$ 

#### Floating Point Library

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

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

Input/Output Routines

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

Print out the floating point number in 'C' language float format "%6.6f". If the number is too small or too big the standard 'C' action will be taken.

EXTERNAL FROC fo.num.to.screen.f (VALUE FLOAT f.VALUE w.d) :

Print out the floating point number in 'C' real format "Xw.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 'ecreen' this is equivalent to 'fn.num.to.acreen', Useful for writing data to files.

EXTERNAL PROC fo.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.

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

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



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 scmeone who knows their way around the canpiler), an option to specify which oource tile 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 fonrer, but the latter rmy be Ignored 1f the symptau ls expected.

The run time display debugger

The utility requires the use of a cursor addressable tenninal. 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 **OD** 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 autanatically entered when any of the process executes any of the source lines on wich a break point ls set. Trace points cause temporary entry into the debug display before resuming normal execution after five seconds pause.<br>If a file has been compiled with the '-G' flag then the value of occan

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 lt.

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.** *'t'* **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.

**'lbere are two special markers that are used,** '>' **oa 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

11>- llave forward half a page of source text.

- **t P- Mc>ve forward a s:age** *ot* **source text.**
- 1U- Move backward half a page of source text.
- t 8- Move backward a page of oource text.
- : <number> Move to given line <a></a>umber> in file.
- $k (or \t fK)$  Move down one line.
- 
- $j (or fJ)$  Move up one line.<br>/<string> Find given <string> in file from current position.
- n Find next string occurrence for match string selected by  $'$  /' command.

<.n  $\sigma$  $\bar{c}$ 

 $\mathbf{r}$ 

 $\mathbf{r}$ 

p - Find the next process ln the file.

## Trace/Breakpoints

- **b** Add breakpoint at currently selected line.
- t Add tracepolnt 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 or 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 arc 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 ls desired ls 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, 1t no symbols have been selected before then the symbol window is opened and the value of the variable or the status of a channel.
- 14 Repeat the previous 'm' conmand. To find the value of the same variable **name again.**

## **Execution control**

- 
- $a -$  Abort the run.<br> $r -$  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.

## **Klecellanoows**

- 
- ? Print out this help information.<br> $FL-$  (or  $TR$ ) Redraw the current displayed information.
- 1 Buffer keyboard channel input text for the program.
- $0$  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' conmand 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 sane cireunstances. Most sequential oode 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 - my be chosen tor execution at a.ny time.**
- $a$  Process waiting for one or more ALT guards to became TRUE.<br> $\pi$  Process waiting for a clock time or for input/output.
- 
- c Process is waiting for one or more child PAR processes to tenninate.

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<sup>1</sup> carriage return and '\*\*' space.

> $\mathbf{u}$  $\mathbf{c}$

> > $\mathbf{1}$

 $\mathbf{I}$ 

#### Non standard occam features.

This compiler to the best of my knowledge (Mr.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 denoribed in this file. These differences are intended to make transfer of occan programs from different implementations feasible.

It is intended to be compatible to the INMOS booklet version and the Prentice liall book definition. OCCAM, INMOS and Transputer are registered trademarks of the INMOS Group of Companies.

#### INIOS proto-occam language revisions

The following additional features introduced into INMOS occam products can now be selected by the compiler flag option '-I'.

STOP primitive.

TIME channel.

IF on finding none 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. Momber 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 permitted 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 enother.

The definition is preceded by the new keyword 'LIBRARY' before 'PROC' and the definition must be at the outer level of program nesting.

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.





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.o to link in the pre-compiled sub.occ file

in J.O. Onte mas 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 notwork, comme [] : EXTERNAL VAR blot [BYTE].spot [].bolic [FLOAT] : EXTERNAL VAR FLOAT hyper.active [] :

## Floating 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 PLOAT  $x, y$ , factor : - Ploating point number declaration VAR num, ply: - Normal occass variables.

 $\mathbf{F}$ 

Floating point number constants are supported these may be in two forms with decimal point or with decimal point and exponent :-

 $x := 1.45$ 

 $y := 2.3e-23 + 3.4e+1$  - 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)

```
+- + 1 \Leftrightarrow \leftarrow > \leftarrow > \leftarrow \leftarrow (monadic minus)
x := 1.3 + (y + factor)IPx > 67.8Y := -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.

```
FROC sum (VALUE FLOAT a [], b [], VAR FLOAT res [], VALUE n) =
  PAR i = [0 FOR nl
    res [1] := a [1] + b [1]:
VAR FLOAT t [23] a [45] v [32] :
sum (t.g.w.12)
```
Floating 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 :=  $x$   $\rightarrow$  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.

# **Ill.** 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 of OCCAM are used, allowed by the Loughborough implementation. 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 Il 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:

*i/o* 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 streams, or single quantities. sending/accepting control

sources/sinks: 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.

delays, links: they simulate simple bulding blocks required for the interconnection of cells or arrays, when pipelines are formed.

cells: 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 a (2x2) 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 :  $-6$ 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  $r = r(1)$ , r(2),  $\ldots$ , r(n) are the roots of the polynomial, which  $r = \text{area}$  are all real and  $\lfloor r(1) \rfloor$   $>$   $\lfloor r(2) \rfloor$   $>$   $\ldots$   $>$   $\lfloor r(n) \rfloor$ . Then -- S[k] = - a[l]\*S[k-l] - a[2]\*S[k-2] - .. - a[k-l]\*S[l] - a[k]\*k<br>-- for k = 1, 2, .., n, and<br>-- S[n+i] = - a[l]\*S[n+i-l] - a[2]\*S[n+i-2] - .. - a[n]\*S[i] S[n+j) •- a(l]\*S[n+j-1)- a[2]\*S[n+j-2]- •• - a(n)\*S[j) forj•l,2, ••• Initially the quantities a(ll•l, a(21\*2, •• , a(nl•n enter the -- as a flag controlling the operation of the Ring. The Multiplexer -- restores the normal ring operation and the Divider calculates<br>-- the sequence : the sequence *:* 5(21/S(l), S(31/S(2), •• , S(i)/S(i-1], for i•l, 2, •• , that -- converges to r[l]. external proc fp.float(value i, var float f):<br>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<br>var float a(no) :<br>-- Degree of the polynomial; size of the ring; input-output time.<br>var n, size, intime, outime :<br>-- Channels chan a.c((2•so)+2), x.c(so+2), s.c(so), ~.c(so+2) *:*  -- Inner Product Step Cell (Subtraction): accumulates the inner product coefficients in negative form.It inputs a, x; if f•l, l.e for a-a(O)•O it outputs s, saves x as news, and propagates a, 0; -- otherwise it outputs 0 and propagates a, x. proc ips (chan ain, xin, aout, xout, sout)=<br>var float a(2], x(2}, s(2} :<br>var f(2) :<br>seg seq -- initialisation par i=[0 for 2]<br>par  $a[i] := 0.0$  $f[i]$  := 0  $x[i] := 0.0$ <br> $s[i] := 0.0$ -- main operation while true seq -- i/O par ain 7 a[0]; f[0]<br>xin 7 x[0] aout  $1 a(1); f(1)$ xout  $1 \times [1]$ sout 1 s(1); f[1]<br>-- calculation par if

 $-$ 

 $f(0) = 1$ par seq  $s[1]$  :  $s[0]$  $s[0] := x[0]$  $x[1] := 0.0$ true par sill :• 0.0  $x(1) := x(0) - (a(0) * a(0))$ <br>a[1]  $x = a(0)$  $f(1) := f(0)$ : Delay Cell *:* propagates its input with one cycle delay. proc delay (chan ain, aout)-<br>var float a(2) :<br>var f(2) :<br>seq -- initialisation par i•[O for 2) par  $a[i] := 0.0$ <br>f[i] := 0 -- main operation<br>while true seq  $- - 1/6$ par ain? a[OJ; f[O} aout I a[l); f[l) -- 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 reel (chan celin, regin, regout)=<br>var float cel, reg[2] :<br>var f :<br>seg -- initialisation par eel :• 0.0 par i•(O for 21 reg[il *:•* 0.0 f :• <sup>0</sup> while true seq par If  $celli$   $2$   $celli$   $f$ regin ? reg(O) regout 1 reg{l)  $f = 1$  $req(1) := cel$ true reg(l) :• reg[O] *:*  -- 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".

proc divide (chan rin, rout,

 $\epsilon$ ū  $\blacktriangle$  $\mathbf{r}$ 

 $\blacksquare$ 

```
value outime, size)=<br>var float r[2], d :<br>seq
    - initialisation
    par 
      par 1•(0 for 2) 
      ril := 0.0<br>d := 0.0
    -- initial delay
    seq i=[0 for outime]
      par 
         rin ? any 
         rout I d 
    while true<br>seq
         \frac{1}{2} accept results
         seq i=[0 \text{ for size}]<br>seq\frac{r(1)}{par} := r(0)rin 7 r(0)rout I d 
              if 
                 r[1] = 0.0d := r[0] / 0.000001true 
                   d:- r(O) /rill 
          wait for results 
          seq i•[O for size) 
            par 
               rin ? any 
               rout Id:
\sim –
-- Source of the ring : pumps in the coefficients of the polynomial<br>-- as a(O)=O, a(1), .., a(n) and the initial values for s(l), .., 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 a( I, 
                value n)• 
  var i :
  seq 
    i J• 0 
    while true 
       seq 
         par 
            If 
              I - 0 
                  par<br>- aout 1 0.0; 1
                    xout i 0.0i \leftarrow npar<br>aout ! a[i-l]; 0
                    var float temp :
                    seq 
                      fp.float(i, temp) 
                       temp := (-a(i-1) + temp))xout I temp 
               true 
                 par<br>aout 10.0;0
                    xout 10.0rout I 0.0 
          i \cdot i + 1 :
```

```
-- Sink of the ring : it accepts the output from the divider.
 proc sink (chan din, 
   value outime, size).<br>var float d :
   seq 
      seq 1•10 for (outime+l)] 
        din ? any 
      while true 
        seq 
           seq 1•(0 for size] 
              seq 
                din ? d 
                fp.put(d, • root •) 
           seq 1-[0 for size]
              din 1 any : 
-- Multiplexer of the ring : for the "input time" cycles it accepts<br>-- input from the source; then the ring is closed.No delay is caused.
proc mux (chan asin, arin, xsin, xrin, aout, xout, 
   var float a, atemp, x, xtemp ;<br>var f, ftemp, i ;
   seq 
     I :• 0 
      while true 
        seq 
           If 
             1 < intime 
                  asin ? a: f
                   asin ? a; f<br>arin ? atemp; ftemp
                   xsin ? 
X 
                   xrin ? 
xtemp 
             true 
                par<br>- asin ? atemp; ftemp
                   arin 7 a; f
                   xsin ? 
xtemp xrin ? X
           par 
             aout l a; f<br>xout l x
          x := 1 + 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 .. seq 
     if 
        (n \ 2) = 0seq 
             intime r = n + 2true 
          intime := n + 1size := (intime / 2)
     outime := (intime +2) + size
     par 
        source(a.c(OJ, x.c(OJ, r.c(O), a, n)
```
 $\blacksquare$  $n<sub>1</sub>$ .... ັບກ

```
\max(a.c[0], a.c[(2*size)+1], x.c[0], x.c[size+1], a.c[1], x.c[1],--
       par i•(O 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])
            rel(s.c[i], r.c[i], r.c[i+1])\frac{1}{2}divide(r.c(size), r.c(size•l), outime, size) 
       sink(r.c[size+l), outime, size) : 
                                                                                               \simproc getdata • seq 
    qet(n, • de9ree of polynomial ") 
    fp.get.n(a, n, • of coefficients ") 
  getdata 
  system 
                                                                                                   par
```
&eq

```
-- 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[O]*x**n + a[l]*x**(n-1) + .. + a[n]
-- it produces the polynomial
       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 :<br>-- Vectors of coefficients for polynomials f(x), g(x).
var float a[no+1], b[no+1] :<br>
-- Actual degree of polynomials and actual number of ips cells.<br>
var n, s,<br>
-- Overall operation time, and initial delay.<br>
-- Channels for fast-moving a's, slow-moving a's and b's.
chan af.c(so+2), as.c((3*s0)+1), b.c((2*s0)+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,
   var float a[4], b :<br>var neg :<br>seq
                 value time)=
  -- initialisation 
       par i•(O for 4) 
          a[i] = 0.0b := 0.0<br>neg := false
     -- main operation
     seq i=(0 for time)<br>seq
          - i/0par 
             ain ? atOJ 
             afout I a{l) 
             asout ! a[2]<br>bout ! b
          -- calculation
          if<br>neg
                "- negate a-slow on alternate cycles.
               par 
                  a(3) - a(0)neg 1• false 
             true 
          par 
               par 
                  a(3) = a(0)neg :• true 
             a(1) := a(0)a(2) := (2.0 * a(3))b := (a[0] + a[3]):
```
T. ហ J. Ō.

-- 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,<br>value time)=<br>var float a[4], b[2] ;<br>seg -- initialisation par par i•[O for 4)  $a[i] := 0.0$ par i•(O for 2)  $b(1) := 0.0$ -- main operation seq i•(O for time] seq  $\frac{1}{\text{par}}$  1/0 afin ? a[O] asin  $? a[1]$ bin ? b[OJ afout 1 a(2] asout 1 a(3] bout 1  $b[i]$ <br>-- calculation<br>par  $a[2] := a[0]$ <br> $a[3] := a[1]$ b[1]  $:= b[0] + (a[0] + a[1])$ ; -- Delay Cell : propagates its input with one cycle delay. proe delay (ehan xin, xout, var float  $x[2]$  :<br>seq -- initialisation par i•[O for 2) x[l) :• 0.0 -- main operation seq i=10 for time)<br>seq -- i/o<br>par xin ? x[0]<br>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  $a()$ ,<br>value n,time  $) =$ seq i•{O for time)  $\frac{1}{2}$   $\leftarrow$  n aout  $i$  a[i] true aout 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 )• seq 1•{0 for time)

 $\sim$   $-$ 

par afin ? any asin ? any if<br>I < delav bin ? 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  $n+2*(n/2)$ ,  $\overline{\phantom{a}}$ proc system =<br>seq If  $(n \; \backslash \; 2) = 0$  $s := (n / 2)$ true  $s := ((n-1)/2)$ del :=  $(2 * s) + 1$  $time := (n + de1) + 1$ par souree(af.c[O), a, n, time) square(af.c(O), af.c(l), as.c(OJ, b.e(OJ, time)  $par 1- [0 for s]$ par ips(af.c[i+l], as.c[(i\*3)+2], b.c[(i\*2)+1],<br>af.c[i+2], as.c[(i\*3)+3], b.c[(i\*2)+2], time) par j•(O for 21 delay(as.c((i\*3)+j), as.c((i\*3)+(j+1)], time) delay(b.c[i\*2], b.c((i\*2)+1}, time)  $sink(af.c[s+1], asc.c[3*s], b.c[2*s], b, del, time)$ :  $$ proc getdata =<br>seq get(n, " degree of polynomial ")  $fp.get.n(a, (n+1), ' of coefficients '')$ :  $\sim$   $\sim$ proc putdata =<br>seq  $fp.$ put.n(b,  $(n+1)$ , " of coefficients ");  $$ seq getdata

**ហា** .... ....

 $\mathbf{r}$ 

system putdata  $\sim$  $-$  A.1.3  $\overline{\phantom{a}}$ -- Systolic ring for the calculation of the Eigenvalues of a symmetric Tridiagonal matrix with diagonal coefficients  $a(1), a(2), \ldots, a(n)$  $---$  and off-diagonal coefficients<br> $---$  b(1), b(2), .., b(n-1) -- b(1), b(2), .., b(n-1)<br>-- The n eigenvalues are calculated in parallel using the<br>-- properties of the Sturm sequence of polynomials and the<br>-- 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 pcoc fp.get.n(vac float v(), value n,s()): -- Max size of matrix.<br>def no - 10 : -- Vectors of matrix diagonal and off-diagonal coefficients.<br>-- Vector b starts with a zero.<br>-- Bisection interval of the form : point •  $(a+b)/2$ ,<br>var float a(no), b(no), x, d :<br>-- Actual size of matrix. var n :<br>-- Channels for eigenvalues, bisection distance, sturm polunomial<br>-- value, number of sign changes; i/o channels. chan x.c(no+4), d,c(no+4), q.c((3•no)+l), 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.  $\sim$ proc check(chan xin,din,sin,xout,dout)=<br>var float x[2], d{2] :<br>var sign, tag[2], count : seq -- initialisation par par 1•10 for 2) par  $x[i] := 0.0$ tag[!) *:•* 2 d[i] := 0.0<br>sign := 0<br>count := 0 -- main operation while true seq -- i/o<br>par par xin ? x(O) *:* tag(O) din ? d(O) sin ? sign  $x$ out  $1 \times [1]$ ; tag $(1)$ <br>dout  $1 \times [1]$ -- calculation if<br>sign > count  $-$  more than required eigenvalues in the interval  $x(1) := x(0) - d(0)$ 

true  $x(1) := x(0) + d(0)$ par<br>tag(1) := tag(0)<br>d(1) := d(0) / 2.0 if tag[O]  $\leftrightarrow$  0 -- last eigenvalue or reset  $count := 0$ true count *:•* count + 1: -- Sturm Sequence Polynomial calculation cell: for given x, q(i-1)  $-$  it clculates q(i) = (a(i)-x)-(b(i)\*\*2/q(1-l)).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, qin, xout, qout,<br>value float a.b)= var float  $x[2]$ , q $[2]$ :<br>var tag $[2]$  :<br>seq -- initialisation  $par_i=[0 for 2]$  $x[i] := 0.0$ tag $[1]$   $-2$ q[i] :•1.0 -- main operation<br>while true seq -- 1/o par xin  $2 \times \{0\}$ ; tag $\{0\}$ qin ? q(OI  $x$ out  $1 \times [1]$ ; tag[1]<br>gout  $1 \times [1]$ -- calculation par  $x[1] := x[0]$  $tag[1] := tag[0]$ <br>seq  $\frac{f}{\ln 101} = 0.0$  $-$  q(i-1) is replaced by e q(0)  $:$  0.00001 q(1) := (a - x(0)) - (b / q(0)) : -- Sturm Sequence Sign calculation cell : for given q[i], it  $-$  checks if q[i] is negative.If yes, the number of sign changes -- is incremented by one. proc sign(chan qin, sin, sout)=<br>var float q :<br>var s[2] :<br>seq --initialisation par  $q := 0.0$ par  $i = \{0 \text{ for } 2\}$ <br>
s $\{i\} := 0$ <br>
-- main operation<br>
while true seq -- i/o par

 $<sub>u</sub>$ </sub>  $\overline{\phantom{0}}$  $\infty$  $\mathbf{1}$ 

 $\blacksquare$ 

```
qin ? q<br>sin ? s[0]&out I a[l) 
              -- calculation 
              If 
                q < 0.0 
                    s[1] := (s[0] + 1)true 
                    s(1) := s(0):
-- Delay Cell for d: propagates d with one cycle delay.
--
proc delay (chan xin, xout)=<br>var float x(2) ;<br>seq
       \frac{1}{n} - initialisation<br>par i=[0 for 2]
       x[i] := 0.0<br>-- main operation<br>while true<br>seq
              - i/o<br>par
                 xin 7 x[0]<br>xout 1 x[1]-- calculation
              x(1) := x[0]:
-- Delay Cell for x: propagates x and tag.
proc delay.x (chan xin, xout)=<br>var float x(2) :<br>var tag(2):<br>seq
        -- initialisation
       par 1•(0 for 2) 
          par<br>x[i] := 0.0\begin{array}{rcl} \texttt{tag[1]} & := & 2 \\ \texttt{--} & \texttt{main operation} \\ \texttt{while true} & \end{array}seq 
              \frac{1}{\pi} i/o<br>par
                 x \ln 7 x(0); tag[0]
              xout I x[1] ; tag[1]<br>-- calculation<br>par
                 x(1) := x(0)tag[0]:\simBranching : produces two channels from one without delay. 
 proc branch(chan xin, xlout, x2out)=<br>
var float x :<br>
seq<br>
-- initialisation<br>
x := 0.0<br>
-- main operation<br>
while true<br>
seq
               - in
              xin 7 x 
               --out 
              xlout I x 
               x2out I 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)=<br>var float x, xtemp, d :<br>var tag, tagtemp, start *i*<br>seg  $\frac{1}{x}$  - initialisation xtemp :• 0.0  $d := 0.0$ tag  $:= 2$ tagtemp *:•* <sup>1</sup> -- main operation<br>while true seq  $\frac{1}{1}$  input not stact par  $x \in \mathbb{R}$   $x : tag$ <br>dlin ? d  $x2in$  7  $xtemp$ ; tagtemp d2in 7 any par xlin 7 xtemp; tagtemp<br>dlin 7 any<br>x2in 7 x ; tag<br>d2in 7 d -- output par if ·not start xlout I xtemp ; tagtemp true xlout  $x$  ; tag  $x2$ out  $1 x ; tag$ <br>d2out  $1 d$ <br>-- control if tag  $-1$ stact :• true *:*  -- Source of the ring : pumps in pairs of the original bisection<br>-- interval in the form  $\rightarrow$  x = bisection point, d = bisection distance. The nth x has  $-$  tag bit equal to 1.  $\overline{\phantom{a}}$ proc source (chan xout, dout, value float x, d, value n)• var i : seq  $\frac{1}{1}$  -- initialisation<br>i := 0 -- main operation<br>while true seq par If  $i - (n - 1)$ xout  $x : 1$ true

 $\blacksquare$  $\mathbf{u}$ <u>.</u><br>ن  $^{\bullet}$ 

```
xout 1 \times 10dout l d 
          1 := (1 + 1):
- Produce q(0) - 1 and s(0) - 0 all the time.
proc feed(chan qout, sout)• 
  while true 
     par 
       qout 1 1.0 
        sout 1 \t0:
--
-- 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.
\sim \simproc sink (chan xin)=<br>var float x:
  var tag :<br>seq<br>-- initialisation<br>par
        x := 0.0<br>tag := 2
      -- main operation<br>while true
        seq 
           xin 7 x : taqIf \begin{array}{c} 1 \text{f} \\ \text{tag} \sim 2 \end{array}\frac{2}{\pi}- no dummy value<br>sec
                   f_p.num.to.screen(x)str.to.screen(* *)
            if<br>tag = 1
                 \frac{1}{2} -- last eigenvalue<br>str.to.screen(* *c *n *) :
 Receive q(n) all the time. 
 proc qn(chan qin)• while true 
      qin 1 any : 
 --<br>
-- Ring configuration<br>
--
 proc system •<br>par<br>-- array for Sturm Sequence computation
       par i=[0 for n]<br>par
            delay(d.c(i),d.c(i+1))sturm(x.c(i),q,c(3•i),x.c(i+l),q.c({3*i)+l),a(i),b{i)) 
            branch(q.c( { 3•1 )+1) ,q.c( ( 3*1 )+2J,q.c( ( 3*1 )+31) 
            sign(q.c((3+i)+2),s.c(1),s.c(1+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),\n  out.c,x.c(0),d.c(0))-- i/o of the ring<br>source(in.c[O],in.c[1],x,d,n)
       sink(out.c) 
       feed(q.c[0], s.c[0])
```

```
qn(q.c[3*n]):
\simproc getdata =<br>seq<br>qet(n, " size of matrix ")
        fp.get.n(a, n, * of diag coeffs *)<br>fp.get.n(b, n, * squares of off-diag coeffs : first coeff = 0 *)<br>fp.get(x, * bisection point *)<br>fp.get(d, * bisection distance *)
       str.to.screen(* *c *n *) :
\simseq
```

```
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 \frac{1}{2} ((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[] ) :<br>external proc fp.get ( var float v, value s[] ) :
external proc fp.qet.n ( var float v( ), value n, s( ) ) :<br>external proc fp.put.n ( value float v[ ), value n, s( ) ) ;
external proc fp.so.x (chan xout, value float x(), value time) :<br>external proc fp.so.d (chan xout, value time) :<br>external proc fp.si.x (chan xin, var float x(], value time) :<br>external proc fp.si.d (chan xin, value tim
def no -10 :
-- Basic cell : Inner Product with modified output<br>-- 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);<br>seq
      -- initialise
      pat 
         a := 0.0par i•(O for 2) 
               b(i) := 0.0x(i) :• 0.0 
      main operation 
      seq i•IO for time) 
         seq 
            -- 1/0<br>par
               ain 7 a
               bin 7 b[OJ 
               x \in \begin{bmatrix} 0 \\ 1 \end{bmatrix}<br>aout 1 b[1]<br>bout 1 b[1]
            xout I x[1]<br>-- calculation
            par 
               b[1] := a + (b[0] * x[0])
               x[1] := x[0]:
 -- Pipeline configuration<br>-- Row i of the array has m = (d+l)-i hip cells, each one
 acceptin9 a coefficient in skewed fashion; b, x are 
 pipelined.The partial results (b), are also produced as 
 the coefficints for row i+l of the array. 
proc pipe ( chan co.io( ), x.in, b.out,<br>value m, base, time ) =
   chan b.c[no], x.c[no] :<br>par
      par i=[0 \text{ for } m]<br>hip (co.io[base+i], b.c[i], x.c[i],
       co.io[(base+m)+(i+l)], b.c[i+l], x.c[i+l], time )<br>fp.so.d ( b.c[0], time )<br>fp.lk ( x.in, x.c[0], time )
```

```
fp.lk ( b.c(m], b.out, time 
     fp.si.d ( x.ciml, time ) 1
-- System configuration
- 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 ) \bullet<br>chan co.c(no*no) :
  var base[no] :<br>seq<br>base[0] := 0<br>seq i=[0 for n]
        base[i+1] := base[i] + ((n + 1) - i)par 
        par i=[0 \text{ for } n]<br>par
         pipe ( co.c, x.in{i}, b.out[i], (n-i), base[i], time ) fp.si.d ( co.c[base[i+l]-1], time ) par i=[0 \text{ for } n]fp.lk ( a.in[i], co.c[base[0]+i], time )<br>fp.so.d ( co.c[base[1]-1], time )<br>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] :<br>seq<br>temp[0] := ax<br>fp.so.d ( axout, t)
      fp.so.x ( axout, temp, 1)<br>fp.so.d ( axout, (time-(t+l))) :
-- Delayed sink for b.
proc si,b ( chan bin, 
                  var float b,<br>value time ) -
   var float temp[1] :<br>seq<br>fp.si.d ( bin, (time-1) )<br>fp.si.x ( bin, temp, 1 )
      b := temp(0) :
-- System interface-driver
proc driv ( value float a(), x, 
                  value n,<br>var float b[] ) =
   chan a.c[no], x.c[no], b.c[no] :<br>var time :
   seq<br>time := n + 1
      par 
         par i•(O for n) 
            par 
               so.ax \{a.c[i], a[i], i, time\}so.ax (x.c[i], x, i, time)<br>si,b ( b.c[i], b[i], time )
         syst \{a,c, x,c, b,c, n, time \}:
```
 $\mathbf{I}$  $\mathbf{u}$  $\bullet$ .....  $\blacksquare$ 

```
-- Main
var float a[no), x, b(nol 1 
var d : 
   The ( d, * degree of polynomial * )<br>fp.get.n ( a, (d+l), * polynomial coeffs * )<br>fp.get ( x, * evaluation point * )<br>driv ( a, x, (d+l), b )
    fp.put.n ( b, (d+1), '' results " )
```

```
\frac{1}{2}- A.1.5
-- Systolic Pipelines for the Bairstow Method.
- For a polynomial f(x) of degree d, and a quadratic
      For a polynomial f(x) of degree d, and a quadratic<br>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).
f(x) = Area = 2*d IPS cells, Time = 2*d+6 IPS steps.
external proc get ( var v, value s[] ) :<br>external proc fp.get ( var float v, value s[] ) :<br>external proc fp.get.n ( var float v[], value n, s[] ) :<br>external proc fp.put.n ( value float v[], value n, s[] ) :<br>external proc f
external proc fp.si.x ( chan xin, var float x[), value time ) : external proc fp.si.d ( chan xin, value time ) :<br>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+1 : b = b+p^b[-1];<br>-- 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, 
    var float a, b[3], p[3];<br>seq
        - initialise
        par 
           a :• 0.0 
            par i=[0 for 3]<br>par
                  b(i) := 0.0<br>p(i) := 0.0-- main operation
         seq i•(O for time) 
                \frac{1}{\text{par}} /o
                   ain 7 a
                   bin ? b[0]<br>pin ? p[0]\begin{array}{c} \text{a} \text{out} & \text{b}(2) \\ \text{b} \text{out} & 1 \text{ b}(2) \\ \text{p} \text{out} & 1 \text{ p}(2) \\ \text{-} \text{c} \text{alculation} \end{array}\frac{p_0 r}{if}(i \setminus 2) = 0<br>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<br>-- The coefficients enter the pipeline in skewed fashion;
```
01  $\infty$ N

 $\mathbf{r}$ 

 $\mathbf{L}$ 

-- p,q and b travel along the pipeline.

 $\mathbf{g}^{\prime}$ 

```
proc pipe { chan co.io(), p.in, b.out, 
               value m, base, time 1 -chan b.c(no), p,c(no) :<br>par<br>par i={0 for m}
       bip i co.io(base+il, b.c(i), p.c(i),
               co.io((base+m)+(i+1)), b.c(i+1), p.c(i+1), time)
     fp.so.d ( b.c(O], time I 
     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 aq, p,<br>value t, time ) =
   var float temp[2]:<br>seq
     temp(0) := aq<br>temp(1) := pfp, so.d ( a point, t)fp.so.x ( apout, temp, 2) 
     fp.so.d ( apout, (tine-(t+2))) :
-- Delayed sink : collects for the last two cycles.
proc si.b ( chan bin,<br>var float b[],<br>value i, time ) =
   var float temp[2] :<br>seq<br>fp.si.d ( bin, (time-2) )
      f_{\text{D}}.si.x (\text{bin}, temp, 2)
      seq j=[0 \text{ for } 2]b(\{1:2\}+j) := temp(j) :
-- System configuration<br>-- 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.
 لمنابط
proc syst ( chan a.in[], p.in[], b.out[],<br>value n, time ) =
   chan co.c[3*no] :<br>par<br>pipe ( co.c, p.in[0], b.out[0], n, 0, time )
      pipe ( co.c, p.in[l), b.out(l), (n-1), (n+ll, time ) 
      fp.so.d ( co.c(n), time ) 
      fp.si.d ( co.c(n), time ) 
      par i•(O for n) 
           fp.lk ( a.in(i], co.c(i), time )fp.si.d ( co.c[(2*n)+1], time ) :
 -- System interface-driver
 proc driv ( value float a[], p, q,<br>value n,<br>var float b[] ) =<br>chan a.c(no), p.c(2), b.c(2) :<br>var time 1<br>seq
      time i = (2 + n) + 2
```

```
par 
       par i=[0 \text{ for } n]so.ap ( a.c(i), a(i), 0.0, (2•1), time 
       pat' 1-10 for 21 
         par 
            so.ap ( p.c(i), q, p, (2•1), time 
            s_i.b (b_c(i), b, i, time)
       syst ( a.c, p.e, b.e, n, time ) : 
\sim --- Main
var float a(no), p, q, b(4) : 
var d : 
seq 
  get ( d, " degree of polynomial " )fp.get.n (a, (d+1)), " polynomial coeffs " )
   fp.get ( p, " p " )<br>fp.get ( q, " q " )<br>driv ( a, p, q, (d+1), b )
  fp.put.n ( b, 4, " results " )
```

```
-- A.1.6\sim-- Systolic Array for the calculation of the characteristic
-- polynomial of an Lower Hessenberg matrix.
 \overline{a}external proc get ( var v, value s( ) ) :
 external proc fp.get ( var float v, value s( ) :
. external 
proc fp.9et.n (vac float v(], value n, s[) ) : 
 external 
proe fp.put.n (value float v(), value n, s(J ) 
 external 
proc fp.br ( chan xin, xoutl, 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, vac 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,<br>value time ) =
   var float h[2], y[2], u[2]:<br>sed
     - initialise
     par i•(O for 2] 
       par 
         h(1) := 0.0y(1) := 0.0ul1 := 0.0main operation 
     seq i=[0 for time]
        seq 
          -- i/O 
          par, hin 2 h(0)hin ? h(OJ yin 1 y(O] uin ? u(O) 
            hout 1 h{l] 
            yout 1 y(1)<br>uout 1 u[1]
          -- calculation
          par 
            y(1) := y(0) + (h(0) * u(0))h(1) : h(0)u(1) := u(0):
 -- Boundary cell
 proc bound ( ehan hin, hout, yin, uout, 
                value float z,<br>value time ) =
   var float h(2), y, u :<br>seq<br>-- initialise
      par 
        par i•(O for 21 
          h(i) := 0.0y := 1.0<br>y := 0.0-- main operation
      seq i=[0 for time]<br>seq
          \frac{-1}{\text{par}}hin 7 h(0) yin 7 y
```
 $\sim$ 

```
hout 1 h(1)<br>uout 1 u-- calculation
            \frac{1}{1}<br>\frac{1}{1} \frac{1}{2} = 1
                 u := (y - (z + u)) / (-h(0))h(1) := h(0) :
--- Linear array configuration
\sim \simproc bdarr ( chan hio[ ], uout,<br>value float z,
   value n, time ) =<br>chan y.c(no), u.c(no+l) :<br>par
      par i = \{1 \} for (n-1)]
      bdips ( hio(i), hio[n+i], y.c[i], y.c[i-1],<br>u.c[i], u.c[i+1], time )<br>bound ( hio[0], hio[n], y.c[0], u.c[0], z, time )
      fp.br (u.c(0), u.c(1), u_0 ut, time )fp.so.d ( y.c(n-1), time ) f(p,si,d \ (u,c(n), time) :\overline{\phantom{a}}-- System configuration
proc syst ( value float h( ), z,<br>var float u( ),<br>value n, time ) =
   chan h.c(2*no), u.c;
     par i•{O for n) 
         par.
            var float h.temp[to] :
            seq 
               seq j•(O for time) h.temp(j) :• h((i•time)+j) fp.so.x ( h,c(iJ. h.temp, time 
            fp.si.d (n+1), time )
      bdarr ( h.c, u.c, z, n, time ) fp.si.x ( u.c, u, time ) :
\sim-- Main
-var float h(no*to), z, u(to):
var d, time :<br>seq
   get ( d, " order of lower hessenberg matrix " )<br>time := (2 * d) + 1fp.get.n ( h, ((d+1)*time), * diagonal seq - upper first * )<br>fp.qet ( z, * evaluation point * )
   syst ( h, z, u, (d+l), time ) 
   fp.put.n ( u, time, " result " )
```

```
\mathbf{I}U1
\infty.... \blacksquare
```
 $- - A.2.1$  $--$  Preprocessor for the systolic array performing block<br> $--$  (2x2) R+F LU/LDU decomposition. Two control signals are  $-$ - used: cl, to align all elements of a submatrix in the  $-$  same liner 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[] ) :<br>external proc get.n ( var v[], value n, s[] ) :<br>external proc fp.put.n ( var float v[], value n, s[] ) :<br>external proc fp.put.n ( value float v[], value n, s[] ) :<br>external proc so. external proc fp.so.x ( chan xout, value float x[], value time ) :<br>external proc si.d ( chan xin, value time ) : external proc fp.si.d ( chan xin, value time ) :<br>external proc fp.si.x ( chan xin, var float x( ), value time ) :<br>external proc dl ( chan xin, xout, value d, time ) : def no =  $10$ , to =  $(no+6)$ : -- Demultiplexer for the alignement of the submatrix elements -- Reformatting delays are configured, according to switch. proc all ( chan xin, xout, ain, left, right,<br>value switch, time ) =<br>var float a, 1, r, reg[4] :<br>var x[2] :<br> $\frac{1}{2}$  var x(2] : -- initialisation  $a := 0.0$  $1 := 0.0$ <br>r := 0.0 par 1•(0 for 4) reg(i) :• 0.0 par  $i=[0 \text{ for } 2]$ <br> $x[i] := 0$ -- main operation seq i={0 for time}<br>seq -- input/output pa<  $xln 7 x[0]$ xout l x(l] ain 7 a<br>left I 1<br>right 1 r -- calculation  $x[1] := x[0]$ if If  $x[0] - 1$ par  $req[0] := 0$ reg(l) :• a true par  $reg[0] := a$ <br> $reg[1] := 0$  $switch = 0$ ••q 1 :• reg(2) r :• reg(3)  $reg[2] := reg[0]$ 

 $\overline{\phantom{a}}$ 

reg(l) *:•* reg(ll true  $\begin{array}{rl} \bar{1} := \mathrm{reg}(0) \ \text{r} := \mathrm{reg}(3) \ \text{req}(3) := \mathrm{req}(2) \end{array}$  $req(2) :=  $req(1)$ :$ -- Delay for correct spacing of submatrices. -- Reformatting delays are configured, according to switch. proc spa ( chan ain[ ], aout[ ], xin, xout,<br>value basein, baseout, switch, time ) = var float a0[4], al[4], a2[4], a3[4], a4(4) : var x[2], overwrite : seq -- initialisation par par i•(O for 4) par  $a0111 := 0.0$  $a1[i] := 0.0$  $a2(1) := 0.0$  $a3$ ii := 0.0  $a4[i] := 0.0$ par  $i=[0$  for 2)<br> $x[i] := 0$ overwrite := false<br>-- main operation  $seq$  i= $[0$  for time]<br>seq -- input/output par par i•(O for 41 par  $ain[basein+i]$   $? a0(i)$  $a$ out(baseout+i) i  $a4(i)$ <br>xin 7 x[0]<br>xout i x[1] -- calculation  $x[1] := x[0]$ if if  $x[0] - 1$ par par i•(O for 4) par  $a2[i] := 0.0$ <br> $a1[i] := a0[i]$ overwrite := true true seq if overwrite par i•(O for 4]  $a2(i) := a1[i]$ true par i•(O for 41  $a2[i] := a0[i]$ overwrite :• false  $s$ witch  $= 0$ par i•IO for 4) seq true  $a4[i]$  :=  $a3[i]$ <br> $a3[i]$  :=  $a2[i]$ 

 $\mathbf{I}$  $\mathbf{u}$  $\bar{\bm{\infty}}$  $\overline{a}$  $\blacksquare$ 

```
par i=[0 \text{ for } 4]<br>a4(i) := a2(i):
-- System configuration, for block tridiagonal matrix
proc system ( var float block(), 
                   value float point(),<br>value cl[], c2[], time ) =chan ain[7], a.c[14], aout[12], cl.c[8], c2.c[7] :
  par<br>-- data sources
     par i=[0 for 7]<br>var float temp.point[to] :<br>seq
     •• seq j=[0 for time]<br>•• temp.point[j] := point[(i*time)+j]<br>fp.so.x ( ain[i], temp.point, time )<br>•• control sources
     so.x ( cl.c(0), cl, time )<br>so.x ( c2.c(0), c2, time )<br>-- preprocessor
     par i•(O for 7) 
        ali ( cl.c(i), cl.c[i+l), aln(i), a.c((2*i)+l), a.c(2*i), 
                (1\sqrt{2}), time )
     par i•(O for 31 
        par 
           spa ( a.c, aout, c2.c[2•i), c2.c((2*i)+l), 
                   ((4•1)+1), (4*1), (i\2), time ) 
           dl ( c2.c[(2*i)+1], c2.c[(2*i)+2], 1, time )
      -- data sinks<br>par i=[0 for 12]
        var float temp.block(to) :<br>sed
           fp, sl.x ( aout[i], temp.block, time )
           seq j•[O for time) 
              block([i*time]+j] := temp.hlock[j]fp.si.d ( a.c{O), time ) 
      fp.si.d ( a.c(13), time )<br>-- control sinks
      s_i.d ( c1.c[7], time ) ;<br>s_i.d ( c2.c[6], time ) ;
\simMain 
var float point[7*to], block[12*to] :<br>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 " )
   get.n ( cl, time, " cl " ) 
get.n ( c2, time, " c2 " ) 
   system ( block, point, cl, c2, time ) 
   fp.put.n ( block, (12*time), " block output " )
```

```
\sim- A.2.2Systolic array for block (2x2) R+F LU/LDU decomposition 
-- of a (2x2) block tridiagonal matrix.
external proc get ( var v, value s[] ) :<br>external proc fp.get.n ( var float v[], value n, s[] ) :
external proc fp.put.n ( value float v(1, x) 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 ) :<br>def no = 10, to = ( no + 6 ) :<br>=
-- Inner Product calculation for (2x2) submatrices :<br>-- A = A + B * C.
proc block.ip ( value float b[], c[], var float a[] \prime = par
     a[0] := (( b[0] * c[0] ) + ( b[1] * c[2) )) + a[0]<br>a[1] := (( b[0] * c[1] ) + ( b[1] * c[3] )) + a[1]<br>a[2] := (( b[2] * c[0] ) + ( b[3] * c[2] )) + a[2]<br>a[3] := (( b[2] * c[1] ) + ( b[3] * c[3] )) + a[3]
     all i = (i + c)(i) + c(i) + (i + c)(3) + c(i) + (i + c)(3) + (i + c)(3) + c(2) + c(3) + c(2) + c(3) + c(2) + c(3) + c(4)a(3) := ((b(2) * c(1)) + (b(3) * c(3)) + a(3):
\sim -Calculation of the Determinant of a (2x2) matrix. 
proc det ( value float a(), var float dt ) \approx 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 \leftrightarrow 0.0
       pa< 
          ai[0] := a[3] / dtai(1) : ((-a(1)) ) \neq dtai(2) := (-a(2)) / dt
          al[3] := a[0] / dttrue 
        par l•{O for 4) ai(i) :• 0.0 : 
- Processor sll ; accepts a (2x2) matrix A and its
determinant d, and produces A/d. 
proc sll ( chan ain[], dtin, aout[],<br>value time \} =var float a0\{4\}, al\{4\}, dt :
  seq 
     -- initialisation 
     pa< 
       par i=[0 \text{ for } 4]<br>par
             a0[i] := 0.0<br>a1[i] := 0.0dt := 0.0<br>-- main operation
     seq j=[0 for time)
       seq 
       . -- i/o<br>par
             par i•(O for 4) 
                par 
                   ain[i] 7 a0[i]aout[i] i al[i]
```
 $\mathcal{L}$ 

 $\sim$ 

# $\overline{u}$ 00 0\  $\mathbf{f}$

```
dtin ? dt<br>-- calculation
          if 
             dt \leftrightarrow 0.0
                par 1•(0 for 4) 
                  al[i] := a0(1) / dttrue 
                par i•[O for 4) 
                   al[1] := 0.0 :
Processor sl2 : 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,<br>value time ) =var float a0(4), al[4), a2{4}, dt[2] :<br>seq
     -- initialisation
     par 
        par i•lO for 4] 
          par 
             a0[1] := 0.0al[i] := 0.0<br>a2[i] := 0.0\begin{array}{cc} \texttt{par} & \texttt{1} \texttt{=} \{0 \text{ for } 2\} \\ \texttt{d} \texttt{t} \{1\} & \texttt{:= } 0.0 \end{array}-- main operation 
     seq i-(0 for time)
        seq 
           -1/0par 
             par 1-(0 for 4)<br>par
                   ain[i] ? a0[i]<br>aout[i] 1 a2[i]dtin 2 dt[0]dtoutl I dt(l) 
           dtout2 l dt(l) -- calculation 
           par i•{O for 41 
              a2[i] := a1[i]par i•IO for 41 
             al(i) :• aO(i) 
           dt[1] \div dt[0] \div--
 Processor sl3 : accepts a (2x2) matrix A and its 
determinant d, and produces dA. 
 proc s13 ( chan ain[], dtin, aout[],<br>value time ) =var float a0[4], a1[4], dt :
    seq<br>-- initialisation
      par 
         par i•(O for 4) 
           par 
              a0[i] := 0.0\begin{array}{r} \text{a1(i)} := 0.0 \\ \text{d}t := 0.0 \end{array}-- main operation
      seq j=[0 for time)<br>seq
            - i / opar 
              par i•IO for 41
```
 $\blacksquare$ 

par ain(i) ? aO[il  $aout(i)$   $i$  alli<br>dtin  $i$  dt -- calculation par i•(O for 41 alii  $i = a0[i] + dt$ : -- 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  $\rangle$  = var float a0(4}, a1(4), u[4], dl(4) :<br>seq<br>-- initialisation par 1•[0 for 4) pa<  $a0[1] := 0.0$ al(i) :• 0.0  $u(i) := 0.0$ di $[i]$  : 0.0 main operation seq j•(O for time) seq -- i / <sup>0</sup> par i=(0 for 4)<br>par ain $[i]$  7 aO $(i)$  $u$ in $(i)$  7  $u[i]$ dloutl(i) 1 dl(i) dlout2 $(i)$  1 dl $(i)$ -- calculation inv ( u, 1.0, al I par  $i=[0 \text{ for } 4]$ <br>dl[i]  $=-0.0$ block.ip ( $at, a0, d1$ ) :  $\leftarrow$  Processor s22: accepts a (2x2) matrix, and outputs  $\leftarrow$  the matrix and its determinant proc s22 ( chan ain[ ], aoutl[ ], aout2[ ], dtout1, dtout2,<br>value time  $) =$ var float a0(4), al(4), dt *:*<br>seq<br>-- initialisation par par i•(O for 4) pa<  $a0[1]: -0.0$ al[i]  $:= 0.0$ <br>dt  $:= 0.0$ -- main operation seq j•(O for time)  $-1/0$ par par i•(O for 4) ain(i) 7 aO(i) par i•(O for 4) par  $a$ outl $[i]$   $i$  al $[i]$ aout2[i] i al[i]<br>dtoutl i dt dtout2 1 dt

 $\mathbf{u}$  $\infty$  $\overline{a}$ 

 $\mathbf{I}$ 

```
-- calculation 
         par i•(O for 41 
           al(i) •• aOiil 
          det \overline{a}\overline{0}, dt \overline{)} :
Processor s23 ; accepts a (2x2) matrix A and its 
-- determinant d, and produces A/d.
proc s23 ( chan ain[], dtin, aout1[], aout2[],<br>value time ) =var float a0(4), a1(4), dt :<br>seq
     -- initialisation
    par 
       par i=[0 \text{ for } 4]a0[1] := 0.0al(i) := 0.0<br>dt := 0.0
     \frac{1}{2} main operation
     seq j=(0 for time)
        \frac{seq}{1 - i} / 0<br>par
            par i=[0 for 4]<br>par
                 par ain(i) ? aO(i] 
                  aoutl[i] i al[i]aout2[i] i al[i]<br>dtin ? dt
           -- calculation
          if 
             dt \leftrightarrow 0.0
               par i•[O for 4) 
                  al[i] := a0[i] / dttrue 
                par i•IO for 4] 
                  allil := 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):<br>seq
      initialisation
      par i=[0 for 4]<br>par
           a0[i] := 0.0all11 := 0.0a2[i] :- 0.0
           1[i] := 0.0uli) : - 0.0
       -- main operation
       seq j•(O for time] 
         seq<br>-- i/ o<br>par i=[0 for 4]
              \frac{\text{par}}{\text{ain}} 2 a0(i)
                lln[l) ? 1111 
                uin(i) ? u(il 
                 aout[i] 1 a2[i]
            calculation
```

```
par i•IO for 4) 
              2111 - 21111par i=[0 for 4]<br>allil := a0[i]
            block.ip ( 1, u, a1 ) :
-- System configuration
proc system (value float ain1[], ain2[], ain3[],<br>var float aout1[], aout2[], aout3[],<br>value time ) =
   value time ) • chan cllo[4), cl2o(4), c13o(4), cl21ld, c1213d, 
   par<br>-- sources
           c2111(4], c2212(4), c2212d, c2313[4], c2221[4], e2223d, 
           c3122(4), c2131(4), c2331[4), c21il4), c23i(4), c3li[4) 
      par i=[0 \text{ for } 4]<br>var float temp.a(to):
          seq<br>seq j=[0 for time]
       temp.a(j) := ainl[(i*time)+j]<br>fp.so.x ( c23i(i), temp.a, time)<br>par i=[0 for 4]
          var float temp.a[to] :<br>seq<br>seq j=[0 for time]
       temp.a(j) := ain2((i*time)+j)<br>fp.so.x ( c3li(i), temp.a, time)<br>par i=(0 for 4)
          var float temp.a(to) :<br>seq<br>seq j=(O for time)
                \ti{emp.a[j]} := \n  ain3[(i*time)+j]fp.so.x (c21i(i), temp.a, time)
        -1 array<br>sll ( c2111, c1211d, c11o, time )<br>s12 ( c2212, c2212d, c12o, c1211d, c1213d, time )<br>s13 ( c2313, c1213d, c13o, time )
        s21 ( c211, c2221, c2111, c2131, time ) s22 ( c3122, c2221, c2212, c2212d, c2223d, time )
        s23 ( c23i, c2223d, c2313, c2331, time). 
        s31 ( c3li, c2131, c2331, c3122, time ) 
        -- sinks<br>par i=[0 for 4]
           var float temp.a[to] :<br>seq
              \{p, si.x \in c13o(1), temp.a, time\}<br>seq j=[0 \text{ for time}]souti[(i*time)+j] := temp.a[j]<br>par i=[0 for 4]
           var float temp.a[to] :
            seq<br>fp.si.x ( c12o[i], temp.a, time )<br>seq j=[0 for time]
         sout2[(i*time)+j] := temp.a[j]<br>par i=[0 for 4]
            var float temp.a(to) :
            seq 
   Main 
               fp.si.x ( c11o(i), temp.a, time )<br>seq j=[0 for time]
                  sout3[(i*time)+j] := temp.a[j] :
   var float ainl(4*to), ain2(4•to), ain3(4*to), 
                  aoutl(4*to), aout2(4•to), aout3(4*to)
```
 $\mathbf{r}$ 

# $\overline{\mathbf{u}}$  $\infty$  $\bullet$  $\mathbf{I}$

 $\blacksquare$ 

```
var n, time : seq 
    get ( n, " problem size " )
    time :• n + 6
     fp.get.n ( ainl, (4*time), " al input seq " )<br>fp.get.n ( ain2, (4*time), " a2 input seq " )<br>fp.get.n ( ain3, (4*time), " a3 input seq " )
    system ( ain1, ain2, ain3, aout1, aout2, aout3, time )
     fp.put.n ( aoutl, (4*time), " al output seq " )<br>fp.put.n ( aout2, (4*time), " a2 output seq " )<br>fp.put.n ( aout3, (4*time), " a3 output seq " )
```

```
- - A.2.3ЦÚ.
-- 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{] ) :<br>external proc fp.get.n ( var float v[], value n, s{} ) :<br>external proc fp.put.n ( value float v(], value n, s{} ) :
external proc fp.so.x ( chan xout, value float x(), value time ) :<br>external proc fp.si.x ( chan xin, var float x(j, value time ) :<br>def no - 10, to = ( no + S ) :
-- Inner Product calculation for (2x2) submatrix, and -- (2x1) subvetors : y = A x + 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 ) =<br>seq<br>dt := ( a[0] * a[3] ) - ( a[1] * a (2] ) :
-- Inversion of a (2x2) matrix. 
proc inv ( value float a[], dt, var float ai[] ) =<br>if
      dt <> 0.0 
         par 
           ai[0] := a[3] / dt
            ai(1) := (-a(1)) / dtai[2] : (-a(2)) \neq dt
            al(3) := a(0) / dttrue 
         par i•IO for 4) 
            ai(1) := 0.0 \in-- Processor s22: accepts a (2x2) matrix, and outputs
 -- the matrix and its determinant
 proc s22 ( chan ain( ), aout[ ], dtout,<br>value time ) =var float a0(4), al[4], dt :<br>seq<br>-- initialisation
      par 
         par i=[0 for 4]<br>par
               a0[i] := 0.0al[i] := 0.0<br>dt := 0.0-- main operation
       seq j•(O for time) 
         seq 
            --i/o<br>par
               par i-10 for 4) ain[i) 7 aO{i] 
               par i=(0 for 4)<br>aout[i] 1 al[i]<br>dtout 1 dt
             -- calculation
```
## **un**  $\infty$ **lo**  $\blacksquare$

```
par 1•[0 for 4] 
            51111 - 80111det ( aO, dt ) : 
-- Processor s23 : accepts a (2x2) matrix and its determinant
and it produces its inverse.
proc s23 ( chan ain[], dtin, aout[],<br>value time ) =
   var float a0(4), al(4), dt :
   seq<br>-- initialisation
     par 
        par 1•[0 for 4) 
           par 
             a0[i] := 0.0alii) := 0.0<br>dt := 0.0-- main operation
      seq j-[0 for time]<br>seq
           -- i/o<br>par
              par 1•[0 for 41 
                ain(i) ? a0(i)dtin 7 dt 
            par i=[0 \text{ for } 4]<br>aout(i) i al[i]<br>-- calculation
            inv ( aO, 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) :
     seq -- initialisation 
       par 
          par i•[O fot 4) 
            a[1] := 0.0par i•[O for 2) 
             par 
               b[i] := 0.0y[i] : - 0.0
               x[i] := 0.0id[0] := 1.0ldlll ·- 0.0 
           id[2] := 0.0<br>id[3] := 1.0-- main opertaion
        seq j•[O for time) 
           \begin{array}{c} \n\text{seq} \\
-1/0\n\end{array}par 
                par i•[O for 4} 
                  ain[i] 7 a[i]par i=[0 for 2]<br>par
                     bin[i] 2 b(1)\frac{\text{yin}[i] \times \text{y}[i]}{\text{xout}[i] \times \text{x}[i]}-- calculation
              par i•[O for 2) 
                por
```
 $y[i] := (-y[i])$  $x[i] := 0.0$ <br>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} :  $seq$   $-$  initialisation par par i•(O for 4)  $a11$  := 0.0 par i•(O for 2) par  $x0(1) := 0.0$  $x1(1) := 0.0$ y(i) := 0.0<br>-- main operation seq j=[0 for time]<br>seq  $-$  i / 0 par par  $i=[0 \text{ for } 4]$ <br>ain[i] 7 a[i] par i•(O for 2] xin(i) 7 x0[i]<br>xout[i] 1 x1[i]  $vout[i]$   $\mid$   $y[i]$ -- calculation par 1•[0 for 2]  $x1111 := x011$  $v[i] := 0.0$ block.ip (  $a, x0, y$  ): -- System configuration. proc system ( value float al(), a2(}, b(),<br>var float x(),<br>value time ) = value time ) • chan as22(4), as23(4), asd[4], ama[4], dts23, bsd(2), ysd[2], xsd[2], xmal21 : par<br>-- sources<br>-- par i=[O for 4]<br>- var float temp.al[to] : seq<br>seq j=[0 for time] temp.al[j] := al[(i\*time)+j]<br>fp.so.x ( as22[i], temp.al, time )<br>par i= $(0 \text{ for } 4)$ var float temp.a2(to) : seq<br>seq j=[0 for time]  $temp.a2[j] := a2[(i*time)+j]$ fp.so.x ( $\text{amalil}$ , temp.a2, time)<br>par i= $[0 \text{ for } 2]$ var float temp.b[to] :<br>seq<br>seq j=[0 for time]  $s_{\text{emp},b(j)} = b((i * t \text{times}) + j)$ 

# ..,  $\overline{\mathbf{o}}$  $\bar{\bullet}$  $\mathbf{1}$

 $\mathbf{L}$ 

fp.so.x  $\{ \text{bsd}[i], \text{temp.b, time} \}$  -- array  $s22$  ( $s22$ ,  $s23$ , dts23, time )<br> $s23$  ( $s23$ , dts23, asd, time ) sd ( asd, bsd, ysd, xsd, time )<br>ma ( ama, xsd, xma, ysd, time ) -- sink<br>par i=[0 for 2]  $var$  float temp.x(to) :<br>seq  $\hat{fp}.$ si.x ( xma[i], temp.x, time )<br>seq j={0 for time}  $x$  $(i + t)$   $i + j$   $i -$  temp.x $(i)$  : ---- Main  $$ var float al(4\*to), a2(4\*to), b(2\*to), x(2\*to) :<br>var n, time :<br>seq<br>get ( n, \* size of problem \* )<br>time := (n + 5) time :• (n + S) fp.get.n ( al, (4\*time), " al input seq fp.get.n ( al, (4\*time), "a2 input seq fp.get.n ( b, (2•time), "b input seq" system ( al, a2, b, x, time )<br>fp.put.n ( x, (2\*time), " x output seq " )

 $- A.2.4$  $\overline{\phantom{a}}$ -- 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[] ) :<br>external proc fp.get.n ( var float v(), value n, s[] ) :<br>external proc fp.put.n ( value float v(), value n, s[] ) : -- Maximum size of problem, def no  $-10$ : -- Diagonal cell. proc diaq ( var float u, eo, se, wo, value float nw, n, ei, value start,  $t \rightarrow -$ If t • start if  $nw = 0.0$  $\frac{1}{2}$  wo  $i = 0.0$ wo  $t = (u / nw)$  $t = (start + 1)$ seq par u :• ei - (wo • n) eo :• wo se :• u : Upper Diagonal cell.  $-1$ proc updi ( var float u, eo, s, wo, value float n, ei, wi, value start,  $t$  )  $=$ If t = start WO :• U  $t = (start + 2)$ seq par  $eo := wi$  $u := ei - (wl * n)$  $s := u :$  $-$ -- First Lower Diagonal cell.  $\overline{\phantom{a}}$ proc ldil ( var float 1, s, value float ei, wo,  $\frac{1}{\text{w}}\text{w}$ <br>if value start,  $t$  )  $=$  $\text{if} \quad t = \text{start} \quad \text{set} \quad \text{set}$  $s = (1 + ei)$ <br> $s := ei$ ; Lower Diagonal cell. proc lodi ( var float 1, s, value float n, ei, wo,  $\sum_{\omega_0}$ value start,  $t$  )  $=$ If  $t - start$ seq

ლ<br>ბ "' ....  $\mathbf{I}$ 

```
1 := 1 + (n + e i)<br>s := n :
-- Generic cell for main diagonal.
proc gend ( chan nwin, nin, eout, ein, seout, sout, wout, win, 
                 var float x,<br>value start, time ) =
   var float nw, n, eo, ei, se, s, wo, wi :
   seq t•(O for time) 
      seq<br>par
           nwin ? nw<br>nin ? n<br>eout ! eo
           ein 7 ei
            seout I se<br>sout I s<br>wout I wo<br>win 7 wi
         diag(x, eo, se, wo, nw, n, ei, start, t) :
Generic cell for other dia9onals. 
proc gene ( chan nin, eout, ein, sout, wout, win,<br>var float x,<br>value type, start, time ) =
    var float n, eo, ei, s, wo, wi :<br>seq t=[0 for time]
       seq<br>par
            par nin 1 n
            eout leo<br>ein ? ein<br>sout ! $
             wout I wo<br>win 7 wi
          if<br>type = 0t_{\text{update}} (x, eo, s, wo, n, ei, wi, start, t)
             type • 1 
                ldi(x, s, e), distact, t)true 
                lodi(x_t, s_t, n_t, ei, wastart, t) :
  -- Dummy source.
  proc soud ( chan xout,<br>value time ) =<br>seq t=[0 for time]
       xout I 0.0 
  -- Dummy sink.
  proc sind ( chan xin,<br>value time ) =
     seq t={0 for time}
        xin 7 any :
   -- Source for elements of new column a.
   proc soua ( chan xout, 
                   value float x, 
                   value start, time ) -
      seq t=[0 for time]<br>if
```

```
t .. start 
        xout I x<br>true
          xout 1 0.0 
-- System configuration.
proc syst ( var float lu().
                value float a(),<br>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.<br>--
   proc sqar ( var float lu(], 
     value n, time ) =<br>par i=[0 for n]
        par j={0 for n}<br>var k :<br>seq
              \begin{array}{l} k := (i * n) + j \\ i f \end{array}k := (1 * n) + j<br>if<br>i = j -- main diagonal cells
                    gend(d.c(i), v.c[k}, hl.c[(k+1)+i], h2.c[(k+1)+i], d.c[1+1], v.c[k+n], h2.c[k+i], h1.c[k+il, lu[k], (2*1), time)
                 v.c(k+n), h2.c(k+1), h1.c(k+1), lu(k), (2+1), time)<br>i < i -- upper diagonal cells
                   gene(v.c(k ), hl.c( (k+l )+i), h2.c( (k+l )+i),v.c(k+n], 
                 h^2 = h^2 \cdot c(k+1), \quad h^2 \cdot c(k+1), \quad lu(k), \quad 0, \quad (i+1)-1), \quad t \neq 0<br>i = (j+1) -- first lower diagonal cells
                   gene(v.c(k), h1.c[(k+l)•i), h2.c((k+l)+i],v.c(k+n), 
                          h2.c(k+i], hl.c[k+i], lu(k), l, ((2~1)+1), time) 
                 true -- rest lower diagonal cells 
                    gene(v.c{k), hl.c[(k+l)+i), h2;c((k•ll+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•(O for n] 
            par<br>soud(v.c(i), time)<br>sind(v.c((n*n)+i), time)<br>sind(hl.c(((n+l)*i)+n], time)
               soua(h2, c)((n+1)*i)+n), a(i), (n+1), time)sind(h2.c((n+l)~i), time) 
          soud(d.c{O), time) 
         sind(d.c(n), time):
    --
    par<br>- sqar(lu, n, time)
       sosi(a, n, time) : 
  -- Main.
  --<br>var float lu[no*no], a[no] :<br>var n, time :<br>seq<br>qet(n, " problem size ")
    fp.get.n(lu, (n•n), • LU row-wise •J 
    fp.get.n(a, n, " new column")time :• (2 ~ n) 
    syst(lu, a, n, time) 
     fp.put.n(lu, (n~n), • LU updated •)
```
**GT** Ō  $\overline{M}$ л

```
-- 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).<br>-- For the updating: the new nth column in the form
F - a = ((L)*-1)An is added to U and the main diagonal is
-- removed; L is modified accordingly. It is assumed that
no permutations needed. 
\simexternal proc get ( var n, value s[] ) :<br>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 i^{(i)}(i,i-1) := i^{(i)}(i,i-1) + r.
 proc dva ( chan ein, win, sin, eout, wout, nout,<br>value time )=
   var float r, u\{2\}, 1\{2\} :
      - initialisation
      par 
        par i=[0 for 2]<br>par
             u[i] := 0.0\frac{1}{100} ; = 0.0
      -~ main operation 
      seq i•(O for time) 
         \frac{seq}{-1/0}ein 7 u[1]
              win 7 1(0)
              \sin 2 u\sineout 1 r 
              nout 1 1(1)
            -- calculations 
            if u(1) = 0.0r := 0.0<br>true
                 t := (u(0) / u(1))1111 := (1101 + r)^{-1}\rightarrow 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,<br>value time )=
     var float r(2), u(3) :
     seq<br>-- initialisation
       par 
          par i•IO for 21 
            r(i) := 0.0par i=[0 for 3)<br>ufil := 0.0-- main operation
        seq i•[O for time)
```
seq <sup>1</sup>- i/o<br>par ein  $2$  u(1) win  $7$  r( $0$ )  $\sin 2$  u(O) eout I rill wout  $1 \le |1|$ <br>nout  $1 \le |1|$ -- calculation<br>par  $u(2) := u(0) - (u(1) * r(0))$  $r(1) = r(0)$ : Inner Product calculating *:* 1'(1,jl :• l[i,j) + r • l(i,j+l) -- Propagates 1(1,j+1), r in opposite directions. proc ipl ( chan ein, win, sin, eout, wout, nout,<br>value time )= var float  $l(4)$ , r $l2$ ) :<br>seq seq -- initialisation par par  $i=[0 \text{ for } 4]$ <br> $1[i] := 0.0$  $par_i = (0 for 2)$ <br>  $r[i] := 0.0$ -- main operation<br>seq i=[O for time]<br>seq  $-1/0$ par ein  $7$  r $[0]$ <br>win  $7$  1(1) sin 2 1101 eout I 1(2) wout 1 r[l) nout 1 1(3} -- calculation  $p_1(3) := 1(1) + (r(0) * 1(0))$  $1[2]$  :-  $1[0]$  $r(1) := r(0)$ : ---- Delay-branching cell. proc dba ( chan ain, u1out, u2out, var float a(2) :<br>seq<br>-- initialisation par i•(O for 2) a(i) :• 0.0 -- main operation seq i•{O for time) seq --i/o<br>par ain 7 a[0] ulout  $t$  a(1)<br>u2out  $t$  a(1) -- calculation  $a[1] := a[0] :$ -- Source for a vector x.

 $\ddot{\phantom{1}}$ 

#### $\mathbf{L}$ **un** Ö ω  $\blacksquare$

```
proc srx (chan xout ,<br>value float x[],<br>value time ) -
   seq i=[0 for time]
       xout 1 x(il : 
-- Sink for a vector x. 
proc six ( chan xin,<br>var float x( ),<br>value time ) =
   seq i=[0 \text{ for time}]xin ? x[i] :
\sim-- Dumm sink. 
proc sid ( chan xin,<br>value time ) =
    seq i=[0 \text{ for time}]xin 7 any : 
-- Delay cell. 
proc del ( chan xin, xout,<br>value time ) =
   var float x(2) :<br>seq
       par i•(O for 2) 
          x!il :• 0.0 
       seq i•(O for time) 
           seq<br>par
                 xin 7 \times 101xout 1 \times [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[], ul[],<br>value float 1[], u[],<br>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<br>-- n vector sources / sinks for U-diagonals; the source
       → vectors have last element from new column a.<br>par i=[0 for n]<br>var float uti(to), uto(to) :<br>seq
               par j•[O for time] 
               \text{util} := u[(i*time)+j]<br>par
                  par srx(ui.c{i), uti, time) six(uo.c(i), uto, time) 
               par j•(O for time) 
       -- 1 vector source for the first element of new column a.<br>var float uti[to] :<br>seq
           par j•(O for time) 
           \begin{bmatrix} \text{utifj} & \text{:= } \text{u} \text{((time+n)+j)} \\ \text{srx(ui,c[n], uti, time)} \end{bmatrix}
```
-- n-1 vector sources / sinks for L-diagonals.<br>par i= (0 for (n-1)) var float lti{to}, lto[to] : seq<br>par j=[0 for time]  $\frac{1}{\text{pi}}$  iti[j] := l{(i\*time)+j]<br>par par<br>srx(li.c(i), lti, time)<br>six(lo.c[i], lto, time) par j= $[0 \text{ for time}]$ <br> $11[(i+time)+j] := 1to[j]$ -- 2 dummy vector sinks for r.<br>sid(ru.c(n-1), time)<br>sid(rl.c(n-2), time) --delay-branching cell for the first element of column a..<br>
dba(ui.c{n}, u.c{n-1}, uo.c{n-1}, time}<br>
-- delay for lowest L-diagonal.<br>
·· main systolic array.<br>
-- main systolic array.<br>
par i={0 for (n-1)}  $ipu(u.c(i+1), vu.c(i), ui.c(i+1), vu.c(i+1), u.c(i), wo.c(i),$ time)<br>par  $i=[0 \text{ for } (n-2)]$  $ipl$ (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), r1.c(0), lo.c(0), time) i --Main --<br>var float l[(no-1)\*to), u[(no+1)\*to], ll[(no-1)\*to], ul[no\*to] :<br>var n, time : seq<br>get(n, " size of matrix ")  $time := (2 * n) + 1$ fp.get.n(u, ((n+1)\*time), " U data seq ")<br>fp.get.n(l, ((n-1)\*time), " L data seq ")<br>system(11, ul, l, u, n, time) fp.put.n(u1, (n\*time), " Ul data seq ")<br>fp.put.n(ll, ((n-l)\*time), " Ll data seq ")

t ហ **O**  $\sim$ 

```
-- A.2.6\overline{\phantom{0}}-- 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[]) :<br>external proc fp.get.n (var float v[), value n, s[]) :<br>external proc fp.put.n (value float v[], value n, s[]) :<br>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,<br>value time )=
   var float u(2), b(2), p(2), m, x :<br>var c:
   seq<br>-- initialisation
      par 
         par i=[0 for 2]<br>par
              u[i] := 0.0b[i] := 0.0p(i) := 0.0\begin{array}{c}\n m : = 0.0 \\
 x : = 0.0 \\
 c : = 0.0\n\end{array}-- main operation
      seq i={0 for time}<br>seq
        seq -- input / output
            par 
               uin 7 u|0)
               bin 1 b(O) 
               pout 1 p[0]<br>mout 1 m
            cout 1 c<br>-- calculation
            par 
               If 
                  u[OJ < 0.0 
                    u(1) := (-u(0))true 
                    u(1) := u(0)If 
                  b[0] < 0.00] < 0.0<br>b(1} := (-b(0})
                  true 
                     ue<br>b{1} := b[0]
            If 
               b(1) >- u(1)<br>par
                    par c :• 1
```
 $-$ 

If p $[0]$  := b $[0]$ <br>x := u $[0]$ true  $c := 0$  $p(0) := u(0)$ <br> $x := b(0)$  $p(0) = 0.0$  $p(1) := 0.000001$ true p(1) := p(0)<br>m := x/p[1] : -- IPS cell producing one off-diagonal element (q,r)  $-$  for given  $c,n$  and  $v,a$ . proc qr ( chan vin, cin, min, ain, vout, cout, mout, qout, value time ) = var float v(2), m(2), a, q :<br>var c(2) :<br>seq<br>-- initialisation par par i=[0 for 2]<br>par  $v(i) := 0.0$ m{i] :• 0.0 c(1) := 0<br>a := 0.0<br>q := 0.0 -- main operation seq i=(0 for time)<br>seq -- input / output<br>par  $v$ in ?  $v(0)$ <br>cin ? c $(0)$  $min$   $7 \text{ m}$  $0$ )  $min$   $7 \text{ a}$ vout  $1 \vee 1$ cout  $1$  c( $1$ ) mout I m(l] qout l q -- calculation par c(1) := c[0]<br>m[1]  $:=$  m[0] if  $c(0) = 1$ par<br>q:=a q := a<br>v(1) := v(0) - (m(0) \* a) true  $\begin{bmatrix} \text{par} \\ \text{q} \end{bmatrix}$  =  $\mathsf{v}[0]$  $v(1) := a - (m[0] * v[0])$ : -- System configuration proc system ( var float p[], q[}, r[], m[],<br>value float a[], bu[], bl[],<br>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

სე<br>ს  $\bar{u}$  $\mathbf{r}$ 

 $\blacksquare$ 

pme ( u.e(O), bl.c, p.e, m.c(O), e.c(OJ. time )  $q_{\text{c}}$  ( u.c(1), c.c(0), m.c(0), a.c, u.c(0), c.c(1), m.c(l), q.c, time ) qr ( u.c(2), c.c(l), m.c(l), bu.c, u.c(l), 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.e, q, time ) fp.si.x ( r.c, r, time ) fp.si.x ( m.c(2}, m, time I si.x  $( c.c121, c, time )$ :  $-$ -- Main  $$ var float a(to), bu(to), bl(to), p(to), q(to), r(to), m(to);<br>var c(to), n, time; var c(to), n, time :<br>seq<br>get ( n, " size of matrix " ) time  $:= (2 + n) + 2$ time := (2 \* n) + 2<br>fp.get.n ( a, time, " main diagonal stream " )<br>fp.get.n ( bl, time, " lower diagonal stream "<br>fp.get.n ( bl, time, " lower diagonal stream " system  $(p, q, r, m, c, a, bu, bl, time)$ fp.put.n ( p, time, " main diagonal stream " )<br>fp.put.n ( q, time, " first upper diagonal stream " )<br>fp.put.n ( r, time, " multipliers stream " )<br>fp.put.n ( m, time, " multipliers stream " ) put.n ( c, time, " pivoting control stream " )

 $- - A.2.7$  $-$ -- Systolic Array for backsubstitution, for an upper<br>-- triangular matrix, with three diagonals.It is used for the -- second step of the Inverse Iteration method. Time : 2n+2.  $\overline{a}$ 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[]) :<br>external proc fp.put.n (value float v[], value n, s[]) :<br>external proc fp.so.d (chan xout, value time) : external proc fp.so.x ( chan xout, value float x[], value time ) :<br>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,<br>value time  $\rangle =$ var float a, x(2), y(2) :<br>seq<br>-- initialise pa< a :• 0.0 par i=[0 for 2]<br>par par x(i) *:•* 0.0 y(i] :•0.0 main operation seq i•(O for time) seq -- i/O par<br>ain 7 a  $x$ in  $2 \times 10$ yin ? y[OJ  $x$ out  $1$   $x$  $1$ ] yout ! y[1]<br>-- calculation par  $y(1) := y(0) + (a * x(0))$ <br> $x(1) := x(0)$ : -- Boundary cell  $\sim$ proc bound ( chan ain, bin, yin, xout,<br>value time  $) =$ var float a, b,  $y$ ,  $x :$ <br>seq<br>-- initialise par a :• 0.0  $b := 0.0$  $y := 0.0$ <br> $x := 0.0$ -- main operation seq i•(O for time] seq -- i/O par ain 7 a<br>bin 7 b

 $\mathbf{I}$ **un**  $\bullet$  $\sigma$  $\mathbf{1}$ 

```
yin ? y 
          -- calculation
          if a = 0.0x := 0.0x := (b - y) / a:
-- System configuration
proc syst ( value float a[), b[], var float x(1),
               value time ) -
  chan a.c[3], b.c, x.c[3], y.c[3]:<br>par
     -- sources<br>par i=[O for 3]<br>var float a.temp[to] :
       seq<br>seq j={0 for time}
          seq j•(O for time) a.temp(jl :• 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)<br>-- 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, \text{ time }\}:
--- Main
var float a(3*to), b(to), x(to) :<br>var n, time :<br>seq<br>get ( n, * order of matrix " )
   time := (2 \times n) + 2fp.get.n ( a, (3•time), "diagonal seq- main first " ) 
   fp.get.n ( b, time, " r.h.s vector seq " )
   syst ( a, b, x, time ) 
   fp.put.n ( x, time, " result " )
```

```
-- A.2.8--- Systolic Algorithm for a system of linearly<br>-- connected systolic processors.It calculates
-- the eigenvector of a symmetric tridiagonal<br>-- 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.<br>def n = 10:
                -- the three diagonals of the matrix<br>-- bl(0) = br(n-1} = 0,<br>-- 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),<br>-- eigenvector
x[n] :<br>-- Actual size of matrix.<br>var nl:
        -- Channels for m, c transfer
chan m.c(n+l], c.c(n+l), -- Not dedicated channels 
        ll.c(n+l], 12.c(n+l), 
        rl.c(n+l), r2.c[n+l) : 
 -- Process performing the calculations that<br>-- take place during the inverse iteration in
 - row i of the matrix.Initiallly 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.
 \overline{\phantom{a}}proc 
invit(chan min, cin, mout, cout, 
                         ill, il2, oll, ol2,<br>irl, ir2, or1, or2,
                  var float p, q, r, m, c, x,
                  value position, nll=
    var float temp[6] :<br>var ig, ib, t, tgl, tbl, tg2, tb2
    var float temp[6] :
    - State performing gauss elimination with<br>- partial pivoting. It produces P, an upper
    partial pivoting. Julian pick produces P, an upper triangular matrix with three diagonals stored in p, q, r.Also the multiplier matrix with the multiplier and the row-exchange control is calculated.
    proc gaussl-
       proc gaussi.calc=<br>seg
             - absolute values for b(i), u
              -- temp(3)-abs(b(i)l, temp(4)•abs(u) 
             par 
                if 
                    p \leqslant 0.0temp[3] := -p
```
 $\mathbf{f}$ ω ب  $\mathbf{r}$ 

```
If 
    If 
         true 
           temp(3) := pIf 
         temp(O) < 0.0 
           temp(4) := -temp(0)true<br>temp[4] := temp[0]
    - if abs(b[i]) >= abs(u) then c=l
      temp(3) > = temp(4)<br>temp[4] := 1.0
      true<br>temp[4] := 0.0- if c=0 swap p, q, r with u, v, w; w=0
    If 
       temp[4] - 0.0seq 
              temp(3) := temp(0)
               temp{S] :• temp(1) 
            temp[2] := 0.0
              temp(0) := ptemp[ll :• q
            temp(2) :• r par 
               p := temp(3)<br>q := temp(5)q = \text{temp}(2)<br>-- m \cdot u'/p' where u' and p' are the values of u = u and p after the interchange.Check first
    -- for zero<br>par<br>If
       If 
          temp(0) = 0.0temp(J) :• 0.000001 
          true 
             temp[3] := temp[0]
         p = 0.0temp(S] :• 0.000001 
          true 
             temp(5) := ptemp(3) := temp(3) / temp(5)<br>-- new u=v'-m*q' and new v=w'-m*r'<br>par
     par<br>temp[1] := temp[1] - (temp[3]*q)
        temp(2) := temp(2) - (temp(3)*c)-- gaussl control and i/o
  t = ig<br>par
        -- send a[i], b[i] to i-1 process
        -- b[i-1] := b[i]
        oll 1 q 
        ol2 1 r 
        p := rt = (1g + 1)seq<br>par
           -- receive m[i], c[i] and u, v from i-1
          -- and a(i+1), b(i+1) from i+1 process<br>min ? m
           cin ? c<br>ill ? temp[0]<br>il2 ? temp[1]
```

```
int_{1}^{1} int_{1}^{1} int_{1}^{2} int_{1}^{2}--now p=b[i], q=a[i+l], r=b[i+l]<br>-- temp[0]-u, temp[1]=v<br>qaussl.calc
     t = (ig + 2)<br>par
           - send m[i+l], c[i+l], u, v to i+l from - locations returned by gaussl.calc
           mout 1 temp(J) 
           cout 1 temp( 41 
           orl I temp( l] 
           or2 I temp(2) : 
-- State performing back substitution for
-- the system P * x = e, where P is the upper
- triangular matrix produced from gaussl and - e is the unity vector.
proc backl• 
  proc back1.calc• 
     seq 
        par<br>seq
               \frac{3}{2} accumulate q*x(i+1)+r*x(i+2)<br>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<br>temp[4] := p
         t = - calculate x(i)<br>x := (1.0 - temp(3)) / temp(4) :
   -- backl control and i/o<br>if
      t = (ib + tq)seq 
               -- receive x(i+1), x(i+2) from i+1irl ? temp(O) 
            ir2 7 temp[1]<br>backl.calc
      t = ((ib + 1) + tg1)<br>par
            -- send x(i), x(1+1) to i-1 process
            oll 1 x 
            ol2 1 temp(O) 
-- State performing a forward substitution on<br>-- vector x produced by back1, by means of the
-- multiplier and control vectors produced by
 -- gaussl.
 -proc gauss2• 
   proc gauss2.calc• 
       seq 
         \frac{1}{16} - check for interchange<br>\frac{16}{16} e = 1.0
                ••q temp(l] :• temp(OJ temp( 0] :• x x :• temp{ 1) 
          -- calculate x
```
#### $\mathbf{I}$ .., .,  $\bar{\mathbf{e}}$  $\mathbf{I}$

<sup>x</sup>*:•* x- (m\* temp(OJ) -- gauss2 control and i/o if t =  $(ig + tb1)$ <br>seq  $\left[- \right]$  receive x[i-1] from i-1 process ill 7 temp{ 0) gauss2.calc  $t = (i(g + 1) + tb1)$ <br>par  $--$  send new  $x[i-1]$  to i-1 and  $- x[i]$  to i+1 oll  $i$  temp(0)<br>corl  $i \times$  $t = ((iq + 2) + tb1)$ -- receive new x(i) from i+1 1 rl 7 x : -- State performing back substitution for the -- system P  $* x = y$ , where y is vector x as<br>-- transformed after gauss2 and x is the same<br>-- vector with the final solution.  $$ proc back2• proc back2.calc• seq par  $\frac{1}{2}$  accumulate q\*x[i+1]+c\*x[i+2]<br>temp[2] := 0.0 + (q \* temp[0]) temp(3) := temp(2) + (r • temp(1)) -- check for zero if  $p = 0.0$ temp(4) :• 0.000001 true<br>temp[4]  $:= p$  $t-$  calculate  $x(i)$ <br> $x := (x - \text{temp}(3))$  / temp(4): -- back2 control and i/o if  $t = (ib + tg2)$ seq<br>par  $\sim$  receive x[1+1], x[1+2] from i+1<br>irl ? temp[0]  $1r2$  ? temp $[1]$ back2.calc  $t = ((ib + 1) + tg2)$ <br>par  $--$  send  $x[i], x[i+1]$  to  $i-1$ oll l x  $ol2$  ! temp[0] : -- The process has four states, gauss1, back1, gauss2,<br>-- 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 1•[0 for 6]

proc back2•  $t = ((n1 - 1) + 1) + tg2)$ <br>par  $\frac{1}{2}$  receive x(1), x(2)<br>irl ? any  $i$ r2 7 any : seq initialisation par par 1·[0 for 2] temp $[i]$  := 0.0 t := 0.0 tgl  $:= n1 + 2$ tbl  $:= (2 + n1) + 3$  $tag2 := (3 * n1) + 5$ tb2 :=  $(4 * n1) + 6$ <br>-- state switching while t < tb2 seq if <sup>t</sup>< tgl gaussl t < tbl backl<br>t < tg2 qauss2 true<br>back2  $t := t + 1$ : -- Boundary process for the right end of the array.<br>  $-$  It supllies process nl-1 with the necessary i/O so<br>  $-$  that calculation continues smoothly.It has the<br>  $-$  same structure as the main process but no<br>  $-$  clculatio proc bound.r(chan min, cin,<br>ill, il2, ol1, ol2,<br>value nl)= var float temp ;<br>var t, tgl, tbl, tg2, tb2 : -- gaussl i/O proc gausslif  $t = ((n1 - 1) + 1)$ par<br> $-$  send a(n+1), b(n+1) oll 1 0.0 ol2 l 0.0  $t = ((n1 - 1) + 2)$ <br>par  $\frac{m}{n-1}$  receive m(n+1), c(n+1), u, v<br>min 7 any<br>cin 7 any<br>i11 7 any cin ? any 112 7 any backl i/o proc backl• if  $t = tgl$ <br>par  $--$  send  $x(n+1)$ ,  $x(n+2)$ oll I 0.0 ol2 1 0.0 :

 $\blacksquare$  $\frac{5}{6}$ ق<br>ا

```
temp[i] := 0.0<br>ig := position
                                          ig := position<br>ib := (n1 - 1) - position
                                          t := 0<br>tq1 := n1 + 2t \bar{b}1 := (\bar{2} * n1) + 3tq2 := (3 * n1) + 5t\bar{b}2 := (4 * n1) + 6-- state switching<br>while t < tb2
                                          seq 
                                             If 
                                                 t < tgl 
                                                    gauss1 
                                                 t < tbl 
                                                 \begin{array}{c} \text{back1} \\ \text{t} \leq \text{t02} \end{array}gauss2
                                                  true 
                                                    back2 
                                              t := t + 1:
                                 - It suplies process of the left end of the array.<br>
- It suplies process 0 with the necessary i/o so<br>
- that calculation continues smoothly.It has the<br>
- same structure as the main process but no<br>
- clculations 
                                 proc bound.l(chan mout, cout,<br>irl, ir2, or1, or2,<br>value nl)=
                                     var float temp[2] :<br>var t, tg1, tb1, tg2, tb2 :<br>-- gaussl 1/o
                                     proc gaussl• 
                                        If 
                                            t = 0<br>par
                                                   -- receive and save a(1), b(1)
                                                   irl ? temp[0]<br>ir2 ? temp[1]
                                             t = 1<br>par= send m[1], c[1], a[1], b[1]
                                                   mout 1 -1.0 
                                                   cout 1 1.0 
                                                   orl I temp(O} 
                                                   or2 ! temp[1] :
                                       backl 1/o 
                                      proc backl-
                                          if 
                                             t = (((n1 - 1) + 1) + tq1)<br>par
                                                    -- receive x(1), x(2)irl ? any<br>ir2 ? any :
                                       \rightarrow qauss2 i/o
                                       proc gauss2• 
                                          \frac{1f}{t} = tbl
                                                 -- send x(0)orl 1 \t 0.0<br>t = (tb1 + 1)
                                                 -- receive new x(O)<br>irl 7 any :
----------------------------------------------------------------------------------------------------------fp~.n-u~m.to.scl."een(v[i)) irl ? any 
                                        back2 1/0
```

```
-- gauss2 i/o 
proc gauss2• 
   If t = ((n1 - 1) + 1) + th1)- receive and save x[nl]
          111 1 temp 
       t = ((n! - 1) + 2) + tb1)-- send back x(nl)
          oll 1 temp :
 = back2 i/o
 proc back2• 
    If 
       t - tg2<br>par
              \frac{1}{2} send x[n+1], x[n+2]
             oll 1 0.0 
             ol2 I 0.0 : 
  -seq 
     initialisation 
    par 
        temp := 0.0<br>t := 0tal := nl + 2tbl := (2 * n1) + 3tq2 := (3 * n1) + 5t_{\text{D2}} := (4 * n1) + 6<br>-- state switching<br>while t < tb2<br>seg
           seq if 
              t < tgl<br>gaussl
               t i tbl
               backl<br>t < tq2
                  gauss2
               true 
                  back2 
            t :• t + 1 
-- The system comprises nl linearly connected
-- processors with two boundary processors .
proc system• 
   par<br>bound.l(m.c(0), c.c(0),<br>r1.c(0), r2.c(0),\frac{11.c(0)}{11.c(0)}, 12.c(0), n1)<br>par i=[0 for n1]
       invit(m.c(i), c.c(i), m.c(i+l), c.c(i+l), il.c(i+l), ll.c(i), 12.c(i), rl.c(i+l), rl.c(i+l), rl.c[i+l], rl.c[i+l], philophilon (m.c[in]), bl(i), m(i), c(i), x[i], i, nl)<br>bound.r(m.c[n]), e.c(i+l), m(i], c[i], x[i], i, nl)
                   \begin{bmatrix} 11.c(n), 12.c(n), \\ r1.c(n), r2.c(n), n1 \end{bmatrix}proc get.f(var float v(],value n,s())• 
    seq str.to.screen("*c *n Input stream ")
        str.to.screen(s) 
       seq i=[0 for n]
           seq<br>fp.num.from.keyboard(v[i])<br>fp.num.to.screen(v[i])
```
 $\mathbf{I}$  $\infty$ õ  $\mathbf{I}$ 

```
str.to.screen(" ");
proc get(var v[], value n, s[])=
   seq<br>str.to.screen("*c *n Input stream ")
      str.to.screen(s)
      \frac{1}{\sec} i=\left(0 \text{ for } n\right)\etaum.from.keyboard(v[i])
             num.to.screen(v[i])
             str.to.screen(""):
 proc get.i(var v, value s( ])=<br>seq (2) is the from
       seq str.to.screen("*c •n Input ") 
       str.to.screen(s) 
       num.from.keyboard(v) 
       num.to.screen(v) 
  -proc getdata \bullet<br>
seq<br>
get.i(nl, "n ")<br>
get.f(bl, nl, "bl ")<br>
get.f(br, nl, "br ")<br>
get.f(br, nl, "br ")
  proc put.f (value float v[], value n, s[])=
      seq<br>str,to.screen("*c *n Output stream " )
         str.to.screen(s) 
         \frac{1}{\sec q} i=\frac{1}{\sec q}\tilde{f}_p.num.to.screen(v(i))
               str.to.screen(* *) :
   proc put (value v( ), n, s( )) =seq<br>str.to.screen("*c *n Output stream ")
          str. to. screen(s)
          seq i={0 for nl
              seq<br>num.to.screen(v[i])<br>str.to.screen(""):
     proc putdata =<br>seq<br>put.f(bl, nl, "bl.")
            put.f(a, nl, " a ")<br>put.f(br, nl, " br *)<br>put.f(m, nl, " m ")<br>put.f(c, nl, " x ") :<br>put.f(x, nl, " x ") :
     \simseq getdata 
         system 
         putdata
```
 $\mathbf{I}$  $\sigma$  $\mathbf{e}$  $\mathbf{r}$ 

```
- A.3.1-- Systolic array for banded mvips, y = y + C * x.<br>-- 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 \cdot no)-1), to = (no+(2 \cdot wo)):
-- Array configuration : w ips cells are required.<br>-- One delay after each cell for x.
proc system (chan cin[], xin, yin, yout,<br>value w, time ) •<br>chan c.c(wo), x.c[(2*wo)+l], y.c[wo+l] :
  par 
    par i•(O for w) 
         ips.l(c.c[i], x.c[i*2], y.c[i],<br>x.c[(i*2)+1], y.c[i+1], time)
          fp.dl(x.c[(i*2)+1], x.c[(1*2)+2],1,time)
          fp.lk(cin(i}, c.c{i}, time) 
     fp.lk(yin, y.c(O), time) 
     fp.lk(xin, x.c(O], 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 souree.c (chan cout[), 
                  value float c{), 
                  value n, w, p, time)=
   par j•[O for w) 
     var float c.temp[to] :<br>var del :<br>seq
        seq i=[0 for time]
          c.temp[i] := 0.0\frac{1}{2}del := (p - 1) + itrue 
            del: (2 * i)seq i•(del for n) 
        \hat{c}.temp(i) := c((j*n)+(i-del))<br>fp.so.x(cout(j), c.temp, time) :
 -- Source for vectors x, y.
 -- Initial delay of p-1 cycles for y.
 proc souree.xy (chan xyout,
```

```
value float xy(),<br>value n, del, time)=<br>var float xy.temp(to) :
   seq 
      seq i=(0 for time)
      xy.\text{temp}(i) := 0.0<br>seq i=[del for n]
        xy.temp[i] := xy[i-de]fp.so.x(xyout, xy.temp, time) 
- Sink for vector y.<br>- Initial delay of w + p - 1 cycles.
proc sink.y (chan yin,
   var float 
y.temp(to) 
   ••q 
                    var float y[ ),
                    var float y[],<br>value n, del, time ) =<br>viemnitol :
      seq i•{O for time) 
        y.temp(i) := 0.0fp.sl.x(yln, y.temp, time) 
        y(i) := y.length[i+del]:
-- Main
chan c.c(wo), x.c, y.c[2] :<br>var float c(no•wo), x(no), yi(no), yo(no)<br>var n, w, p, time :<br>seq<br>get(n, " size of matrix C ")<br>get(w, " bandwidth of matrix C ")<br>get(p, " size of upper semiband ")
   fp.get.n(c, (w*n), " of matrix diagonals, upper diag. first")f_{\mathbf{p}}, qet.n(x, n, " vector x ")
    fp.get.n(yi, n, " vector y ") 
   time := (n + w) + (p - 1)<br>par
       source.c(c.c, c, n, w, p, time)
       source.xy(y.c(0), yi, n, (p-1), time)<br>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 ")
```
# $\infty$  $\sim$

```
- A.3.2
\rightarrow Systolic pipeline for Jacobi, JOR iterative methods for the \rightarrow solution of a linear system A^*x = b. The methods have the form
- x^r = 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(J): 
 external 
proc 
 external proc fp.dl(chan xin, xout, value d, time):<br>external proc fp.lk(chan xin, xout, value time):<br>external proc fp.br(chan xin, xlout, x2out, value time):
 external proc ips.1 (chan cin, xin, yin, xout, yout,
                         fp.get.n(var float v( ), value n,s[ );<br>fp.put.n(value float v( ), value n,s[ ));<br>fp.so.x(chan xout, value float v(), value time):<br>fp.si.x(chan xin, var float v(), value time):<br>fp.si.d(chan xin, value time):
                                   value time): 
def no = 10, wo = ((2*n<sub>O</sub>)-1), ko = 10, to = (nn+(k<sub>O</sub>+(2*n<sub>O</sub>))):
-- Block configuration
- A mvm array performs the mvips operation x'(-y) = C^*x+y;<br>- (w+l) 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[],<br>value stage, w, delay, time I •<br>chan c.c(wo), x.c((2*wo)+l), y.c(wo+1), c.l(3*wo), y.l(3]:<br>par
       par -- mvm. array 
       par i•[O for w) 
          par<br>ips.1(c.c[i], x.c[i*2], y.c[i],
       -- brancing and delays<br>par
              x.c[(i*2)+1], y.c[i+1], time)<br>fp.dl(x.c[(i*2)+1], x.c[(i*2)+2], 1, time)
          par i•(O 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) fp.br(y.l[O], y.l[1], y.c[O], time)
       fp.dl(y.l[1], y.l[2], delay, time) -1/0 links
       par 
          par i=[0 for w]<br>par
                 fp.lk(cin[(stage*w)+i], c.l(i], time)f_{p.1k(c,1(i+(2iw)), cout[((stage+1)*w)+i], time)<br>fp.lk(yin[stage], y.l[0], time)
          fp.lk(y,1|2), yout[stage+1], time)<br>fp.lk(xin(stage), x.c[0], time)
          fp.lk(y.c(w), xout(stage+1), time)<br>fp.si.d(x.c(2*w), time) :
-- Source for matrix C
--- Initial delay of p-1 cycles for the uppermost diagonal;<br>--- additional delay of 1 cycle for the diagonals up to
-- the main; from then on 2 additional delays per diagonal.
proc source.c (chan coutil.
                          value float c(),
```
value n, w, p, time)=<br>par i=[0 for w] var float c.temp(to) :<br>var del :<br>seq seq i=[0 for time] c.temp(il *:•* 0.0 if<br>j<р del :=  $(p - 1) + j$ true del :=  $(2 + 1)$ seq i={del for n]<br>c.temp[i] := c[(j\*n)+(i-del)] fp.so.x(cout(j], c.temp, time) : -- Source for vectors x, y.<br>-- Initial delay of p-1 cycles for y. proc source.xy (chan xyout, value float xy[],<br>value n, del, time)= var float xy.temp(to) : seq seq 1•(0 for time]  $xy.temp{1} := 0.0$  $seq$  i=(del for n)<br>xy.temp[i] := xy[i-del] fp.so.x(xyout, xy.temp, time) :  $--$  Sink for vector x.<br>-- Initial delay of k\*(w + p - 1) cycles. proc sink.x (chan xin,<br>var float x[},<br>value n, del, time) = var float x.temp(to) : seq  $seq$   $i = \{0 \text{ for time}\}\n x. temp[i] := 0.0$ fp.si.x(xin, x,temp, time) seq i•IO for n)  $x[i] := x.$ temp $[i+de]$ : -- Sink for matrix C proc sink.e (chan cin[], value  $k$ ,  $w$ , time) =  $par$  j=[0 for w] fp.si.d(cin((k•w)+jJ. time) : -- Main chan c.c(ko\*wo), x.c(ko), y.c(ko) :<br>var float c(no\*wo), y(no), xi[no], xo(no) :<br>var n, w, p, k, delay, time : seq -- input<br>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  $*)$ <br>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)

 $\mathbf{1}$ 

```
time := n + (k * delay)<br>-- process<br>par
  source.c(c.c, c, n, w, p_* time)
  source.xy(y.c(O), y, n, Cp-1), time) 
  source.xy(\bar{x}.c{O}, \bar{x}i, n, \bar{0}, time)<br>par i={0 for k}
   system(c.c, x.c, y.c, c.c, x.c, y.c, 1, w, delay, time)<br>sink.c(c.c, k, w, time)
fp.si.d(y.c(k), time)<br>sink.x(x.c(k), xo, n, (k * delay), time)<br>== 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 \sim (n*n) system Ax-b with a[i,i] \lt 0 i=1,2, ..,n and -- an overrelaxation factor w, 0 \lt w \lt 2, the
-- preprocessor forms matrix \ddot{C} and vector \ddot{y} with :<br>-- (w * a(i,j)) \ddot{A}(i,i) i <> j
-- c(i, j) -1-- (1 - w) i - j<br>-- y[i] = (w * b[i)) / a[i,i], i= 1, 2, .., n.<br>-- 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[]) :<br>external proc fp.get(var float v, value s[]) :<br>external proc fp.get.n(var float v[], value n, s[]) :<br>external proc fp.put.n(value float v[], value n, s[]) :<br>external proc fp.so.x(chan
external proc fp.sl.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.<br>value time ) -
   var float r(3) :
   seq 
      por 
         par i•(O for 2) 
            r(1) := 0.0\tau(2) : \left(1.0 - y\right)seq t•(O for time) 
         seq 
            por 
                ain 7 r[O]
                cout 1 rt21 
                wout I r(l) 
                eout 1 r(1)If 
                r[0] - 0.0[1] [-1.0]true 
                   r(1) := (w/r(0)):
 Diagonal cell : it receives v/a(i,i) and calculates 
-- -(w * a[i,j]) / a[i,i].\overline{\phantom{a}}proc diag ( chan ain, cout, in, out, 
   var float r(4) :<br>seq
      par i={0 for 4}<br>r(i) := 0.0
       seq t=10 for time)
          seq 
            par
```
 $^{\circ}$ ....

 $\mathbf{L}$ 

 $\mathbf{1}$ 

ain  $7$   $\lceil$ (0)

```
cout l r[ll 
            in ? r(2) 
          _{\text{part}} \frac{1}{\sqrt{3}}r(1) := (- (r(0) * r(2)))r(3) = r(2)-- Vector cell : it receives w/a[i,i] and calculates
- (w * b[i]) / a[i,i].
proc vect ( 
chan bin, yout, 
in, 
                value time ) =var float r(3):
  seq 
     par i•(O for: 3} 
        r:{i) :- o.o 
     seq t-[0 for time]
        seq<br>par
             bin 2 r(0)yout I rill 
           \frac{1}{2}n ? r[2]
              r(1) := (r(0) + r(2)):
-- Array configuration.<br>-- In delays : vector and upper-main diag : p-1,p-2,...0<br>-- 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,<br>value float w,<br>value wa, p, time ) =
   chan m.c[wo+1], a.c(wo], c.c[wo], b.c, y.c :<br>par
      -- array, delays for A
      par i•(O for wa) 
         if 
           i < (p-1)<br>par
                 \{p, d1(\text{ain}(i), \text{ a.c}(i), ((p-1)-i), \text{ time}) \}diag(a.c(i), c.c(i), m.c(i+1), m.c(i), time)
                 f_{p,d}l(c.c(i), cout(i), ((2*1)+1), time)
            i - (p - 1)<br>par
                 fp.dl(ain(i), a.c(i), 0, time)<br>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<br>par
                 fp. d1(ain[i], a.c[i], (2*(i-(p-1))), time)
                 diag(a.c(i), c.c(i), m.c(i), m.c(i+i), time)
                  fp.\tilde{d}l(c.c[i],\ \text{cout}[i],\ ((2*p)-1),\ \text{time})cell, dealys for b 
       par<br>fp.dl(bin, b.c, p, time)
          vect(b.c, y.c, m.c(0], time)<br>fp.dl(y.c, yout, 0, time)
       -- dummy sink for last lower diagonal cell
       fp.si.d(m.c[wa), time) : 
  \sim \sim-- Main
  chan a,c[wo], c.c[wo], h.c, y.c i
  var float a(wo•to), b(tol. c(wo•to), y(to), w 1
```
var n, wa, p, time :<br>seq -- getdata<br>get(n, " size of matrix A ") get{wa, " bandwidth of matrix A")  $get(p, "upper semiband of matrix A")$ <br>time  $:= n + (2 * wa)$  $f_{D, qet, n(a, (wa*n), r, wattix, A upper diagonal first r)}$ fp.get.n(b, n, • vector b "I fp.get(w,  $\dot{m}$  overrelaxation factor ")<br>-- driver<br>par system (a.c, b.c, c.c, y.c, w, wa, p, time)<br>-- source for matrix A<br>par 1-[0 for wa] var float tempa(to) :<br>seq seq j•[O for time) tempa(j) :• 0.0 seq  $j = [0 \text{ for } n]$ <br>tempa[j] := a[(i\*n)+il  $f<sub>D</sub>$ , so.x(a.c $[i]$ , tempa, time) -- sink for matrix C<br>par i=[0 for wa] var float tempe(to) : seq<br>fp.si.x(c.c(i), tempc, time) seq i=[0 for time]  $c$ [(i\*time)+j] := tempc[j]<br>-- source for vector b var float tempb[ to) : seq seq j•[O for time) tempb(j) := 0.0<br>seg i=[0 for n] sempb(j) :• b(j)<br>fp.so.x(b.c, tempb, time)<br>-- sink for vector y<br>fp.si.x(y.c, y, time)<br>-- putdata fp.put.n(c, (wa\*time), " matrix C ") fp.put.n(y, time, " vector y "I

```
-- 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)^{2} = C(1)*x(2) + y(1) and x(2)' = C(2)*x(1) + 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]+q[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[]):<br>external proc fp.get.n(var float v(], value n,s[]):<br>external proc fp.put.n(value float v[], value n,s[]):<br>external proc fp.put.n(value float v[], value time):<br>external proc fp.so.x(chan
external proc ips.l (chan cin, xin, yin, xout, yout, value time): 
def no = 10, wo = ((2*no)-1), ko = 10, to = (nno+(ko*(2*w))) :
-- Block configuration<br>-- A mym array performs the mvips operation x'(-y) = C*x+y;
-- (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+<sub>p-1</sub>) delays for x'.proc system (chan cio[], yio[], xin[], xout[],<br>value stage, w, p, m, time ) =<br>chan c.c(wo], x.c[(2*wo)+1], y.c[wo+l], c.l[wo], y.l :
   pat -- mvm array 
      par i={0 for w}<br>par
            ips.l(c.c{i), x.c{i*2}, y.c{i},<br>x.c{(i*2)+1}, y.c{i+1}, time)<br>fp.dl(x.c{(i*2)+1], x.c{(i*2)+2}, 1, time)
      -1/o links, branching, delays
      par i=(0 for w)<br>par
            fp.br(cio((stage*w)+i), c.l(i), c.c(i), time)<br>fp.dl(c.l(i), cio(((stage+1)*w)+i), m, time)
       fp.br(yio(stage), y.l, y.c(0), time)<br>fp.dl(y.l, yio(stage+1), m, time)<br>fp.lk(xin(stage), x.c(0), time)
       fp.dl(y.c[w], xout(stage+1), (m-((w+p)-1)), time)f_{\text{p.si.d(x.c[2*w]},\text{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[],<br>value float c(),<br>value n, w, p, time)=<br>par j=[0 for w]
       var float c.temp(to) :<br>var del :
```
seq seq i•(O for time)  $c.$ temp $[1]$  := 0.0 If j < p del :=  $(p - 1) + j$ true del :=  $(2 * j)$ <br>seq i=[del for n]  $c.$ temp $[i] := c[(j*n)+(i-de])]$ fp.so.x(cout[j), c.temp, time) : -- Source for vectors x, y.<br>-- Initial delay of p-1 cycles for y. proc source.xy (chan xyout,<br>value float xy(),<br>var float xy.temp(to) :<br>var float xy.temp(to) : seq<br>seq i=[0 for time]  $xy.\text{temp}(i) := 0.0$ <br>seq i={del for n} xy.temp[i] := xy[i-del]<br>fp.so.x(xyout, xy.temp, time) : -- Sink for vector x.<br>-- Initial delay of (k \* m) cycles. proc sink.x (chan xin, var float x.temp[to] : var float x[], value  $n.$  del, time)  $=$ seq seq i•(O for time] x.temp(i] *:•* 0.0 fp.si.x( $xin, x.temp, time$ )<br>seq i=[0 for n]  $x[i] := x.$ temp[i+del] : -- Sink for matrix C proc sink.c (chan cin[],<br>par j•[O for w]<br>par j•[O for w]<br>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),<br>y2(no), xi2(no), xo2(no); var n, wl, pl, w2, p2, k, m, time : seq<br>-- input<br>get(n, " size of matrices Cl, C2 ") get(wl, " bandwidth of matrix Cl ")<br>get(pl, " size of upper semiband ") fp.get.n(cl, (wl\*n), " of matrix diagonals, upper diag. first ") fp.get.n(yl, n,  $*$  of r.h.s vector yl<sup>"</sup>)<br>fp.get.n(xil, n,  $*$  of initial vector xl  $*$ ) get(w2, • bandwidth of matrix C2 ")  $get(p2, " size of upper semiband "1")$ fp.get.n(c2, (w2•n), • of matrix diagonals, upper diaq. first •) fp.get.n(y2, n,  $"$  of r.h.s vector  $y2"$ )

 $\mathbf{I}$  $\sigma$  $\ddot{\phantom{1}}$  $\sigma$  $\blacksquare$ 

fp.get.n(xi2, n, " of initial vector  $x2$  ") get(k, • no of iterations •) if<br>  $(w1 + p1)$  >=  $(w2 + p2)$ <br>
m :=  $((w1 + p1) -1)$ true<br> $m := ((w2 + p2) -1)$ time :• n + (k • m) -- process<br>par par<br>source.c(cl.c, cl, n, wl, pl, r<br>source.c(cl.c, cl, n, wl, pl, time)<br>source.xy(xl.c(0), xil, n, 0, time)<br>source.xy(xl.c(0), xil, n, 0, time) source.xy(x1.c(0),  $\overline{x_1}, \overline{n}, \overline{0}, \overline{t}_{\text{ine}}$ source.c(c2.c, c2, n, w2, p2, source.c(c2.c, c2, n, w2, p2, time)<br>source.xy(y2.c(0), y2, n, (p2 - 1), time) source.xy(x2.c(0), xi2, n, 0, time) par i=[0 for k]<br>par par<br>system(cl.c, yl.c, x2.c, x1.c, i, wl. pl, m, time)<br>2.c, i, w2. p2. m, time) system(cl.c, yl.c, x2.c, x2.c, i, w2, p2, m, time)<br>system(c2.c, y2.c, x1.c, x2.c, i, w2, p2, m, time)<br>.c(cl.c, k, wl, time) sink.c(cl.c, k, wl, time)<br>fp.si.d(yl.c(k), time)  $sink.x(x) . c(k), xo1, n, (k * m), time)$ <br> $sink.c(c2.c, k, w2, time)$ fp.si.d(y2.c[k], time)<br>sink.x(x2.c[k], xo2, n, (k \* m), time) -- output<br>fp.put.n(xol, n, " of solution vector xl ")<br>fp.put.n(xo2, n, " of solution vector x2 ")

 $- - A.3.5$ 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<br> $- x(2)'$   $-$  C(2)\*x(1) + y(2) + orf x(2), where C(1), y(1), i-1,2<br> $--$  are derived from A, b, using the overrelaxation factor orf-1.  $x(2)' = C(2)*x(1) + y(2) + orf x(2)$ , where C[i], y[i], i=1,2<br>  $-$  are derived from A, b, using the overrelaxation factor orf-1.<br>  $-$  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[]):<br>external proc fp.get(var float v, value s[]):<br>external proc fp.get.n(var float v[], value n,s[]):<br>external proc fp.put.n(value float v[], value n,s[]):<br>external proc fp.so.x(chan xout, external proc fp.si.x(chan xin, var float v(), value time):<br>external proc fp.si.d(chan xin, value time): external proc fp.dl(chan xin, xout, value d, time):<br>external proc fp.lk(chan xin, xout, value time):<br>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 =  $(nno+(ko*(2*wo)))$  :  $B = B$  mvm array performs the mvips operation x<sup> $1(-y) = C*2+y$ ;</sup> -- (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 xl'. Before the mvm array an ips cell computes y•y+orf•xl. There are reformatinq delay& for the -- xl stream and synchronization delays for C, x2. Notice that  $--$  there are two input and ouput x streams. proc system (chan ciol ), yio( ), xio( ), orfio{ ), xin( ), xout( ), chan c.c[wo], x.c[(24wo)+1), y.c(wo+1), c.l(24wo), y.l{2],  $x.1$ [3], orf.1[2] : par mvm ar:ray par i=(0 for w)<br>par  $ips.$ l{c.c[i], x.c[i\*2], y.c[i],  $x.c$ [ii],  $x.c$ [i+1], time) fp.dl(x.c[(i\*2)+1],  $\bar{x}.\bar{c}$ (i\*2)+2), 1, time)<br>fp.si.d(x.c[2\*w], time)  $f_p$ .additional ips cell<br>ips.l(orf.l(l), x.l(0), y.l(1), x.l(1), y.c(0), time)<br>fp.si.d(x.l(1), time)  $f_1/0$ , branching, delays for C par  $i=[0 \text{ for } w]$ par<br>fp.br(cio((stage\*w)+i), c.l(i), c.l{w+i}, time)<br>fp.dl(c.l[w+i], c.c(i), 1, time) fp.dl(c.l[i], cio[((stage+1)\*w)+i], (m+1), time)  $-1/0$ , branching, delays for y fp.br(yio(stage), y.l(0], y.l(1], time) fp.dl(y.l(0), yio[stage+1], (m+1), time) -- i/o, branching, delays for orf fp.br(orfio(stage), orf.l(O), orf.l(l], time) fp.dl(orf.l(O), or:fio(stage+l), (m+l), time) -- i/O branching, delays for xl, xl

 $\infty$  $\overline{\phantom{a}}$ 

 $\mathbf{r}$ 

 $\blacksquare$ 

```
fp.dl(xio[stage], x.l[0], (p-1), time)<br>fp.dl(xin(stage), x.c(0], 1, time)<br>fp.dl(y.c(w), x.l[2], (m-((w+p)-1)), time)
      fp.br(x.1(2), xio{stage+l), xout[stage+l], time) : 
-- Source for matrix C<br>-- 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[],<br>value float c[],<br>value n, w, p, time)=<br>par i=[0 for w]
      var float c.temp(to) :<br>var del :<br>seq
         seq i•(O for time) 
            c.temp[i] := 0.0
         if 
            j ( p 
                del 1• (p 
- 11 + j 
            true 
         del := (2 <sup>*</sup>)<br>seq i=[del for n]
         \texttt{c.temp}(i) := \texttt{c}(\{j*n\}*(i-dc1))<br>fp.so.x(cout(j), c.temp, time) :
-- Source for vectors x, y.<br>-- Initial delay of p-1 cycles for y.
proc source.xy (chan xyout,<br>value float xy(),<br>var float xy.temp(to) :<br>var float xy.temp(to) :
   seq 
       seq i = [0 for time]<br>xy. temp[i] := 0.0seq i=(del for n)<br>xy.temp[i] := xy[i-del]
      fp.so.x(xyout, xy.temp, time) :
-- Source for orf.
 proc source.orf (chan orfout,<br>value float orf,<br>value time)=
    var float orf.temp[to] :
    seq 
       seq i=[0 for time)<br>orf.temp(i) := orf
       fp.so.x(orfout, orf.temp, time) : 
 -- Sink for vector x.<br>-- Initial delay of k * (m + 1) cycles.
 proc sink.x (chan xin,<br>var float x(),
    value n, del, time) =<br>var float x.temp[to] :
    seq 
       seq i•(O for time) 
          x.temp(i) :• 0,0 
        fp.si.x(xin, x.temp, time) 
        seq i•[O for n) 
          x[i] := x.temp(i+del) :
```

```
Sink for matrix C 
proc sink.c (chan cin[],<br>value k, w, time) =<br>par j•(0 for w]<br>fp.si.d(cinf(k*w)+i], 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), xl2.c(ko) : var float cl(no*wo), yl[no), xil(no), xol(no), c2[no*wol, y2[no), xi2[no], xo2[no], orf :
var n, wl, pl, w2, p2, k, m, time :<br>seq
   -- input<br>get(n, " size of matrices Cl, C2 ")<br>-- data for pipeline 1
    get(wl, " bandwidth of matrix Cl ")<br>get(pl, " size of upper semiband ")<br>fp.get.n(cl, (wl*n), " of matrix diagonals, upper diag. first ")
    fp.get.n(yl, n, " of r.h.s vector yl")<br>fp.get.n(xil, n, " of initial vector xl ")<br>-- data for pipeline 2
    qct(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 ")<br>fp.get.n(xi2, n, " of initial vector x2 ")
    fp.get(orf, M = 1 - 0 verrelaxation factor M)
    get(k, w no of iterations ") 
    if 
        (w1 + p1) >= (w2 + p2)<br>
m := ((w1 + p1) -1)<br>
true
           m : = ((w2 + p2) -1)time := n + (k * (m + 1))-- process 
         -- sources for pipeline 1<br>source.c(cl.c, cl, n, wl, pl, time)
         source.xy(yl.c[0], yl, n, [p] - 1), time)<br>source.xy(xl.c[0], xil, n, 0, time)<br>source.xy(xl2.c[0], xil, n, 0, time)<br>source.vir(orfl.c(0), orf, time)<br>-- sources for pipeline 2
         source.c{c2.c, c2, n, w2, p2, time) 
         source.xy(y2.c(0), y2, n, (p2 - 1), time)<br>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)<br>-- pipelines
         par 1•(0 for k) 
            par<br>system(cl.c, yl.c, xl.c, orfl.c, x21.c, x12.c,
                            i, wl, pl, 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)<br>fp.si.d(y1.c(k), time)
          sink.x(xl.c[k], xol, n, (k * (m + 1)), time)
          fp.si.d(xl2.e(k], time) 
          fp.si.d(orfl.c(k), time)
```
# $\frac{6}{10}$  $\infty$

```
-- sinks for pipeline 2 
  sink.c(c2.c, k, w2, time) 
  fp.si.d(y2.c(k], time) 
   \frac{1}{\sinh x(x^2, c[k], x^2, n, (k + (m + 1))}, \text{ time})fp.si.d(x2l.c(k), time) 
f_{p.s.i.d(orf2.c(k), time)}<br>== output
-- output fp.put.n(xol, n, 
fp.put.n(xo2, n, 
                            of solution vector xl ") 
of solution vector x2 "I
```
 $\rightarrow$ 

 $\ddot{\phantom{a}}$ 

var float  $x(2)$  :

••q

 $-- A.3.6$ ---- Systolic array for the multiplication of a banded<br>-- matrix C of size (n\*n), bandwidth w = p+q-l, 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<br>-- semiband; max size of address vector.<br>def no = 10, wo = ((2\*no)-1), po = no, ado = (no + wo) : -- Vectors of matrix C diagonals; Vector x; Vector y.<br>var float c[ado^wo], x[no], y[no]:<br>-- Actual size of matrix C; bandwidth w; semiband p.<br>var n, w, p.<br>-- Overall operation time; address vector and it size.<br>time, ad, a[  $-$ - Channels for x's, y's and a's.<br>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(),<br>value time)= var float  $x(2)$ ,  $y(2)$  :<br>var a $(2)$  :<br>seq -- initialisation pa< par i•(O for 2) pa<  $x[1] := 0.0$  $y[i] := 0.0$  $a(1) := 0$ -- main operation seq i={0 for time}<br>seq -- i/0<br>par  $x \in 7$   $x(0)$  $\gamma$ in ?  $\gamma(0)$  $ain 7 3(0)$ xout 1 x( 11 yout 1 y[1]<br>aout ! a[1] -- calculation par  $x(1) := x[0]$ <br> $a(1) := a[0]$  $y(1) := y(0) + (x(0) * c(a(0)))$ : -- Delay Cell : propagates its input with one cycle delay. proc delay (chan xin, xout,<br>value time )-
```
-- initialisation 
     par i=[0 \text{ for } 2]<br>x[i] := 0.0x(IJ :• 0.0 -- main operation 
     seq i=[0 for time]<br>seq
           -- i/o<br>par
             xin ? x(0)<br>xout 1 x(1)-- calculation
           x(l) :• x(OJ : 
Source of the array : vectors x, y and a. No delay for x,a ; a has n+w1valid elements. 
\overline{a}proc source (chan xout, yout, aout,<br>value float x[],
                   value a(), n, ad, time)=
 seq i•(O for time) 
     par 
         \frac{if}{i} >= n
              xout 1 0.0
            true 
               xout l x(i) 
         yout l 0.0 
         If 
            i \rightarrow -idaout 1 0 
            true 
               aout I 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<br>var del ;<br>seq
      \det: -w + (p-1)seq i=[0 for time]<br>par
            xin 7 any
             If 
                i < del 
                  yin ? any 
                true 
                   yin ? y( !-dell 
             ain ? any 
 -- Array configuration : w ips cells are required.<br>-- One delay after each cell for x.Each cell has
 n+W-1 memory locations for a diagonal of matrix C. 
  -proc system =<br>
par<br>
source(x.c[0], y.c[0], a.c[0], x, a, n, ad, time)<br>
par i=[0 for w]
          -- load diagonal to cell<br>var float c.temp[ado] :<br>seq
              seq j=[0 for ad]
                 c, \text{temp}(j) := c((i * ad) * j)
```

```
par ips(x.cti~2), y.c(i), a.c{i), x.c(1+2)+1, y.c(1+1), a.c(1+1),
                  c.temp, time) delay(x, c(1*2)+1), x, c(1*2)+2), time)sink(x, c(2*w), y, c(w), a, c(w), y, w, p, time):
\simproc getdata •<br>seq<br>get(n, " size of matrix C ")<br>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 "I : 
\overline{a}proc putdata =<br>seq<br>fp.put.n(y, n, " vector y ") :<br>--
seq getdata 
   system 
   put data
```

```
\mathbf{I}\mathbf{r}\circ\blacksquare
```

```
-- A.3.7<br>--<br>-- Iterative systolic array for matrix-vector multiplication.
\simIthe array performs k successive mvms, producing (A***)*x, where A is a (m*m) matrix, and x is a vector of m elements.
--- Area = (m+1) IPS cells and (m+n) memory.<br>--- Time = (2*m)*(k+1)+1 cycles.
external proc get ( var v, value s[] ) :<br>external proc fp.put ( value float v, value s[] ) :<br>external proc fp.get.n ( var float v( ], value n, s[] ) :<br>external proc fp.put.n ( value float v( ], value n, s[] ) :<br>external pr
external proc fp.si.x ( chan xin, var float x(J, value time ) : 
external proc fp.so.d ( chan xout, value time ) :
external proc fp.si.d ( chan xin, value time ) : 
def mo•lO, ko•lO, to•(((2•mo)*(ko+l)l+ll : 
-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,<br>value cycle, reset, time ) =
    var float w[2], n[2], s[2], acc :
    seq<br>-- initialise<br>par
          acc := 0.0par i•(O for 21 
                w(i) := 0.0n(il :• 0.0 
                s(i) := 0.0-- main operation
        seq i•(O for time] 
           seq 
              -- i/O 
                 par win ? w{O) nin? n(O) sin ? s{O) 
                 nout ! n[1]<br>sout ! s[1]
              s = - calculation<br>acc s = acc + (w[1] * n[0])\begin{bmatrix} \texttt{par} \\ \texttt{w}[1] \end{bmatrix} := \texttt{w}[0]n(1) := s[0]s[1] := n[0]<br>-- control
              If 
                 ( i \setminus cycle ) = reset<br>seq
                       n(1) := acc<br>acc := 0.0 :-- Multiplexer
   -- The initial vector enters the array through s.
   proc mux ( chan sin, sout, fin, fout, 
      value cycle, reset, time ) =<br>var float s, f[2] :<br>seq
```

```
-- initialise 
    par 
       par i = \{0 \text{ for } 2\}<br>f[i] := 0.0flim 0.0<br>-- main operation
     seq i·lO for time] 
       seq 
           -- i/o<br>par
             sin 7s<br>fin 2f(0)sout 1 f[1]
              fout 1 f{l) 
           f(1) := f(0) + s:
-- Source for initial vector: operates every two steps.<br>-- For the first mpy cycle it sents the initial vector, and
-- then it sends dummy elements.
proc so.n ( chan nout,<br>value float x[],<br>value cycle, time ) •<br>var float temp(2*mo) :<br>seq
      seq j={0 for cycle}
         \frac{\text{if}}{(\text{j}\times2)=0}\text{temp}(j) := x(j/2)true 
      temp[j] := 0.0<br>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[],<br>value float a(),<br>value k, m, cycle, time ) =<br>par i=[0 for m]
      var float temp(2*mo) :
       seq 
         seq j=[0 for cycle]<br>if<br>(j \ 2) = 0
                  (j \ 2) - 0 temp[j) :• a((i*m)+(j/2)) 
               true 
                  temp(jl :• 0.0 
          fp.so.d( wout(i), i) 
          seq j=[0 for k]<br>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( 1, value m, delay, cycle, time ) =
    var float x.temp[2*mo] :
    seq 
        fp.si.d( nin, delay )<br>fp.si.x( nin, x.temp, cycle )
       fp.si.d( nin, delay )
```
 $\Lambda$ 

 $\mathbf{1}$  $\sigma$ ..... .....

```
seq j-10 for cycle) 
        if 
           (j \times 2) = 0x(j\overline{y}2) := x.temp[j] :
\sim \sim-- Array Configuration<br>-- Linear array of m IPS cells; north side has a Mux cell.<br>-- 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( ),<br>value m, k,
                   var float y() ) =chan h.c(mo), u.c(mo+2), d.c(mo+21 
   var time, cycle : 
      sycle := (2 * m)<br>time := (cycle * (k + 1)) + 1<br>par
         \overline{-}- ips cells
         par i = [0 \text{ for } m]<br>var reset :
            seq<br>reset := ((cycle + (i - 1)) \setminus cycle)<br>ips ( h.c(i), d.c(i+1), u.c(i+1), u.c(i+2), d.c(i+2),
                       cycle, reset, time ) 
         mux ( d.c(0), u.c(0), u.c(1), d.c(1), cycle, 0, time)<br>-- sources, sinks
         so.w ( h.c, a, k, m, cycle, time )<br>so.n ( d.c{O}, x, cycle, time )<br>si.n ( u.c{O}, y, m, (time-cycle), cycle, time )
          fp.so.d(u.c(m+1), time)
          fp.si.d( d.c(m+l], time ) : 
 --
 Main 
 var float a(motmo), x(mo}, y(mo) 1 
 var m, k : 
  seq<br>qet ( 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 *)
```
 $\lambda$ 

 $\mathbf{I}$  $273$  $\mathbf{r}$ 

```
-- A.4.1-- Systolic Array for matrix-matrix inner product step
-X = A * B + Cexternal proc get ( var v, value s[] ) :<br>external proc fp.get.n ( var float v[], value n, s[] ) :<br>external proc fp.put.n ( value float v[], value n, s[] ) :
external proc fp.so.x (chan xout, value float x(), value time) :<br>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 ) :<br>external proc fp.1k ( 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) :
\rightarrow-- Array configuration<br>-- 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(1, bin(1, cin(1, xout(1, value w, time ) =
    chan a.c(wo*(wo+1)], b.c(wo*(wo+1)], c.c[(wo+1)*(wo+1)] :
    \begin{bmatrix} \texttt{par} \\ \texttt{--} \texttt{los} \texttt{cells} \end{bmatrix}par i=[0 \text{ for } w]<br>par i=[0 \text{ for } w]\overline{ips.h} \overline{ (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+1)],
                        c.c[(i*(w+1))+(\overline{j}+(w+2))], time)
       - i/o links for A, Bpar_{\text{par}} i=[0 for w]
            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)*i), time)
       - i/o links for C_1 Xpar i•(O for ({2•w)+l)) 
          If 
            i = 0<br>par
                par fp.so.d 
c.c(w), time 
                   fp. si .d 
c.c(w), time 
                <- w par fp.lk 
cin(i-1}, c.c(w-i), time ) 
                   f_{p,1k} ( c.c(w+(i*(w+1))), xout(i-1), time)
             i \left(2^{*w}\right)<br>par
                 par fp.lk 
cin(i-1), c.c((w+l)*(i-w)), time ) 
                    fp.ik { c.c ((((w+1)*(w+1))-1)-(1-w)], xout(i-1), time)
             true 
                 par<br>- fp.so.d ( c.c[(w+1)*w], time
                    f_{p,si,d} ( c.c (w+1)*w], time
  \sim \sim-- Main
  var float a(wo*to), b(wo*to), c(((2*wo)-l)*to), x(((2*wo)-l)*to) :<br>chan a.c(2*wo), b.c(2*wo), c.c(2*((2*wo)-1)), x.c(2*((2*wo)-1)) :<br>var n, w, time :
```
 $-$ 

```
seq<br>-- getdata
  get ( n, • matrix order • ) 
  \begin{bmatrix} \n\text{get} & \n\begin{bmatrix} \n\text{w} & \n\end{bmatrix}^n bandwidth for A, B ^n )<br>time := (n + w)the :- (n + m) 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 ( \tilde{c}, (((2*w)-1)*time), \tilde{c} c sequence, upper diagonal first \tilde{c})
   -- driver 
   par<br>par i=[0 for w]
         var float atemp[to], btemp[to] :
         oeq 
            seq j=[0 for time]<br>par
                  \varepsilonatemp(i) := a((i*time)+j]
                  btemp(j) := b[{i*time}+j]
            par<br>fp.so.x ( a.c[i], atemp, time )
      fp.so.x ( b.c(i), btemp, time )<br>par i=10 for ((2^n w)-1)]
         var float ctemp[to], xtemp[to] :
         ••q seq j•(O for time) ctemp[j) :• c[(i*time)+j) 
               fp.so.x (c.c[i], ctemp, time)
                fp.si.x (x.c[i], xtemp, time )
             seq j•(O for time) 
      x[(i*time)+j} := xtemp[j]<br>system ( a.c, b.c, c.c, x.c, w, time )
```

```
-- putdata<br>fp.put.n ( x, (((2*w)-l)*time), * X sequence, upper diagonal first * )
```
 $\mathbf{r}$ *w* 

 $\blacksquare$ 

```
- A.4.2
-- Systolic Pipeline for s successive squarings for a<br>-- (n*n) banded matrix of bandwidth w - p + q - 1, p - q.<br>-- 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<br>-- final bandwidth = (2**s)w - ((2**s)-1),
f(x) = \text{channel count} + \text{sum } ((2**j)w - ((2**j)-1)), j=0,1,2...external proc get ( var v, value s() ) :<br>external proc fp.get.n ( var float v[], value n, s[] ) :<br>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.<br>-- Two copies of A are produced, A•A and B•A transposed<br>-- pelays for A(B) : q(p)-1, q(p)-2, ••, 1 for the q(p)-1 lower<br>-- (upper) diagonals; here p=q=(w+1)/2.
 -- A hex-connected mmm array produces C (=A**2) = A * B.<br>-- 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{],<br>value s, w, base, time ) =<br>chan al.c(wol, bl.c(wol,
           a.c(wo*(wo+1)], b.c[wo*(wo+1)], c.c[(wo+1)*(wo+1)];
     par branch 
       par 1•{0 foe w] 
        par i=[0 for w]<br>- fp.br ( ain[base+i], al.c[i], bl.c[(w-1)-i], time )<br>-- delays
       var p : 
       seq 
          p := (w + 1) / 2par i=[0 for w]
             if 
                i < p 
                   par 
                       fp.dl 
al.c(i], a.c(i*(w+l)], 0, time 
                       f_{p,dl} (bl.c[i], b.c[i], 0, time)
                true<br>par
                    par<br>fp.dl ( a1.c[i], a.c[i*(w+l)], ((i-p)+l), time
                       fp.dl (bl.c(i), b.c(i), ((i-p)+1), time)
        hex-array 
        par i=[0 \text{ for } w]<br>par j=[0 \text{ 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<br>par i=[0 for w]
           par<br>fp.si.d ( a.c) (i*(w+1))+w, time )
```

```
fp.si.d ( b.c((w*w)+1}, time ) 
    = i/0 links for C = A**2par 1·(0 for ((2*W)+1)) 
      if 
         i = 0<br>par
            par<br>fp.so.d ( c.c[w], time
               fp.si.d ( c.c(w), ti \leftarrow w<br>par<br>fp.so.d (c.c(w-i), time)
                fp.so.d ( c.c(w-1], time )<br>fp.lk ( c.c[w+(i*(w+1))), aout[(base+w)+(i-1)], time )
          i \leq (2 \cdot w)<br>par
             par<br>fp.so.d ( c.c((w+1)*(i-w)), time)
                f_{p.1k} ( c.c(((w+1)*(w+1))-1)-(i-w)],
                         aout[(base+w)+(i-1)], time )
          true 
             par<br>fp.so.d (c.c[(w+1)*w], time
                fp.si.d ( c.c((w+1)*w), time
\sim-- Main
var float a[wi*no], c[wo*no] :<br>chan a.c(co) :<br>var n, wa, wc, s, p2, base, time 1
seq<br>-- getdata<br>-- 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 • ) 
   p2 :- 1
   base :• wa 
  seq i=\{0 \text{ for } s\}p2 := (p2 + 2)base : = base + ((p2 * wa) - (p2 - 1))we :• (p2 • wa) - (p2- 1) 
   base := base - wc<br>time := n + base<br>-- driver<br>par
      -- input of matrix A<br>par i=[0 for wa]
        var float atemp(to) :
        seq 
           seq j•(O for time] 
             \text{atemp}\{\texttt{j}\} := 0.0seq j•(O for n) 
              atemp[j] := a[(i*n)+j]fp.so.x ( a.c(i], atemp, time 
      -- output of matrix C = A**2***<br>par i=[0 for wc]
        var float ctemp[to] :
         seq fp.si,x ( a.c(base+i), ctemp, time ) 
           seq j=[0 \text{ for } n]c[(1*n)+j] := ctemp[base+j]
      -- pipeline of s blocks<br>par i=[0 for s]
         var w, p2, base :
         seq<br>p2 := 1b3se :• wa
```
 $.19$  $"$ 

 $\blacksquare$ 

```
seq j = [0 for i]52 := (p2 * 2)base :• base + ((p2 * wa) - (p2 - 1))<br>w :• (p2 * wa) - (p2 - 1)
       base := base - w
system (a.c, a.c, 1, w, base, time)<br>-- nutdata
```
fp.put.n ( c, (wc\*n), " C sequence, upper diagonal first " )

```
\overline{\phantom{0}}
```

```
- A. 4.3
```
-- Systolic Array Jacobi-Hotelling method.<br>-- Time Expansion.

-- A full mmm reusable array is used, with source-sink drivers

```
... configured for the calculation of Successive Matrix Squares<br>-- after an initial Matrix Multpilication.
```

```
-- Matrix A is the Jacobi matrix obtained by the homogeneous - transformation of the original linear system of equations. - area \leftarrow (m+l)**2, where m is the order of the matrix A;
```

```
-- time = (2 \cdot m)*(s+2)+1, where s, are the number of stages for
```

```
-- the matrix squaring.
```
external proc get ( var v, value s[] ) :<br>external proc fp.get.n ( var float v[], value n, s[] ) : external proc fp.put.n ( value float v[], value n, s[] ) :<br>external proc fp.so.x.t ( chan xout, value float x[], value t[]<br>external proc fp.si.x ( chan xin, var float x[], value time ) : external proc fp.so.d ( chan xout, value time ) :<br>external proc fp.si.d ( chan xin, value time ) :<br>external proc fp.float ( value fix, var float flt ) : external proc ips.r ( chan ein, eout, win, wout, nin, nout, sin, sout, external proc fp.so.x.t ( chan xout, value float x[], value t[], time ) : value cycle, reset, time ) external proc mux.r ( chan sin, sout, fin, fout,

```
value time ) : 
def mo•lO, so•lO, to•((2*mo)*(so+2)), co•(mo+2):
```

```
\overline{a}
```

```
\ddotsc-- 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.
المربين
```

```
proc so.w ( chan wout,<br>
value float a[],<br>
var float temp[to], a.temp[to] :<br>
var fb[to] :<br>
var fb[to] :<br>
seq
       -- input vectors
       par 
           seq 
              seq j = [0 for time]<br>temp[j] := 0.0seq j = [0 for cycle](j \times 2) = 0a, temp[1] := a((i * m) * (j/2))true 
              a.temp[j] := 0.0<br>seq j=[0 for cycle]
                  \text{temp}(i+i) := a.\text{temp}(j)
```

```
seq j•(O for time)
```

```
f_{\text{b}}(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 1. time ) •
```

```
5<sub>1</sub>\overline{u}
```
var float temp[to] :<br>var fb[to] :<br>seq 1- input vectors par seq seq j•(O for time)  $temp[j] := 0.0$ temp((2\*il+i] :• 1.0 seq j•[O for time]  $f_{\text{b}}(j) := 1$ <br>-- source fp.so.x.t( nout, temp, fb, time ) :  $\sim$ -- Sink for results on the west side -- Collects the first column in the last mpy cycle. proc si.w ( chan rin,<br>var float x[],<br>value i, cycle, time I = var float x.temp[to]:<br>var delay :<br>seg  $f_{p,si.x( \text{rin}, \text{ x.temp, time } )$ delay :=  $((time + 1) - cycle)$ x[i] *:•* x.temp[delay) : -- Array Configuration<br>-- Square array of (m\*m) IPS cells; west and north side have an<br>-- 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[],<br>value m, s,<br>var float x[] ) =<br>chan r.c[mo\*co], l.c[mo\*co], u.c[mo\*co], d.c[mo\*co];<br>var time, cycle :  $seq$ <br>cycle  $:=(2 * m)$ time  $:=$  (cycle  $*(s + 2)) + 1$ <br>par -- ips cells par  $i=[0 \text{ for } m]$ <br>par  $j=[0 \text{ for } m]$ var di, dj, reset :<br>seq  $d_i := (i * (m + 2)) + (j + 1)$ dj :=  $(j * (m + 2)) + (i + 1)$ reset := ((cycle + ((1 + j) - 1))  $\setminus$  cycle) ips.r ( l.c(di+l}, r.c[di+l], r.c(di], l.c(di], d.c(dj), u.c(dj!. u.c!dj+l), d.c{dj+l], cycle, reset, time ) -- mux's<br>par i=[0 for m]<br>var wn : seq<br>wn :=  $(1 * (m + 2))$ <br>par par ( r.c(wn), l.c(wn), l.c(wn+1], r.c(wn+1], time )<br>mux.r ( d.c(wn), u.c(wn), u.c(wn+1), d.c(wn+1), time ) -- sources, sinks par i=(0 for m)<br>var wn, es :<br>seq

```
-- Main
                 wn := (i * (m + 2))<br>es := wn + (m + 1)<br>par
                     so.w ( r.c(wn), a, i, m, cycle, time )<br>so.n ( d.c(wn), i, time )<br>si.w ( l.c{wn], x, i, cycle, time )
                     fp.si.d( u.c[wn], time I 
                     fp.so.d( 1.c(es), time
                     fp.so.d( u.c(es), time
                     fp.si.d( r.c(es), time ) 
                     fp.si.d( d.c[es], time ) :
var float a[mo*mo], x[mo] :<br>var m, s :<br>seq<br>qet ( m, " size of matrix A " )<br>fp.get.n ( a, (m * m), " matrix A row-wise " )
```

```
get (s, " iterations")<br>system (a, m, s, x)fp.put.n(x, m, n result ")
```
 $\sim$ 

```
-- A.4.4-- Systolic Array Gauss-Seidel-Hotelling method.<br>-- Time Expansion.
-- A full mmm reusable array is used, with source-sink drivers
-- configured for the calculation of a series of Matrix-Matrix<br>-- Multiplications followed by Successive Matrix Squarings.
-- Matrix A is the Jacobi matrix obtained by the homogeneous<br>-- transformation of the original linear system of equations.<br>-- 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[] ) :<br>external proc fp.get.n ( var float v( ], value n, s[] ) :<br>external proc fp.put.n ( value float v( ], value n, s[] ) ;<br>walue float v( ], value
external 
proc 
external 
proc 
 external proc fp.so.x.t ( chan xout, value float x[ ), value t[ ], time ) i
 external proc fp.si.x (chan xin, var float x(), value time) :<br>external proc fp.si.x (chan xin, var float x(), value time) :<br>external proc fp.si.d (chan xin, value time) :<br>external proc fp.si.d (chan xin, value time) :<br>exte
 external 
proc 
 external 
proc 
 external 
proc 
 external 
proc 
                                     value cycle, reset, time ) 
 external proc mux.r ( chan sin, sout, fin, fout,<br>value time ) :
def mo•lO, so•lO, to•((2•mo)*((mo+so)+l)), co•(mo+2): 
\overline{\phantom{a}}\rightarrow-- Source for row i of input matrices, west side.
-- Operates after an initial delay equal to i and every two steps.<br>-- For j-O,l,..,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,<br>value i, m, cycle, time ) •<br>var float temp(to), a.temp(to) :<br>var loc, pre, post, fb(to) :<br>seq<br>-- initialise<br>par
           loc := ipre:•(2•i) 
        post := (cycle - pre)<br>-- input vectors
        par 
           seq 
              seq j=[0 for time]
                  temp(j) 1• 0.0 
               seq l - \lbrace 0 \text{ for } m \rbraceif 
                     1 - i 
                         seq 
                             seq j=(0 for cycle)<br>if
                                   (1 \times 2) = 0a.temp[jl :• a[(i•m)+(j/2)) 
                                   true 
                             a.temp[j] := 0.0<br>seq j=[0 for cycle]
                             temp(loc+j) := a.temp(j)<br>loc := (loc + cycle)
```

```
true 
                       \log : (\log + \text{pre})temp( loci :• 1.0 
                      loc := (loc + post)seq 
           seq j•(O for time) 
              f_{\text{b}}(1) \rightarrow 1
           seq j = [0 for ((m*cycle)+(i-1))]<br>
fb(j) := 0-- source f 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<br>-- 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,<br>value i. time ) =
   var float temp(to) \mathbf rvar fb[to] :<br>seq
      -- input vectors
     par 
        seq 
           seq i-[0 for time]
              temp[i] := 0.0temp((2 - i) + i] := 1.0
        seq j=(0 \text{ for time})<br>fb[j] := 1
      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. temp\left[\text{to}\right]^{\circ}:<br>var delav :
   seq<br>fp.si.x( rin, x.temp, time )
                 var float x().
                 value i, cycle, time ) =delay := ((time + i) - cycle)<br>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(),<br>value m. s.
  var float x[] ) •<br>
chan r.c(mo*co), l.c(mo*co), u.c(mo*co), d.c(mo*co) ;<br>
var time, cycle :<br>
seq<br>
cycle : = (2 * m)<br>
time := (cycle * ((m + s) + 1)) + 1
     par 
        -- ips cells 
        par i=[0 \text{ for } m]par j•(O for m)
```

```
\mathbf{I}^{\circ}\overline{a}\mathbf{1}
```

```
var di, dj, reset 
                \det := (i * (m + 2)) + (j + 1)
                dj := (j * (m + 2)) + (1 + 1)<br>reset := ((cycle + ((i + j) - 1)) \setminus cycle)ips.r ( l.c{di+1}, r.c(di+1}, r.c(di), l.c(di),
                             d.e(dj), u,c(dj), u.c(dj+l), d,c(dj+l), 
                             cycle, reset, time I 
        -- rnux's 
        par i=[0 \text{ for } m]<br>var wn :
           \frac{seq}{wn} := (1 * (m + 2))<br>
\frac{par}{nw}mux.r ( r.c(wn), l.c(wn), l.c(wn+1), r.c(wn+1), time )<br>mux.r ( d,c(wn), u.c(wn), u.c(wn+1], d.c(wn+1), time )
        sources, sinks 
        par i=(0 for m)<br>var wn. es :
-- Main
           seq<br>wn := (i * (m + 2))
              es := wn + (m + 1)<br>par
                 so.w ( r.c[wn], a, i, m, cycle, time )<br>so.n ( d.c[wn], i, time )
                 si.w ( l.c{wn], x, i, cycle, time ) 
                  fp.si.d( u.c(wn), time ) 
                  f<sub>p.so.d</sub>(1.c(es), time)fp.so.d( u.c(es), time
                  fp.si.d( r.c(es), time ) fp.s..d( d.c[es], time ) :var float a(mo*mo), x[mo] :<br>var m, s :<br>seq<br>qet ( m, * size of matrix A " )<br>fp.get.n ( a, (m * m), " matrix A row-wise " )
  get (s, -i) iterations ^*)<br>system (s, m, s, x)f_p.put.n( x, m, " result ")
```

```
- A.4.5-- Systolic Pipeline for s iterations of the Jacobi-<br>-- Hotteling method for the solution of a linear system.
- A is the Jacobi (n*n) banded matrix of bandwidth - \cdots w = p + q - 1, p = q; y is the corresponding rhs vector,
\rightarrow which is finally transformed to the solution vector.<br>\rightarrow area = SUM ( w[j]**2 + w[j] ), j=0,1,2,..,s-1, and
- w(j) = (2**)**- ((2**j)-1) the bandwidth of stage j+1.
- time \bullet n + SUM ( w(j) + p(j) - 1 ), j \bullet 0,1,2,..,s-1, and<br>-- p(j) \bullet ( w(j) + 1 ) / 2 the semiband of stage j+1.
- final bandwidth for matrix squaring - (2**s)\cdot - ((2**s)-1),
- channel count = SUM ( w[j] + 1 ), \frac{1}{1} = 0, 1, 2, \frac{1}{1}, s.
external proc get (var v, value s[] ) :<br>external proc fp.get.n (var float v[], value n, s[] ) :<br>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.<br>-- Two copies of A are produced, A=A and B=A transposed<br>-- Delays for A{B) : q(p)-1, q(p)-2, .., 1 for the q(p)-1 lower
(upper) diagonals; here p•q•(w+l)/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[],<br>value w, p, time ) =<br>chan a1.c(wo), bl.c(wo],
          a.c(wo*(wo+1)), b.c[wo*(wo+1)], c.c[(wo+1)*(wo+1)] :
   pa< 
       1/0 links, branch delays for A,B 
      par 1=[0 for w]<br>par
            fp.br ( ain(1), a1.c(1), b1.c((w-1)-i), time )
            If 
               i \leftrightarrow ppat 
                     fp.d. ( al.c[i], a.c[i*(w+1)), 0, time
                     f_{p,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], bl.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•IO for w) 
          par j•(O for w) 
             ips.h ( a.c[(i*(w+l))+j), b.c((i*w)+j), e.c((i*(w+l))+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
```
e 18

 $\mathbf{I}$ 

 $\blacksquare$ 

```
par i•(O for ((2•w)+l))
```

```
lf 
           1 - 0par 
                  fp.so.d 
c,c(w), time 
                  fp.si.d 
c.c[w), time 
           1 \leftarrow w<br>par<br>fp.so.d { c.c[w-i], time }
                  fp.so.d ( c.c(w-i), time )<br>fp.dl ( c.c(w+(i*(w+l))], aout[i-1], (p-1), time )
            i < (2**)par 
                   f_{p,so.d} ( c.c ((w+1)*(i-w)], time )
 \rightarrowfp.dl ( c.c((((w+1)*(w+1))-1)-(i-w)), aout(i-1),
                              {p-1), time ) 
            true 
                  fp.so.d [ c.c[(w+1)*w], time )fp.si.d ( c.c(w+1)*w], time ) :
- mvips Block configuration<br>
- A mvm array performs the operation y = A*x + y, (x=y).<br>
- Delay of (p-1) cycles for the uppermost diagonal; additional<br>
- delay of 1 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,<br>value w, p, time ) =<br>chan a.c\{2*wo\}, x.c\{(2*wo)+1\}, y.c(wo+2) :
   par<br>-- linear array
      par i=[0 for w]<br>par
             ips.h ( a.c[1], x.c[1*2], y.c[1*1],<br>a.c[1*W], x.c[(1*2)+1], y.c[1+2], time)
             fp.dl ( x.c((1*2)+1), x.c((1*2)+2), 1, time)
       -- i/o, delays for A
       par i=[0 for w]<br>par
             par var del 
             seq 
                 \begin{array}{cc} \text{if} & \\ \text{i} & < \text{p} \\ \text{del} & \text{if} & (\text{p} - 1) + \text{i} \end{array}true<br>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.lk ( y.c(w+l), yout, time ) 
fp.si.d ( x.c{2*w), time ) : 
 -- Pipeline Block configuration<br>-- 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 
 f(x-1) for (p-1) = ((w+1)/2)-1 cycles for synchronisation.
  proc system ( chan a.c(), y.c(),<br>value s, w, base, time ) =<br>chan al.c(wo), a2.c(wo), a3.c(wo), yl.c, y2.c :
     vac p : 
     {^{seq}} p := (w + 1)/2par
```
-- input links, branch<br>par i=[0 for w] fp.br ( a.c(base+i), al.c(i), a2.c(i), time )<br>fp.lk ( y.c[s], yl.c, time ) for. Intear arrays<br>hex ( al.c, a3.c, w, p, time ) lin ( a2.c, yl.c, y2.c, w, p, time -- output links<br>par i=[0 for ((2\*w)-1)] fp.lk (  $a3.c(1)$ ,  $a.c($  (base+w)+i], time ) fp.lk  $(y2.c, y.c(s+1), time)$ : Main var float a(wi\*no), yi[no], yo[no] :<br>chan a.c[co], y.c(so+l] ;<br>var n, wa, w[so], wc, s, p2, base[so], delay, time : seq<br>-- getdata<br>get ( n, " matrix order " )<br>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  $y$  " )  $get ( s, " squarings " )$ <br>  $-$  setup<br>  $p2 := 1$ <br>  $w[0] := wa$  $base(0) := 0$ delay :=  $(wa + ((wa + 1) / 2)) - 1$  $seq$  i=[1 for  $(s-1)$ ]<br> $seq$  $p2: (p2 * 2)$  $w[i]$  :=  $(p2 + wa) - (p2 - 1)$ base(i) := base(i) + w(i-1)<br>delay := delay + ((w(i) + ((w(i) + 1) / 2)) - 1) time  $:=$   $n +$  delay<br>we  $:=$   $((2 * w[s-1])-1)$ <br>-- driver<br>par  $-$  i/o of matrix A par par  $i=[0 \text{ for } w_a]$ <br>var float atemp(to): seq seq j•(O for time] atemp[j]  $:= 0.0$ <br>seq j=[0 for n]  ${\begin{array}{l} \texttt{item}[j] := \texttt{a}((i*n)+j) \\ \texttt{fp.so.x} \ (\ \texttt{a}.\texttt{c}(i)\ \ \texttt{atemp, time} \ ) \\ \texttt{par} \ \texttt{i} = [0 \ \texttt{for we} ] \end{array}}$ fp.si.d (  $a.c$ ) (base[s-l]+w[s-l])+i], time ) -- i/o of vector y<br>par var float ytemp(to) :<br>seq seq i•IO for time)  $v$ temp $(i)$ :  $= 0.0$ seq i•(O for n)  $y$ temp $\{i\}$  :=  $yi[i]$ fp.so.x ( y.c{O}, ytemp, time )<br>var float ytemp(to] : seq<br>fp.si.x ( y.c(s), ytemp, time ) seq 1•(0 for n) yo(i] :• ytemp(i+delay) -- pipeline of s blocks

## $\mathbf{f}$  $\sigma$  $\bullet$  $\mathbf{I}$

```
par i•(O for sJ 
system ( a.c, y.c, i, w[i], base[i], time )
fp.put.n (yo, n, " solution vector ")
```
 $-- A.4.6$  $\sim$   $\sim$ -- Systolic pipeline for Power Method<br>-- Area Expansion.  $--$  Calcualtion of the dominant eigenavalue e and the corresponding -- eigenvector u of a (n\*n) matrix C.The method has the form :<br>--  $y(i+1) - c*x(i); x(i+1) - y(i+1)/m(y(i+1)),$  $-$  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  $v - p+q-1$  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[]):<br>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):<br>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 =  $(no+(ko*(2*wo)))$ : 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.  $\overline{\phantom{a}}$ proc nor (chan yin, yout, eout,<br>value time) = var first.found :<br>var float y(2), first :<br>seg -- initialisation<br>par first  $:= 1.0$ par  $i-[0$  for  $2)$ y{l) :- 0.0 first.found := false<br>-- main operation seq i•(O for time) seq -- i/O par yin ? y(O) yout  $1 \times 1$ <br>eout  $1 \text{ first}$ -- calculation if (not first.found) and  $(y[0] \leftrightarrow 0.0)$ par<br>first := y[0] first.found :• true  $y(1) := y(0)$  / first : -- Block configuration<br>-- A mvm array performs the operation x'(=y) = C\*x;<br>-- w branching elements for C; (w+p) delays for C to next block.

 $\sigma$  $\sim$  $\bullet$ 

 $\mathbf{I}$ 

÷.

-- A normaliser cell is connected to the output of mvm array.

 $-1$ 

 $\lambda$ 

```
proc system (chan c.io[), x.io(), e.o(], 
               value stage, w, delay, time ) -
  chan c.c(wo), x.c((2^iwo)+1), y.c(wo+2), c.1(3•wo), e.c :<br>par<br>'-- mvm array
    par 1 - [0 \text{ for } w]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[(1+2)+1], x.c[(1+2)+2), 1, time)
     brancinq and delays 
    par 
       par i•[O for w) 
         par 
           fp.br(c.1[i], c.1[i+w], c.c[i], time)f_{p,d}l(c,l(i+w), c,l(i+(2*w)), delay, time)
     normaliser 
     nor(y.c(w), y.c(w+l), e.c, time) 
     -- 1/o links 
     par 
       par i=[0 \text{ for } w]<br>par
           fp.lk(c.io((stage*w)+i), c.l(i), time)
            fp.Ik(c.1[i+(2*v)], c.i0((stage+1)*w)+i), time)fp.lk(x.io[stage], x.c(0), time)fp.lk(y.c(w+ll. x.io{stage+l), time) 
       fp.lk(e.c, e.o(stage), time) 
       fp.so.d(y.c[O), time) 
       fp.si.d(x.c(2*w), time) :
-- Source for matrix C<br>-- Initial delay of p-1 cycles for the uppermost diagonal;<br>-- 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_i, p_i, time) = par j=[0 for w]
     var float c,temp(no) :<br>var del :
     seq 
       if<br>__j < p
          j < p del :• (p 
- 1) + j 
         true<br>del := (2 * j)fp. so. d(cout[j], de])seq 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, 
  seq 
                  value float x(), 
                  value n, time)• 
     fp.so.x(xout, x, n) 
     fp.so.d(xout, (time-n)) 
-- Sink for vector x.<br>-- Initial delay of k * (w + p) cycles.
proc sink.x (chan xin,
```
var float x( l, value  $n$ , del, time) =<br>var float x.temp(no) :<br>seq fp.si.d(xin, del) fp.si.x(xin, x, n)  $\sim$ -- 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  $k \cdot (w + p)$  cycles. proc sink.e (chan ein[],<br>var float e, value stage, k, delay, time) = var float e.temp[1] seq If stage <> (k-1) fp.si.d(ein(stage), time) true seq<br>fp.si.d(ein(stage), delay) fp.si.x(ein(stage), e.temp, 1) fp.si.d(ein{stage), e :• e.temp(OJ : (time-(delay+l))) -- Sink for matrix C proc sink.c (chan cin[],<br>value k, w, time) =<br>par j=[0 for w]  $fp.si.d(cin({(k*v)+j}),$  time) : -- Main chan c.c(ko•wo], x.c(ko+l], e.c(ko] 1 var float c(no•wo), xi(no), xo(no), e :<br>var n, w, p, k, delay, time : seg<br>== input "- input<br>get(n, \* size of matrix C \*)<br>get(p, \* size of upper semiband ")<br>fp.get.n(c, (w\*n), " of matrix diagonals, upper diag. first ")<br>fp.get.n(xi, n, " of initial vector x ")  $get(k, "no of iterations")$ <br>delay :=  $(w + p)$ time :=  $n + (k * delay)$ <br>-- pipeline<br>par source.c(c.c, c, n, w,  $p$ , time) source.x(x.c{O], xi, n, time) par i•(O for k)  $system(c, c, x, c, e, c, i, w, delay, time)$ 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<br>fp.put(e, " dominant eigenvalue ") fp.put.n(xo, n, " of corresponding eigenvector ")

 $2^{\circ}$ .....

 $\blacksquare$ 

 $\mathbf{1}$ 

```
- - A.4.7-- Systolic Pipeline for s 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.
p- area • SUM (((2**j)w - ((2**j)-1))**2), j•0,1,2...s-1
- time = n + \text{sum } ((2^{n+1})w - ((2^{n+1})-1)), i=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, values() ) : 
external 
proe 
fp.get.n ( var float v(], value n, s(] ) : 
external proc fp.put.n (value float v(), value n, s()) :<br>external proc fp.put.n (value float v(), value v() using
external proc fp.so.x ( chan xout, value float x(), value time
external 
proc 
external proc fp.so.d ( chan xout, value time ) :<br>external proc fp.si.x ( chan xin, var float x(), value time )
 external 
proc 
 external proc fp.si.d ( chan xin, value time ) :<br>external proc fp.dl ( chan xin, xout, value d, time ) :<br>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 sea ( chan cin, cout, 
                 value float scale,<br>value time \rangle =
    var float c[2] :<br>seq<br>-- initialise
       par i=[0 \text{ for } 2]<br>c(i) := 0.0
       -- main operation
       seq i={0 for time}<br>seq
             -1/0par 
               cin 7 c[0]<br>cout 1 c[1]- calculation
             c(l) :• (c[O) /scale) : 
 -- Matrix Squaring Block configuration.<br>-- Two copies of A are produced, A=A and B=A transposed<br>-- 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••l) • A * B. 
 C is scaled over a povec of 2. 
  -- Two dummy diagonals for C, one on each side, for uniformity.<br>-- s is the stage number; base is used for the mapping function
  -- of the channels.
  proc system ( chan a.io[],
                      value float scale,<br>value w, base, time ) =
     chan al.c(wo], bl.c(wo],<br>a.c(wo*(wo+l)), b.c(wo*(wo+l)], c.c((wo+l)*(wo+l)];
    par<br>-- branch<br>-- 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) / 2par i•(O for w] 
         if 
           i < p 
              par 
                 fp.dl ( al.c(i), a.c(i*(w+1)), 0, time
                 f_{p,dl} (bl.c[i], b.c[i], 0, time)
            true 
              par 
                 f_{p}.d_{l} ( al.c[i], a.c[i*(w+1)], ((i-p)+1), time
                 fp.dl ( bl. c[i], bl. c[i], ((i-p)+1), time)
     -- hex-array
     par i=[0 for w]
       par j•(O for w] 
         ips.h ( a.c[(1*(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+l))+(j+(w+2))), time) 
     - - 1/0 links for A, Bpar i = [0 \text{ for } w]par 
         fp.si.d ( A.C (i*(w+1))+w], time )<br>fp.si.d ( b.c ((w*w) + i), time )
     -1/0 links, scaling for C - A**2par i=[0 \text{ for } ((2+w)+1)]If 
         i = 0<br>parpar<br>fp.so.d ( c.c[w], time
            fp.si.d ( c.c(w), time<br>
e* w<br>
par
            fp.so.d ( c.c(w-1), time) 
              sca ( c.c(w+ (i*(w+1))), a.i0( (base+w)+(i-1)),scale, time ) 
         1 < (2^{*}v)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+l)•w), time 
               fp.si.d 
c.c((w+l)•w), time 
\sim \simMain 
var float a[wi*no], c[wo*no] :<br>chan a.c[co] :<br>var float scale[so+1] :<br>var n, s, w[so+1], base[so+1], time :<br>seg
  -- getdata
  get ( n, " matrix order " ) 
  get (w(0), * bandwidth for A : w = 2p - 1 * )
  fp.get.n ( a, (w[O]*n), " matrix A upper diagonal first " )
  get ( s, • squarings • ) 
  f_p, qet.n ( scale, (s+1), " scaling factors " )
  base(O) :• 0 
  seq i•(1 for s) 
    seq 
       \overline{v}(i) := ((2 + w(i-1)) - 1)base\{1\} := base\{1-1\} + w\{1-1\}time := n + (base[s] + s)<br>-- driver
```
 $\sigma$  $\overline{v}$  $\mathbf{v}$  $\mathbf{I}$ 

 $\blacksquare$ 

par<br>-- input of matrix A<br>par i=[0 for w[0]]  $var$  float atemp(no) : seq  $seq j=(0 for n)$ <br>atemp[j] := a[(i\*n)+j]  $fp. so. x (a.c[i], atom, n)$ fp.so.d (  $a.c(i)$ , (time-n) )  $--$  output of matrix  $C = A**2**s$  scaled<br>par i=[0 for w[s]] var float ctemp[no] :<br>seq  $fp,si.d$  (  $a.c(base[s]+i),$  (time-n) ) fp.si.x (  $a.c(base(s)+i), ctemp, n$  )<br>seq  $j=[0 \text{ for } n]$  $c((i*n)+j):$  - ctemp $(j)$ pipeline of s blocks  $par$  1-[0 for  $s$ ] system ( a.c, scale[i], w[i], base[i], time )<br>-- putdata fp.put.n ( c,  $(w{[s]^*n})$ ,  $"$  C sequence, upper diagonal first  $"$  )

 $- - A.4.8$  $\sim$ -- Systolic Array for Power Method.<br>-- Time Expansion. Time Expansion. Calculation of the dominant eigenvalue e and corresponding<br>  $\frac{1}{2}$  = eigenvector u of a (m\*m) matrix A.Form of the method :<br>  $\frac{1}{2}$  y(i+l) = A\*x(i); x[i+l) = y[i+l]/m(y[i+l]),<br>  $\frac{1}{2}$  = where m(x)  $w = m(y(k))$ ,  $u/m(u) = x[k]$ , where k is the no of iterations. Area • (m+l) IPS cells and (m\*m) memory.  $--$  Time  $(2*m)*(k+1)+1$  cycles. external proc get ( var v, value s[] ) :<br>external proc fp.put ( value float v, value s[] ) :<br>external proc fp.get.n ( var float v(], value n, s[] ) :<br>external proc fp.put.n ( value float v[], value n, s[] ) : external proc fp.so.x ( chan xout, value float x[], value time I : external proc fp.si.x ( chan xin, var float x[], value time ) :<br>external proc fp.so.d ( chan xout, value time ) : external proc fp.si.d ( chan xin, value time I : def mo=10, ko=10, to= $(((2*mo)*(ko+1))+1)$ :  $-$ -- Inner Product Step Cell : Multiply - Accumulate<br>-- When the last accumulation occurs the result is sent as feedback<br>-- and the accumulator is reset for next cycle of calculations. proc ips ( chan win, nin, nout, sin, sout,<br>value cycle, reset, time ) = var float w(2), n(2), s(2), acc **:**<br>seq -- initialise<br>par  $acc := 0.0$  $par = [0 for 2] par$  $w(i) := 0.0$  $n(i) := 0.0$  $s[i]$  := 0.0 main operation seq i={0 for time}<br>seq  $\frac{1}{\text{par}}$  i/o win  $2$  w(0) nin ? n{O) sin ? s(O) nout ! n{1}<br>sout ! s[1]  $s-$  calculation<br>acc := acc + ( w[1] \* n[0] )<br>par  $w(1) := w(0)$ <br>n(1) := s(0) nll := n[0]<br>| sill := n[0]<br>| if  $( i \setminus cycle ) = reset$ <br>seq  $n(1) := acc$ <br>acc := 0.0 : -- Multiplexer -- For each mpy cycle it normalises the feedback vector.

-- The initial vector enters the array through s.

т.

```
-- Output: the normalised vector and the eigenvalue. 
proc mux ( chan sin, sout, fin, fout, eout,
  value cycle, reset, time ) •<br>var float s, f(2), first :<br>var first.found :<br>seq
      -- initialise
      par 
        par i•(O for 2) 
           f(i) : 0.0s := 0.0first := 1.0first.found := false
      seq i = \{0 \text{ for time}\}\\frac{1}{2} = i/o<br>par
              sin 7 s<br>fin 7 f(0)sout I flll 
              fout 1 f(1)
            eout 1 first<br>-- control
            if 
            if 
              (i \setminus cycle) = reset<br>par
                    first.found :- false
                    first :• 1.0 
               (not first.found) and (f(O) <> 0.0) 
                  par first.found :• true 
                     first : f[0]f(1) := (f(0) + s) / first:
 -- Source for initial vector: operates every two steps.<br>-- For the first mpy cycle it sents the initial vector, and
  -- then it sends dummy elements.
  proc so.n ( chan nout,<br>value float x( I,<br>value cycle, time ) =
     var float temp(2*mo) :<br>seq
        seq j•IO for cycle] 
if 
             (j \times 2) = 0temp(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[],<br>value float a[],<br>value k, m, cycle, time ) =<br>par i=(0 for m)<br>var float temp[2*mo] :<br>seq
```

```
seq j=[0 for cycle]<br>if
            \frac{f f}{(1 \times 2)} = 0temp{jl :• a[(i*m)+(j/2)) 
                true 
                   temp[i] := 0.0fp.so.d( wout[i], i)<br>seq j=[0 for k]
             f_p, so. x( wout(i), temp, cycle )
          fp.so.d( wout[i], (time - ((k*cycle) + 1)) :
-- 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<br>
var float x,temp(2*mo) :<br>
seq<br>
fp.si.d( nin, delay )<br>
fp.si.x( nin, x.temp, cycle )
           seq j•(O for cycle) 
              if (3 \times 2) = 0x[j/2] := x.\text{temp}[j]<br>var float e.temp[l] ;
       seq<br>fp.si.d( ein, delay )
           fp.si.x( ein, e.temp, 1 
           fp.si.d( ein, (cycle-1) e := e.temp(0) :
 -- Array Configuration<br>-- Linear array of m IPS cells; north side has a Mux cell.<br>-- 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[],<br>value m, k,<br>chan h.c(mo], u.c(mo+2], d.c(mo+2], e.c :<br>var time, cycle :<br>seq<br>cycle := (2 * m)<br>time := (cycle * (k + 1)) + 1
        par<br>-- ips cells<br>-- par i=[0 for m]
               var reset :<br>seq<br>reset := ((cycle + (i - 1)) \ cycle)<br>ips ( h.c[i], d.c[i+l], u.c[i+l], u.c[i+2], d.c[i+2],<br>cycle, reset, time )
            - mux ( d.c[0], u.c[0], u.c[1], d.c[1], e.c, cycle, 0, time )
            -- sources, sinks<br>so.w ( h.c, a, k, m, cycle, time )<br>so.n ( d.c(O), xi, cycle, time )<br>si.n ( u.c(O), e.c, xo, e, m, (time-cycle), cycle, time )
            fp.so.d( u.c(m+l), time ) 
            fp.si.d( d.c(m+l], time ) :
```
 $- -$ 

-- Main

var float a[mo\*mo], xi[mo], xo[mo], e :<br>var m, k :

- seq<br>qet ( m, " size of matrix **A "** ) det (m, "size of matrix A " )<br>fp.get.n ( a, (m \* m), " matrix A row-wise " )<br>fp.get.n ( xi, m, " initial vector " )
- 
- get ( k, " number of iterations ")
- system ( a, xi, m, k, xo, e J
- 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<br> $-$  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<sup>s</sup>s respectively.
- $\sim$
- external proc get ( var v, values() ) :
- 
- external proc fp.get.n ( var float v(), value n, s() ) :<br>external proc fp.put.n ( value float v(), value n, s() ) external proc
- external proc 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 ) : <br>external proc fp.so.d ( chan xout, value time ) :
- 
- external proc
- external proc
- 
- external proc external proc fp.si.d ( chan xin, value time ) :<br>external proc fp.float ( value fix, var float flt ) :<br>external proc ips.r ( chan ein, eout, win, wout, nin, nout, sin, sout, value cycle, reset, time ) :

82  $\overline{u}$ 

 $\mathbf{r}$ 

 $\mathbf{r}$ 

def mo=10, so=10, po=10, to= $((2*mo)*(so+(po+2)))+1)$ , co=(mo+2):

 $\sim$ 

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.  $\sim$   $\sim$
- proc mux.s ( chan sin, sout, fin, fout, value cycle, reset, time ) •<br>var float s[2], f(2) :<br>var j, fb :<br>seq -- initialise

par pac i•(O for 2)

par<br> $s[i] := 0.0$ 

f(i) :=  $0.0$  $j := 0$ 

 $fb := 0$ 

```
seq i=(0 for time)<br>seq
```
fin  $7 f(0)$ <br>sout  $1 f(1)$ fout  $| f(1) |$  -- control

if

If

 $fb - 1$ seq

true

seq

- 
- 
- $i$ /o
	-
	-
	-

 $(i \setminus cycle)$  = reset j:•(j+l)

- 
- 
- 
- sin 7 s(O); fb

 $f(1) := ((f(0) + s(0)) \neq scale(j))$ <br> $s(1) := f(1)$ 

- 
- par
- 
- 
- 
- 
- -
	-
- 
- 
- 
- -
- 
- 
- 
- 
- 
- 
- 
- 
- 
- -- main operation
- 
- 
- 
- 
- 
- 
- 
- 
- 
- 

```
f(1) := f(0)s[i] := (f(0) / scale[j]) :
-- Source for row i of input matrices, north 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 ( ehan nout, 
  value i, cycle, time ) •<br>var float temp(to) :<br>var fb(to) :<br>seq
     -- input vectors
 - par
        seq 
          seq j•(O for time) 
             temp[j] := 0.0temp[i+(2*i)] := 1.0seq j=[0 for time]<br>fb[j] := 1<br>-- source
     fp.so.x.t( nout, temp, fb, time):-- Source for a row i of input matrices, west side.<br>-- Operates after an initial delay of i and every two steps
-- For j=0 and s+l,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 I chan wout, 
                value float a(), 
   value i, s, p, m, cycle, time ) =<br>var float a.temp(to), temp(to) :<br>seq<br>seq
      -- input vectors
     pa r
        seq 
           seq j=[0 \text{ for time}]<br>temp[j] := 0.0
           seq j•[O for cycle] 
              if 
                (1 \times 2) = 0a.temp(j] :• a((i*m)+(j/2)) 
                true 
                   a.temp(j) :• 0.0 
           seq j•[O for cycle] 
              temp(i+j) :• a.temp(j] 
           seq 1-(s+1) for p<br>seq j=(0 for cycle)seq 
                \text{temp}(\text{i}+(1*\text{cycle}))+j := a.temp[j]
           seq j=[0 for time]<br>fb[j] := 0<br>seq j=[0 for (i+((s+1)*cycle))]
      seq j•{O for (i+((s+l)*cycle))) fbljl ,_ 1 source fp.so.x.tl 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,<br>var float xl[}, x2[],
                value i, m, cycle, time ) •
```

```
var float x.temp[to] :
  var delay :<br>seq
     f_{\text{p.si.x(}} rin, x.temp, time )
     rp.si.x( \text{cm}, x.temp, \text{cm}e, \frac{2}{x} • cycle))<br>
delay := ((time + i) - (2 * cycle))<br>
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 <br>-- 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[],<br>value m, s, p,
  var float xl[, x2[] ) =<br>chan r.c[mo*co], l.c[mo*co], u.c[mo*co], d.c[mo*co] :<br>var time, cycle :<br>seq<br>cycle := (2 * m)time := (cycle * (s + (p + 2))) + 1<br>par
         -- ips cells
        par i=[0 \text{ for } m]<br>par i=[0 \text{ for } m]var di, dj, reset :
              seq 
                 di := (i + (m + 2)) + (j + 1)
                 dj := (j * (m + 2)) + (i + 1)<br>reset := ((cycle + ((i + j) - 1)) \setminus cycle)i_{DS, T} ( l.c(di+1), r.c(di+1), r.c(di), l.c(di),
                            d.c{dj), u.c(dj), u.c(dj+l), d.c(dj+lJ, 
         cycle, reset, time )<br>
par i={0 for m}<br>
var wn, reset :
            seq<br>wn := (i * (m + 2))
              reset := ((cycle + i) \setminus cycle)<br>par
                 mux.s (r.c(wn), l.c(wn), l.c(wn+1), r.c(wn+1), scale, cycle, reset, time)
                  mus.s (d.c(wn), u.c(wn), u.c(wn+1), d.c(wn+1),
                             scale, cycle, reset, time ) 
         -- sources, sinks
         par i=[0 for m]<br>var wn, es :
            seq<br>wn := (i * (m + 2))<br>es := wn + (m + 1)<br>par
                  so.n ( d.c(wn), i, cycle, time )<br>so.w ( r.c(wn), a, i, s, p, m, cycle, time )
                  si.w (1, 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), xl(mo], x2[mo), scale((so+po)+4) 
 varm,s,p:
```
 $\mathcal{P}^{\mathcal{P}}(\mathcal{P})=\mathcal{Q}^{\mathcal{P}}(\mathcal{Q})$ 

*0\*   $\sim$ *0\* 

seq<br>get ( m, " size of matrix A " )<br>fp.get.n ( a, (m \* m), " matrix A row-wise \* ) get ('s, " matrix squaring steps ")<br>get ( p, " multiplication steps ") fp.get.n ( scale,  $((s+p)+4)$ , " scaling factors " ) system ( a, scale, m, s, p, xl, x2 ) fp.put.n( xl, m, result-1 " I fp.put.n( x2, m, " result-2 " )

 $-$ 

 $-$  A.4.10

-- Systolic Array for Matrix Exponential.<br>-- Time Expansion. -- A full mmm reusable array is used, with source-sink drivers -- configured for the calculation of a Matrix Exponential, which is<br>-- analysed to the calculation of a Matrix Polynomial followed by 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<br>-- feedback-multiplexing operations accordingly. The source drivers -- are assumed to have some memory-calculation capabilities.<br>-- area = m\*\*2, where m is the order of the matrix;<br>-- time = (2\*m)\* (k+s+2)+m-1where k, are the number of stages for -- the matrix polynomial, and the matrix squaring respectively.  $\overline{a}$ external proc get ( var v, value s[] ) : external proc get ( var v, value s[] ) :<br>external proc fp.get.n ( var float v( c), value n, s[] ) :<br>external proc fp.so.x.t ( chan xout, value float x(), value t(),time ) :<br>external proc fp.so.x.t ( chan xout, value float external proc fp.get.n ( var float v(), value n, s() ) :<br>external proc fp.put.n ( value float v(), value n, s() ) : external proc fp.si.x ( chan xin, var float x(), value time ) : external proc fp.so.d ( chan xout, value time ) :<br>external proc fp.si.d ( chan xin, value time ) : external proc fp.so.d ( chan xout, value time ) :<br>external proc fp.si.d ( chan xin, value time ) :<br>external proc fp.float ( value fix, var float flt ) :<br>external proc ips.r ( chan ein, eout, win, wout, nin, nout, sin, sout external proc fp.float ( value fix, var float flt ) : value cycle, reset, time ) 1 external proc mux.r ( chan sin, sout, fin, fout, value time ) : def mo•lO, ko•lO, so•lO, to•((2\*mo)\*((ko+so)+2)), co•(mo+21:  $\sim$   $-$ -- 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 For  $j = 0, 1, \ldots, k+1, k$  mpy cycles it sents dummy elements; fb is on. proc so.w ( chan wout,<br>value i, k, cycle, time ) <del>•</del><br>var float temp(to) :<br>var fact, loc, pre, post, fb(to) :<br>seq -- initialise par fact  $: 1$ seq  $i=[1$  for  $k]$  $fact := (fact * 1)$ loc :• <sup>i</sup> pre :• (2 \* i) post := (cycle - pre)<br>-- input vectors par seq seq j= ${0 \text{ for time}}$ <br>temp{j} := 0.0<br>seq j= ${0 \text{ for (k+1)}}$ var float ffact :<br>seq<br>loc := (loc + pre)<br>fp.float ( fact, ffact ) temp(loc) :=  $(1.0 / ffact)$  $loc := (loc + post)$ If

 $\mathbf{1}$  $\sigma$ N

-1

 $\blacksquare$ 

l < k

fact := (fact /  $(k - j)$ )

```
seq j•(O for time) 
     \begin{bmatrix} 1 \\ -1 \\ -2 \end{bmatrix} fb(j) i = 1fp.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 /<sup>2**</sup>s) into the array. For the kth mpy it sends matrix
-- I; from then on dummy elements; fb is on after the kth mpy.
\sim -proc so.n ( chan nout,<br>value float a[].
               value i, k, s, m, cycle, time ) \simvar float a.temp[to], temp(to), div: 
  var pre, fb[to]:<br>seq
     -- initialise
     par 
        pre := (2 * i)<br>seq
          div := 1.0seq j•(l for s) 
             div - 4i (div + 2.0)
     -- input vectors
     par 
        seq 
          seq j•(O for time) 
             temp{j) :• 0.0 
           seq j={0 for cycle]<br>if
             if<br>
(j \ 2) = 0<br>
a.temp[j] := (a[((j/2)*m)+i} / div)
                true 
                  a.temp[1] := 0.0seq j=[0 for k]<br>seq 1=[0 for cycle]
                femp((j*cycle)+(1+i)) := a.temp[l]
           temp((k*cycle)+(pre+il) :• 1.0 
         seq 
           seq j•[O for time) 
             fb(j) \rightarrow 1seq j•(O for (((k+l)*cycle)+(i-1))) 
      fb(j) := 0<br>-- 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,<br>var float x[ ],
                value 1, m, cycle, time ) • 
    var float x.temp[to] :<br>var delay :
    seq<br>fp.si.x( rin, x.temp, time )
      delay := ( time = (cycle -1)) - ((m-1) -i)seq j={delay for cycle}<br>
if
            ((j - delay) \setminus 2) = 0x(\{i*m\}+\{i\}-\text{delay})/2) := x.temp[j] :
  -- Array Configuration<br>-- Square array of (m*m) IPS cells; west and north side have an<br>-- 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.  $\sim$ proc system (value float a(),<br>value m, k, s, var float x{} ) •<br>chan r.c[mo\*co], l.c[mo\*co], u.c[mo\*co], d.c[mo\*co] :<br>var time, cycle : seq<br>cycle := (2 \* m)<br>time := (cycle \* (k + s + 2)) + (m-1) po< -- ips cells par i•IO for m) par  $j=[0 \text{ for } m]$ var di, dj, reset :<br>seq  $\hat{d}i := (i + (m + 2)) + (j + 1)$ dj :=  $(j * (m + 2)) + (i + 1)$ <br>reset :=  $((cycle + ((i + j) - 1)) \setminus cycle)$ 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+l], d.c(dj+l), cycle, reset, time )  $\begin{bmatrix} -\text{mix}'s \\ \text{par } i = [0 \text{ for m}] \\ \text{var w}n \end{bmatrix}$ seq<br>wn :=  $(i * (m + 2))$ <br>par po< mux.r ( r.c(wn), l.c{wn), l.c{vn+lJ, r.c(wn+l), time mux.r ( d.c[wn), u.c(vn), u.c(wn+l), d.c(vn+l), time -- sources, sinks<br>par i=[0 for m] var wn, es :<br>
seq<br>
wn :=  $(i * (m + 2))$ <br>
es := wn +  $(m + 1)$ par<br>so.w (r.c(wn), i, k, cycle, time) so.n ( d.c $[wn]$ , a, i, k, s, m, cycle, time ) si.w  $( 1.c(wn), x, i, m, cycle, time)$ fp.si.d( u.c[vn), time )  $f_{\text{p}}$ , so.d(  $1$ , c(es), time fp.so.d( u.c(es), time ) fp.si.d( r.c{es], time ) fp,si.d( d.c(es), time )  $\bullet$ -- Main  $$ var float a{mo\*mo], x[to\*mo] :<br>var m, k, s : - m, m, m = 1<br>get ( m, " size of matrix A " )<br>fp.get.n ( a, (m \* m), " matrix A row-wise " ) seq get ( m, size of matrix A • ) fp.get.n ( a, (m \* m), " matrix A row-wise<br>get ( k, " number of Taylor series terms ")<br>get ( s, " scaling-squaring factor ") system ( a, m, k, s, x )<br>fp.put.nl x, (m \* m), " result " )

 $\mathbf{r}$ 

## $\sigma$  $\mathbf{v}$  $\infty$  $\mathbf{I}$

 $-- A.4.11$ -- Systolic Pipeline for calculation of matrix polynomial  $p(A)$  =  $b0*1+ b1*A + b2*A**2 + ... + bs*A**s$ , where A is  $-$  (n\*n) banded matrix of bandwidth w = p + q - 1, p = q, <br>-- and bO, bl, .., 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 \*  $((s-1)*w - (s-2))$ ,<br>-- final bandwidth  $-$  s\*w -  $(s-1)$ ,  $f(x) = -$  channel count for result  $f(x) = SUN$  (j\*w - (j-1)), j=0,1,..,s.  $--$  The area in each stage is equal to the max area required;  $\leftarrow$  only part of it is used and the rest is replaced by delays;  $-$  optimum time  $n + w +$ SUM (j\*w - (j-1)),  $y=1,2,...,s-1$ . external proc get ( var v, value s[ ] ) :<br>external proc fp.get.n ( var float v[ ], value n, s[ ] ) :<br>external proc fp.put.n ( value float v[ ], value n, s[ ] ) :<br>external proc fp.so.x ( chan xout, value float x{ ], value t 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 ) :<br>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)) :<br>-- $\overline{\phantom{a}}$  $--$  Matrix Polynomial Block configuration.<br>-- A hex-connected mmm array produces C = X + A \* B, with X • bs\*l, 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-l, win•2pin-l 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<br>-- outputs are significant.Two dummy diagonals for C, one on<br>-- each side, for uniformity.base is used for the mapping<br>-- function of the channels carrying the re proc system ( chan ain{}, cin(), aout(), cout(), value float x[],<br>value s, wa, win, w, base, time ) = chan al.c[2\*wo], a.c[wo\*(wo+l)], b.c[wo\*(wo+l)], c.c((wo+l)\*(wo+l)] :<br>par -- i/o, branch, delays for A par i={0 for wa}<br>par fp.br {  $ain[(s*wa)+i], al.c[i], al.c[i+wa], time$ } fp.dl ( al.c(i), aout(((s+l)•wa)+i), w, time) -- hex-array par  $i=[0 \text{ for } w]$ par j•(O 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+l))+(j+l)], b.c[(i\*w)+(w+j)],<br>c.c[(i\*(w+l))+(j+(w+2))], time )<br>-- i/0. input delavs for A.B var da, din, pa, pin : seq  $\overline{da}$  := (w - wa) / 2  $pa := (wa + 1) / 2$ din  $t = (w - win) / 2$ 

pin  $:=$  (win + 1) / 2 par i•(O for w) par If If  $(1 > - da)$  and  $(1 < (wa + da))$ If i < (pa + do)  $f_{p}.d1$  (  $a1.c$  [(i+wa)-da],  $a.c$ [i\*(w+1)], 0, time I true fp.dl ( $a1.c((i+wa)-da)$ ,  $a.c(i*(w+1))$ ,  $((i-da)-(pa-1))$ , time) true fp.so.d (  $a.c(1*(w+1))$ , time )  $(i > = dim)$  and  $(i < (win + dim))$ If  $i \lt (pin + din)$  $f<sub>p</sub>.d<sub>l</sub>$  ( cin(base+(i-din)), b.c(i), 0, time ) true true fp.dl  $($  cin(base+(i-din)], b.c[i],  $((i-din)-(pin-1))$ , time) 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 )<br>--  $1/o$  for X.C var wout, dout *:*<br>seq  $\text{wout} := (\text{wa} + \text{win}) - 1$ dout :=  $(1(2 + w) + 1)$  - wout) / 2 par 1•(0 for ((2•w)+l)) par If I < V fp.so.d ( c.c[w-i], time ) If  $\ddot{\phantom{0}}$ fp,so.x c.c{w-i), x, time true  $fp. so.d { c.c(*w+1*)*(i-w)} , time$ i < dout  $f$ p.si.d (  $c.c(w+(1*(w+1)))$ , time i <• w  $fp.lk$  (  $c.c(w+(i*(w+1)))$ ),  $1 \lt ($ wout + dout) fp.lk (  $c.c(( (w+1)*(w+1))-1)-(1-w)$ ), true  $\text{cout}$  ((base+win)+(i-dout)], time ) cout((base+win)+(i-dout)), time fp.si.d (  $c.c$  ((((w+1)\*(w+1))-1)-(i-w)], time ) : Main var float a(wi\*no), c(wo\*to), x(so+l) :<br>chan a.c(wi\*(so+l)), c.c(wo\*so) :<br>var n, s, wa, w, wc, base, 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 • ) get ( s, " stages • ) fp.get.n  $\{ x, (s+1), " polynomial coefficients " )$  $base : = 0$  $,eq$   $|-(0$  for  $s)$ 

 $\sigma$  $\mathbf{v}$ **O**  $\mathbf{r}$ 

```
base := base + ((i * wa) - (i - 1))
w := ((s - 1) * wa) - (s - 2)\frac{1f}{W} \leq wa
 w < wa<br>
w := wa<br>
wC := (s * wa) - (s - 1)delay := (s + w)time := n + delay<br>-- driver
pa< 
   - i/o of matrix Apar i=[0 for wall<br>par
         var float atemp(to) :
          seq 
             seq j•(O for time) 
                \alphatemp[j] := 0.0seq j=[0 \text{ for } n]<br>atemp[j] := a[{i*n}+j]
             fp.so.x ( a.c(i), atemp, time )
          fp.si.d (a, c((s+wa)+1), time)\frac{-1}{\pi} i/o for C-p(A)
       var float xtemp(to) :<br>seq
          \begin{bmatrix} \text{seq} & \text{1} & 0 & \text{for time} \\ \text{stemp[i]} & \text{1} & \text{1} \end{bmatrix}f_{p,s_0,x} (c.c(0), xtemp, time )<br>par i=[0 \text{ for we}]var float ctemp[to] :<br>seq<br>fp.si.x ( c.c[base+i], ctemp, time )<br>seq j=[0 for n]
                cf(i*n)+j := ctemp[delay+j]
     -- pipeline of s blocks<br>par i=[0 for s]
       var float xtemp(to) :<br>var win, base :<br>seq<br>seq j=(0 for time)
             xtemp(j) i = x(s-(i+1))base :• 0 
           seq j•(O for il 
               \frac{1}{2} \frac{1}{2} = \frac{1}{2} 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 
  -- putdata<br>fp.put.n ( c, (wc*n), " C sequence, upper diagonal first " )
```
 $\mathbf{I}$  $^{\circ}_{\circ}$  $\bullet$  $\mathbf{r}$ 

```
-- A.5.1-- Soft-Systolic simulation for an Optical Processor
- performing Matrix-Vector Multiplication ( Ax = y )<br>- 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<br>external proc get ( var v, value s[] ) :
external proc get.n ( var v[], value n, s[] ) :<br>external proc put.n ( value v[], n, s[] ) :<br>-- optical-systolic routines
external proc emit ( chan datain, dataout, beamout[],<br>value start, num, step, time )
external proc modl (chan datain, dataout, beamin [], beamout]],<br>value stert, num, step, time 1:
external proc shda (chan datain, dataout, beamin(),<br>value start, num, ster, 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 inp.shda ( chan xout, value time ) :<br>external proc outp.shda ( chan xin, var x[], value time )
--<br>-- max matrix size, max bandwidth, max time.
def no = 10, wo = 10, to = (no + wo):
-- optical processor configuration.
proc system (vary(), 
   value a(\}, x(\}, w, time )=
   chan a.c{2*wo], x.c{wo+1}, y.c{2}, em.c{wo], md.c{wo} :<br>-- help occam get the types right.<br>var temp :<br>seq
      temp :• x(O) 
      temp := y(0)<br>par
         -- w-pixel emitter for vector x<br>par<br>driver.em ( x.c(0), x, time )<br>par i=[0 for w]
            emit (x, c[i], x.c[i+1], em.c, i, 1,1,time )<br>outp.em (x, c[w], time)--- w-pixel modulator for matrix A<br>par i= [0 for w]
            -- diagonal of matrix A
            vat temp.a(to) 
            seq 
               par j•(O for time) 
                ` temp.a[j] := a[(i*time)+j]<br>par
                  driver.em ( a.c[i], temp.a, time )<br>modl ( a.c[i], a.c[i+w], em.c<sup>2</sup>, md.c,i,J,j,time )
                   outp.em { a.c(i+w), time ) 
         -- 1-pixel shift detector array for result vector y.<br>par
             inp.shda ( y.c(0), time )<br>shda ( y.c(0), y.c(1), md.c, 0, w,4,time )
             outp.shda ( y,c(l), y, time ) : 
 --- main program.<br>--- input and output data for calculation Ax --- y.
```
 $-\bullet$ 

```
var a[wo*to], x[to], y[to],<br>-- actual matrix size; bandwidth; time.
    n, w, time 1
seq 
  \det ( n, " size of matrix A " )
  get ( w, " bandwidth of matrix A • ) 
   time := (n + w) - 1get.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 " )
```

```
-- A.5.2-- Soft-Systolic simulation for an Optical Processor
- performing Matrix-Vector Multiplication (Ax - y)
f(x) = 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.
\overline{\phantom{0}}-- 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, timel:
 external proc modl ( chan datain, dataout, beamin[l,beamout[],
                         value $tart, num, step, time):
 external proc shda ( chan datain, dataout, beamin[],
 value $tart, num, step, time);<br>external proc driver.em ( chan xout, value x( ), time ) :
 external 
proc 
 external proc outp.em ( chan xin, value time ) :<br>external proc inp.shda ( chan xout, value time ) :<br>external proc outp.shda ( chan xin, var x[], value time )
 external 
proc 
-- max matrix size, max bandwidth, max time. 
def no = 10, wo = 10, to = (no + wo) :
-- optical pcocessor configuration. 
proc system ( var y[J, 
  value a[], x[], w, time ) =
  chan a.c[2*wo), x.c[2], y.c[wo+1], em.c[wo], md.c[wo] :<br>-- help occam get the types right.<br>var temp :<br>seq
     temp := x[0]temp := y(0)par 
       -- 1-pixel emitter for vector x<br>par
          driver.em (x, c[0], x, \text{ time } )emit (x.c(0), x.c(1), em.c, 0, w,1,time)<br>outp.em (x.c(1), time)-- w-pixel modulator for matrix A
       par i=[0 for w]
          -- diagonal of matrix A<br>var temp.a[to] :
          seq 
            par j•(O for time) 
              temp.a[j] := a[(i*time)+i]
            par 
              driver.em ( a.c[i], temp.a, time )
               modl ( a.c(i), a.c(i+w), em.c , md.c, i,1,itime
              outp.em ( a.c[i+w], time )
       w-pixel shift detector array for result vector y. 
       par 
          inp.shda ( y.c(O), time 
         par i•{O for wJ 
          shda ( y.c(i), y.c(i+1), md.c, i, 1,1,time ) outp.shda ( y.c[w], y, time ) :
main program.
```
 $- -$ 

```
-\frac{1}{2} input and output data for calculation Ax = y.
var a{wo*to}, x[to], y[to],<br>-- 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 • I 
  get.n ( x, time, • vectoc x data seq • ) 
  system ( y, a, x, w, time ) 
  put.n ( y, time, " vector y data seq " )
```

```
-- A.5.3-- 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. 
\rightarrow 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[],<br>value start, num, step, time ) :
 external proc emit \ chan datain, data out, beamin(), beamout | ,<br>external proc modl ( chan datain, data out, beamin(), beamout | ,
 external proc shda (chan datain, dataout, beamin(),
external proc outp.cm ( chan xin, value time ) :
 external proc outp.cm ( chan xin, value time ) :<br>external proc inp.shda ( chan xout, value time ) :<br>external proc inp.shda ( chan xout, value time) :
 external proc 
outp.shda ( chan xin, var x(), value time ) 
                            value start, num, step, time ) : 
value start; num, step, time ) :<br>external proc driver.em ( chan xout, value x(), 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) :
    em.co • help occam get the types right.<br>var temp, -- bandwidth of resulting matrix C
         WC 1
    ••q temp :- a{O) 
      temp :• b(O) 
      temp :• c(O) 
      wc := ((2 * w) - 1)<br>par
         -- w-pixel emitter for matrix A<br>par i=[0 for w]
            -- diagonal of matrix A<br>var temp.a[to] :
            seq 
               par j•(O for time) 
                 temp.a[j] := a[(i * time) + i]par<br>driver.em ( a.c(i), temp.a, time )<br>emit ( a.c(i), a.c(i+w), em.c, (i*w), w, l, time )
                 outp.em ( a.c(i+w), time ) 
          -- w*w-pixel modulator for matrix B<br>par i=[0 for w]
            - diagonal of matrix B var temp.b(to) :
            oeq 
               par j•(O for time)
```

```
temp.b[j] := b[{(i * time) * j}]driver.em ( b.c(i), temp,b, time )
              par i=[0 for w]
                 \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} ( \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}, b.c((i*w)+(j+w)], em.c, md.c,
                          ((1+i)(+j), 1, 1, \text{ time})outp.em ( b.c((w*w)+i), time )
       -- (2*w-l)-pixel shift detector array for result matrix C<br>par i={0 for wc)
         -- diagonal of matrix C<br>var temp.c(to) :
         seq 
            par 
               inp.shda (c.c(i), time)if i \in wi < w<br>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-1), (w-1), time )
            outp.shda ( c.c(i+wc), temp.c, time )<br>par j={0 for time}
               par j•{O for time] c((i*time)+j) :• temp.e(j) : 
-- main : p-1 cycles delay for the input of A so that B reaches
            the appropriate position, 
-\overline{\phantom{a}}-- 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 I n, " size of problem " ) 
get I w, " bandwidth of input matrices ") 
get ( p, " upper semibbnd " ) 
  time := (n + p)get.n ( a, (w•time), " matrix A data seq " 
  qet.n ( b, (w*time), " matrix B data seq " )
  system ( c. a, b, w, time ) 
  put.n i c, i((2 \sqrt{w})-1)*time), "matrix C data seq " )
```
 $\blacksquare$ 

 $-- A.5.4$ -- Soft-Systolic simulation for an Optical Processor -- performing Matrix-Matrix Multiplication of two banded<br>-- (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(  $)$  i -- optical-systolic routines. external proc emit ( chan datain, dataout, beamout[ ]. external proc modl chan datain, dataout, beamin(), beamout(), external proc modi (chan datain, bum, step, time) :<br>external proc shda (chan datain, dataout, beamin(), external proc show value start, num, step, time ) :<br>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 inp.shda ( chan xout, value time ) :<br>external proc outp.shda ( chan xin, var x[], value time ) value start, num, step, time ) : -- max matrix size, max bandwidth, max time. def no = 10, wo = 10, to =  $(no + wo)$ : -- optical processor configuration. proc system ( var c $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$ , b(), w, time )= -- data communication. chan a.c $[2*wo]$ , b.c $[2*wo]$ , c.c $(\text{vo+1})*(\text{wo+1})$ , em.c $[\text{wo*wo}]$ , md.c $(\text{wo*wo})$ : em. help occam get the types right.<br>var temp :<br>seq  $temp := a[0]$  $temp := b[0]$ temp :<del>•</del> a[O]<br>temp := b[O]<br>temp := c[O] par -- w-pixel emitter for matrix A<br>par i={0 for w} -- diagonal of matrix A<br>var temp.a[to] : seq par j•(O for time) temp.a[j) :• a((i•time)+jl par<br>driver.em I a.cii], temp.a, time) emit (  $a.c(i)$ ,  $a.c(i+w)$ , em.c,  $(w*ii)$ ,  $w$ , 1, time ) outp.em ( a.c(i+w), time ) w-pixel modulator for matrix B par i=[0 for w]<br>-- diagonal of matrix B<br>var temp.b[to] : seq par j•(O 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<br>par
           par i•(O for w) 
              par j•IO for wl 
                 shda ( c.c)(((i+1)*(w+1))+j], c.c)((i*(w+1))+(j+1)), md.c.
                           ((i * w) * j), 1, 1, 1 time )
           par i=[0 \text{ for } ((2*u)+1)]\frac{1f}{1} \left(-\frac{1}{2}\right)inp, shda ( c.c[i*(w+1)], time )true 
           inp.shda ( c.c(i+(w*w)), time )<br>par i=[0 \text{ for } ((2*w)+1))- diagonal of matrix C var temp.c(to) :
              seq 
                 \begin{array}{c} \text{if} \\ \text{if} \\ \end{array}outp.shda ( c.c[i], temp.c, time )
                    true 
                      outp.shda ( c.c[i+((i-w)•w)), temp.c, time) 
                 par j•(O for time] 
                   c([i*time)+1] := temp.c(i) :
-- main : two dummy diagonals for C; the uppermost<br>-- and the lowermost.
\rightarrow input and output data for calculation AB = C.<br>var a(wo*to), b(wo*to), c(((2*wo)+l)*to),<br>-- actual matrix size; bandwidths; time.
n, w, time : seq 
   get ( n, • size of problem • ) 
   get ( w, • bandwidth of input matrices ") 
  time := ((n + w) - 1)<br>get.n ( a, (w*time), " matrix A data seq " )
  get.n ( b, (w*time), * matrix B data seq * )<br>system ( c, a, b, w, time )
  put.n (c, ((2 * w) + 1) *time). * matrix C data seq * )
```
 $-$ 

L.

```
- 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 \bullet w; detector size \bullet we.
-- i/o routines.<br>external proc get ( var v, value s[] ) :
external proc get.n ( var v(), value n, s[] ) :<br>external proc put.n ( value v(), n, s[] ) :
external proc emit (chan datain, dataout, beamout(],<br>external proc emit (chan datain, dataout, beamout(],<br>value start. num, step, time ) ;
 external proc emer value start, num, step, time ) :<br>external proc modl (chan datain, dataout, beamouti), beamout[],
 external proc moul (comme start, num, step, time) :<br>external proc shda (chan datain, dataout, beamin(),<br>value start, num, step, time) :
 ext{ernal 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(),<br>value a(l, b(l, v, time )=
    -- data communication. 
    chan a.c(2•wo), b.c(2•vo), c.c((vo+l)•(wo+l)), 
     -- help occam get the types right.<br>var temp,
     -- wordlwngth of result number c<br>we :<br>seq
        temp :• a[O] 
        temp := b[0]<br>temp := c[0]wc := ((2 * w) - 1)<br>par
           \begin{array}{l}\n\ldots \text{ w-pixel emitter for number a}\n\text{par i}=[0 \text{ for w}]\n\ldots \text{ one bit of number a}\n\text{var temp.a[t0]:}\n\text{seq}\n\text{par } j=[0 \text{ for time}]\n\end{array}temp.a[j] := a[(i*time)+j]par<br>- driver.em ( a.c[i], temp.a, time )<br>- 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<br>par i=[0 for w]
               -- one bit of number b var temp.b[to] :
               ••q par j•(O for time) temp.b(j) 1• b((i•time)+j) 
                   par<br>driver.em ( b.c[i], temp.b, time )<br>modl ( b.c[i], b.c[i+w], em.c, md.c, i, w, w, time )
```

```
outp.em ( b.c[i+w], time )<br>-- (2*w-1)-pixel shift detector array for result number c<br>par i=[0 for wc]
         -- one bit of number c<br>var temp.c(to) :
         seq 
             par<br>| inp.shda ( c.c(i), time )
               if \atop i < wshda ( c.c(1), c.c(1+wc), md.c, i, (1+1), (w-1), time )
-- Main
                  true 
                     ue<br>shda { c.c[i], c.c[i+wc], md.c, ((i*w)-((w-1)*(w-1))),
                              (wc-1), (w-1), time )
             outp.shda ( c.c[i+wc], temp.c, time )<br>par j=[0 for time]
               cf(i+time)+j] := temp.c[j] :
input and output data for calculation ab • c var a(wo•to), b(wo•to), e(((2•wo)-1)•to), 
-- actual wordlength 
     w \rightarrowseq<br>qet ( w, " wordlength of input numbers ")
   get.n ( a, (w•to), " number a bit seq " ) 
get.n ( b, (w•to), "number b bit seq" ) 
   system ( c, a, b, w, to ) 
  system (c, a, b, w, w), "number c bit seq " ) \infty \omega
```
 $\alpha$ 

```
\mathbf{u}
```
 $\mathbf{r}$ 

```
-- A.6.1\sim-- Soft-Systolic Simulation Library
external proc str.to.screen(value s[J): 
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<br>-- get put formation of the screen
-- get put
library proc fp.get(var float v,value s(J)• 
    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{))=<br>seg
    seq str.to.sereen("*c •n Input stream ") 
    str.to.screen(s) 
    seq i=[0 for n]<br>seq
         f_{p,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[])=<br>seg
     str.to.screen("*c *n Input stream ")
     str.to.screen(s) 
     seq i = {0 for n}<br>seqnum.from.kevboard(v[i])num.to.screen(v(i]) 
         str.to.screen("") :
library proc fp.put {value float v,value s())• 
   ••• 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.scceen(s) 
     seq i=[0 for n]<br>seq
          f_p.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) z 
library proc put.n (value v[],n,s[])=<br>seq
     str.to.screen("*c *n Output stream ")
     str,to.screen(s) 
     seq i=[0 for n]<br>seq
          num.to.screen(v[i])
          str.to.screen(" ") :
    Source/Sink of fixed-/floating-point vectors of dummy/significant<br>so si enterprise to d x
                                                                  d • 
elements with/without tag 
               t 
-library proc so.x (chan xout, value x{ }, time} = seq t={0 for time}
     xout I x(t) : 
library proc fp.so.x (chan xout, value float x), value time) = seq t={0 for time}
     xout t x(t) : 
library proc so.d (chan xout, value time) = seg 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]
     x \in \mathbb{R} x \in \mathbb{R}:
library proc fp.si.x (chan xin, var float x(), value time) • 
   seq t=[0 for time)<br>xin 7 x[t] :
library proc si.d (chan xin, value time) - seq t=[0 \text{ for time}]xin ? any : 
 library proc fp.si.d (chan xin, value time) = \sec t = 0 for time]
      xin ? any : 
 library proc \mathfrak{sox.t} (chan xout, value x( ), tag( ), time) = seq t={0 for time}
      xout i x[t]; tag[t]:
 library proc fp.so.x.t (chan xout, value float x{], value tag{], time) = seq t=[0 for time]
      xout 1 x[t]; taq[t]:
library proc so.d.t (chan xout, value time) \bullet seq t=[0 for time]
      xout 1 0; 0 : 
 library proc fp.so.d.t (chan xout, value time) = seq t=[0 \text{ for time}]xout 1 O.o, 0 :
```
 $\ddot{\phantom{0}}$ 

 $\sigma$ t a l 0\

 $\blacksquare$ 

```
library proc sing t (chan xin, var x{}, tag(), value time) =<br>seq t={0 for time] \frac{1}{x}<br>xin 7 x(t); tag(t) :
library proc fp.si.x.t (chan xin, var float x[], var tag[], value time) =<br>seq t=[0 for time]<br>xin 7 x(t); tag{t]:
library proc si.d.t (chan xin, value time) -<br>var r(2) :<br>seq t={0 for time}
     xln 7 r(O]; r(l) : 
library proc fp.si.d.t (chan xin, value time) =<br>var float r :<br>var s :<br>seq t={0 for time]
     xin \overline{z} r; s :
-- Fixed-/Floating-point conversions 
library proc fp.float (value fix, var float flt) = \frac{seq}{111} := fix :
library proc fp.fix (value float flt, var fix) =<br>seq
      fix := fit :-- Delay/ for fixed-/floating-point variables with/without tag -- dl fp t
def do = 20 :<br>library proc dl (chan xin, xout,
   value d, time ) =<br>var x[do] :<br>seq
      par i=[0 for (d+1)]x[1] := 0seq i=[0 for time]
        seq<br>seq
              xin ? x[0]xout 1 x(d) 
            seq j•(O for d) 
              x(\bar{d}-j) := x((d-j)-1):
 library proc fp.dl (chan xin, xout,<br>value d, time )-
   var float x(do) :<br>seq
      par i•(O for (d+l)J 
        x[i] := 0.0seq i=[0 for time]
         seq<br>seq
              xin 7 x[0]xout 1 x(d] 
            seq j•(O for d) 
              x(d-j) := x((d-j)-1):
  library proc 
dl.t (chan xin, 
xout, 
    var x[do], t[do] :
    seq 
       par i•IO 
for (d+l)) 
                             value d, time )=
```
 $\mathcal{L}^{\text{max}}$ 

```
par<br>x(1) := 0tiil:\bullet 0
     seq 1•(0 for time) 
        seq<br>seq
             xin 7 x[0]; t[0]xout 1 \times (d); t(d)seq j=[0 \text{ for } d]par 
                x[d-1] := x[(d-1)-1]t(d-j) := t((d-j)-1) \bar{i}library proc fp.dl.t (chan xin, xout, value d, time )• var float x(do) : var t(do) 1 
   seq 
     _{par} i=[0 for (d+1)]par 
          x[i] := 0.0t(i) ; = 0seq i=[0 \text{ for time}]seq 
             xin 7 x(0); t[0]xout l x(d]; t(d) 
           seq j•(O for d) 
             par 
                x(d-1) := x((d-1)-1)t(d-1) := t((d-1)-1):
-- Branching for fixed-/floating-point variables with/without tag<br>-- br t
     br fp t
library proc br (chan xin, xoutl, xout2, 
   value time) •<br>var x :<br>seq<br>x := 0
      seq i=[0 for time]<br>seq
           \begin{array}{ccc}\n\text{x in 7 x} \\
\text{par} \\
\text{x out1 1 x} \\
\text{x out2 1 x 1}\n\end{array}library proc fp.br (chan xin, xoutl, xout2,<br>var float x :<br>seq x := 0.0seq i•(O for time) 
         seq<br>xin ? x
           par<br>xoutl i x<br>xout2 i x :
 library proc br.t (chan xin, xoutl, xout2, 
    value time) - value time) -seq<br>par
         x := 0t :• 0
```
 $\mathbf{r}$  $\mathbf{v}$ .....

```
seq i=[0 for time}<br>seq
           x \nvert n \nvert 7 \nvert x; t<br>par
              xoutl 1 x; t
              xout2 1 x; t :
library proc fp.br.t (chan xin, xoutl, xout2,<br>value time) =
   var float x : varities
   seq 
        x := 0.0t :• 0 
      seq i=[0 for time]<br>seq
            xin 7 x; t<br>par
               x \cdot txout2 + x<sub>i</sub> t :
-- Links for fixed-/floating-point variables with/without tag \begin{array}{c} -r & t \\ r & t \end{array}library proc 1k (chan xin, xout, value time) = var x :<br>seq
      x := 0seq t=[0 for time]<br>seq
            x \in \mathbb{R} x \in \mathbb{R} x \in \mathbb{R}library proc fp.lk (chan xin, xout, value time) = var float x :<br>seq x = 0.0seq t•[O for time] 
         seq<br>xin ? x
            xout 1 x :library proc lk.t (chan xin, xout, value time) = var x, tag :
   seq<br>par
         x := 0.0tag :• 0 
       seq t={0 for time}<br>seq
             seq xin 7 x; tag 
             xout 1 x; tag : 
 library proc fp.lk.t (chan xin, xout, value time) -\text{var} float x :<br>var tag :
    seq<br>par
          x := 0.0tag :• 0 
       seq t=[0 for time]<br>seq
             xin 7 x; tag
             xout 1 x; tag 
 \overline{\phantom{a}}
```

```
-- Inner Product Step cell for a Linear array 
library proc ips.l (chan cin, xin, yin, xout, yout, value time)=<br>var float c, x(2), y(2);<br>var float c, x(2), y(2);
   seq 
     -- initialisation 
        c := 0.0par i=(0 for 2)
          par 
             x[1] := 0.0v(1) := 0.0-- main operation
     seq i•(O for time) 
        seq 
           -1/0par<br>cin 7 c
             cin 7 c xin ? x(OJ 
             yin 7 y(OJ 
             \begin{bmatrix} x_{\text{out}} & f \\ x_{\text{out}} & y_{\text{out}} \end{bmatrix}-- calculation<br>par
             \begin{array}{l} x[1] := x[0] \\ y[1] := y[0] + (x[0] + c) \end{array}Inner Product Step Cell for the Hex-connected array 
library proc ips.h ( chan ain, bin, cin, aout, bout, cout,.
   var float a[2], b[2], c[2] :<br>seq
     -- initialise 
     par i=[0 \text{ for } 2]a[i] := 0.0b[i] := 0.0<br>c[i] := 0.0-- main operation
     seq i=[0 for time]<br>seq
           -1/0par 
             ain 7 a(OJ bin 1 biOI 
             cin ? c[O]<br>aout ! a[l]
             bout 1 b(1) cout 1 c(1)
          -\frac{1}{2} calculation<br>par
             c(1) := c(0) + (a(0) * b(0))<br>a(1) := a(0)b(1) := b(0):
-- Building Blocks for the reusable matrix multiplication array
\qquad \qquad --- 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,<br>value cycle, reset, time ) =
  var float e[2], w[2], n[2], s[2], acc :
```
 $\mathbf{\hat{g}}$ 0)  $\mathbf{r}$ 

••q initialise seq  $\begin{bmatrix} par & i-1 & 0 & for & 2 \end{bmatrix}$  $e[i]$  := 0.0  $w[i] := 0.0$  $n(1) := 0.0$  $s[i] := 0.0$ <br>acc := 0.0 -- main operation  $seq$  i= $[0$  for time)<br>seq  $\frac{2}{\pi}$  i/0<br>par ein ? e[0] win ? w{ 0] nin 7 n[0]  $sin 7 s(0)$ eout 1 e(l) wout  $1 \leq 1$ nout  $\{n\}$  sout  $\{s\}$ -- calculation<br>par  $\bar{p}_{acc} := \bar{p}_{acc} + {w[0] * n[0]}$ e[1]  $1 - w(0)$  $w(1) := e(0)$  $n(1)$  :-  $s(0)$  $s(1) := n(0)$ -- control If  $(i \setminus cycle)$  • reset seq par  $w(1) := acc$ <br> $n\{1\} := acc$  $acc := 0.0 :$  $\overline{a}$ -- Multiplexer -- It adds the source input to the array feedback if fb -- is true; otherwise it accepts the source input. --<br>library proc mux.r ( chan sin, sout, fin, fout, value time ) = var float s $\{2\}$ , f $\{2\}$  :<br>var fb :<br>seq -- initialise par  $par$  i= $[0 for 2]$ <br>par  $s(i) := 0.0$ f(i)  $: 0.0$ fb  $:= 0$ -- main operation seq 1-(0 for time)<br>seq <sup>-</sup>- i/o<br>par sin  $? s(0);$  fb fin  $2 f[0]$ sout 1 s(l) fout  $1$   $f(1)$ -- control par

If  $fb - 1$  $f[1] := (f[0] + s[0])$ true f[1]  $:=$  s[0]  $s[1]$   $i = f[0]$   $i$ 

```
- - A.6.2--
-- Soft-systolic simulation library of optical
-- systolic algorithms.
\sim \sim-- 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<br>-- systolically to the next pixel.
 --<br>library proc emit ( chan datain, dataout, beamout[],
                          value start, num, step, time ) -
   var data(2) : 
     - initialisation
     par i=[0 \text{ for } 2]<br>data[i] := 0
      -- main operation
     seq i•(O for time) 
        seq<br>par
             datain ? data[O]
              dataout 1 data(l) 
           par<br>par j=[0 for num]
                beamout(start+(j*step)) i data[1]
              data(1) := data(0) :
-- Definition of a pixel of a modulator : a beam from an<br>-- 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<br>def bo = 10 :
    var data[2], beam[bo) :
    seq<br>-- initialisation
      par 
         par i•{O for 2) 
            data[i] := 0par i=[0 \text{ for num}]<br>beam[i] i=0-- main operation
       seq i•(O for time) 
         seq 
            par 
              datain 7 data[0]
              dataout ! data[1]<br>par j=[0 for num]
            beamin(start+(j*step)) ? beam(j)<br>par j=(0 for num)
              \text{beam}[j] := \text{data[0]} * \text{beam}[j])
            par<br>par j=[0 for num]
                  beamout(start+(j*step)) l beam[j]
               data(1) := data(0) :Definition of a pixel of a (shift) detector array : the 
 -- incident light beams are transformed to electic charges<br>-- 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(),<br>value start, num, step, time )=<br>def bo = 10: var data(2), sum, beam(bo) : seq<br>-- initialisation par par i•(O for 21  $data[i] := 0$ par  $i = [0 \text{ for num}]$ <br>beam[i] := 0  $\mathbf{r}$ -- main operation seq i•(O for time] seq par<br>- datain 7 data[0] dataout ! data[1]<br>par j=[0 for num] par j =  $\binom{6}{1}$ <br>sum := 0<br>par j={0 for num} par i= sum + beam(j)<br>data(1) := (data(0) + sum) : -- Driver for emitter, modulator. --<br>library proc driver.em ( chan xout, value x(), time )• seq i•(O for time] xout l x(i) -- Dummy output for emitter, modulator. library proc outp.em ( chan xin, value time )• seq i=[0 for time]  $x$ in 7 any : -- Input for (shift) detector array. library proc inp.shda chan xout, value time )• seq i•(O for time)  $x$ out  $1 \t0$  : -- Output of a (shift) detector array. library proc outp.shda chan xin, var x(], var x[].<br>value time )= seq i= $[0 \text{ for time}]$ <br>xin  $7 \times [1]$  :

 $\blacksquare$  $\sigma$  $\blacktriangle$  $\bullet$  $\mathbf{I}$ 

-- The result moves systolically to the next pixel.