A Multiple-SIMD Architecture for Image and Tracking Analysis

Darren James Kerbyson

A Thesis Submitted to the University of Warwick for the degree of

Doctor of Philosophy

Department of Computer Science

University of Warwick

December 1992

Summary

The computational requirements for real-time image based applications are such as to warrant the use of a parallel architecture. Commonly used parallel architectures conform to the classifications of Single Instruction Multiple Data (SIMD), or Multiple Instruction Multiple Data (MIMD). Each class of architecture has its advantages and dis-advantages. For example, SIMD architectures can be used on data-parallel problems, such as the processing of an image. Whereas MIMD architectures are more flexible and better suited to general purpose computing. Both types of processing are typically required for the analysis of the contents of an image.

This thesis describes a novel massively parallel heterogeneous architecture, implemented as the Warwick Pyramid Machine. Both SIMD and MIMD processor types are combined within this architecture. Furthermore, the SIMD array is partitioned, into smaller SIMD sub-arrays, forming a Multiple-SIMD array. Thus, local data parallel, global data parallel, and control parallel processing are supported.

After describing the present options available in the design of massively parallel machines and the nature of the image analysis problem, the architecture of the Warwick Pyramid Machine is described in some detail. The performance of this architecture is then analysed, both in terms of peak available computational power and in terms of representative applications in image analysis and numerical computation. Two tracking applications are also analysed to show the performance of this architecture. In addition, they illustrate the possible partitioning of applications between the SIMD and MIMD processor arrays.

Load-balancing techniques are then described which have the potential to increase the utilisation of the Warwick Pyramid Machine at run-time. These include mapping techniques for image regions across the Multiple-SIMD arrays, and for the compression of sparse data. It is envisaged that these techniques may be found useful in other parallel systems.

Contents

Summary i

1 Introduction 1

1.1 Background 1

1.1.1 Image analysis 2

1.1.2 Parallel architectures 3

1.2 Outline of this Thesis 4

2 Parallel Architectures 7

2.1 Introduction 7

2.2 Classification of Parallel Architectures 9

2.2.1 Memory Structure 10

2.2.1.1 Shared memory 10

2.2.1.2 Distributed shared memory 11

2.2.1.3 Distributed memory 12

2.2.2 Inter-connection networks 13

2.2.2.1 Bus based systems 13

2.2.2.2 Static interconnection networks 14

2.2.2.3 Dynamic interconnection networks 17

2.2.3 The control of parallel processors 19

2.2.4 SIMD vs. MIMD arrays 21

2.2.4.1 Active Resources 22

2.2.4.2 Technology 23

2.2.4.3 Programming 24

2.2.4.4 Local autonomy 25

2.3 Survey of Existing Parallel Machines 27

2.3.1 SIMD architectures 27

2.3.1.1 Bit-serial SIMD processor arrays 28

2.3.1.2 Multi-bit SIMD processor arrays 33

2.3.2 MIMD Architectures 36

2.3.2.1 Shared Memory machines 36

2.3.2.2 Distributed shared memory 38

2.3.2.3 Distributed memory........ 39
2.4 Summary........ 41

3 **Architectural Options for Image Analysis**........ 43
3.1 Introduction........ 43
3.2 Image Analysis Processing........ 44
3.2.1 Iconic processing........ 46
3.2.2 Intermediate processing........ 47
3.2.3 Symbolic processing........ 48
3.3 Computational Requirements for Image Analysis........ 48
3.4 Image Analysis Architectures........ 51
3.4.1 Dedicated hardware........ 52
3.4.2 Homogeneous pyramids........ 53
3.4.3 Reconfigurable........ 55
3.4.4 Heterogeneous architectures........ 56
3.5 Summary........ 60

4 **The Warwick Pyramid Machine**........ 62
4.1 Introduction........ 62
4.2 Overview of the WPM........ 63
4.3 Implementation of the WPM........ 65
4.3.1 The SIMD array........ 66
4.3.1.1 Functionality of the DAP PE........ 67
4.3.1.2 The SIMD associative Count........ 70
4.3.2 The MIMD array........ 72
4.3.3 The Cluster Controller........ 74
4.3.3.1 The Cluster Bus........ 75
4.3.3.2 The interface between the DAP SIMD array and the controller........ 76
4.3.3.3 The scalar ALU........ 77
4.3.3.4 Instruction sequencing........ 79
4.3.3.5 The shared memory between the controller and the Transputer........ 82
4.3.4 Connecting Clusters together........ 83
4.3.4.1 Communication at the SIMD level........ 84
4.3.4.2 The synchronisation of Clusters........ 86
4.3.5 The prototype system........ 90
4.3.6 M-SIMD Operation of the WPM........ 92
4.4 Programming the WPM........ 93
4.4.1 The Cluster assembler - CLASS........ 94

4.4.1.1 The use of CLASS for Cluster operations 96
4.4.2 Remote procedure calls between C and CLASS 97
4.4.3 Pyramid C++ 99
4.5 Performance of the WPM 101
4.6 Summary 103

5 Mapping and Processing Data on the WPM M-SIMD Array 104
5.1 Introduction 104
5.2 The Advantage of M-SIMD over SIMD 105
5.2.1 Local Autonomy 105
5.2.2 Associative Response Operations 108
5.2.3 Count Response 111
5.2.4 Data communication 112
5.3 Mapping Data onto the M-SIMD Array 113
5.3.1 Sheet mapping 114
5.3.2 Crinkled mapping 114
5.3.3 Comparison of Data mappings 115
5.4 Image Operations on the M-SIMD Array 117
5.4.1 SIMD filtering operations 117
5.4.2 The use of the WPM associative count 121
5.4.2.1 Histogram generation 122
5.4.2.2 Rank order filters 124
5.4.2.3 Mean and Variance calculation 126
5.4.2.4 Image moment calculations 127
5.5 Matrix Operations on the WPM 128
5.5.1 Mapping considerations 129
5.5.2 Matrix Algorithms 131
5.5.2.1 Matrix addition and subtraction 131
5.5.2.2 Matrix multiplication 131
5.5.2.3 Matrix inversion 133
5.5.2.4 Matrix transpose 134
5.5.3 Performance of matrix operations 135
5.5.3.1 Addition and subtraction performance 137
5.5.3.2 Multiplication performance 138
5.5.3.3 Inversion performance 141
5.5.3.4 Transpose performance 141
5.6 Summary 143

6 The Performance of Tracking Operations on the WPM 145
6.1 Introduction 145

6.1.1 Estimation of unknown quantities 146
6.2 Tracking Algorithms 148
6.2.1 Tracking models 150
6.2.2 Data Association 154
6.2.2.1 The Nearest Neighbour Standard Filter 156
6.2.2.2 The optimal Bayesian approach 156
6.2.2.3 The Probabilistic Data Association Filter 157
6.3 Tracking Application 1 - Object Tracking 159
6.3.1 Object tracking models 160
6.3.1.1 Position tracking model 160
6.3.1.2 Size tracking model 161
6.3.2 Object detection and segmentation processing 163
6.3.2.1 Object Detection 164
6.3.2.2 Object Segmentation 167
6.4 Tracking Application 2 - Generic Target Tracking 170
6.4.1 Image processing operations 171
6.4.1.1 Image thresholding 171
6.4.1.2 Monotonicity Operator 171
6.4.1.3 Corner detection 172
6.5 Performance on the WPM 174
6.5.1 The image processing 174
6.5.2 Tracking operations 175
6.5.2.1 The data association 176
6.5.2.2 Processing of the Kalman filters 178
6.6 Summary 182

7 Load-Balancing on the WPM 183
7.1 Introduction 183
7.2 Minimising Clusters Used for Processing 185
7.2.1 Calculation of Image Shifts 186
7.2.2 Voting on an image shift 188
7.3 The Bin Packing of Clusters 191
7.3.1 Three-dimensional bin packing 192
7.3.2 Two-dimensional bin packing 194
7.3.3 Comparison between the FDDH and the FPDS algorithms 197
7.3.4 Implementation of the load-balancing algorithms 200
7.4 Load-Balancing of Sparse Data 201
7.4.1 The Monotonic Lagrangian Grid 202

7.4.2 The Partitioned Monotonic Lagrangian Grid 207
7.4.3 Comparison between the MLG and the PMLG 209
7.4.4 Construction of an MLG on an SIMD array 212
7.5 Summary 213

8 Conclusions 215

Bibliography 220

Appendix A - Cluster Bus Ports 234

Appendix B - The Cluster Assembler 240

Appendix C - The Kalman Filter 247

List of Figures

2.1 A shared memory system 11
2.2 A distributed shared memory system 11
2.3 A distributed memory system 12
2.4 Some static interconnection network topologies 15
2.5 Three dynamic interconnection topologies 18
2.6 The organisation of an SISD architecture 19
2.7 The organisation of an SIMD architecture 20
2.8 The organisation of an MISD architecture 20
2.9 The organisation of an MIMD architecture 21
2.10 The organisation of an M-SIMD architecture 26
2.11 The configuration of a two-dimensional SIMD array 27
2.12 The MPP processing element 29
2.13 Example reconfiguration of the RPA PEs 31
2.14 Two CM2 Nodes with associated memory and floating point unit 32
2.15 Four clusters from the DASH prototype 38

3.1 Example image analysis processing flow 44
3.2 The levels of processing within image analysis 46
3.3 The configuration of the SYDAMA-2 system 53
3.4 The structure of a multiple-layer homogeneous pyramid architecture 54
3.5 The Orthogonal Multi-Processor with an Enhanced mesh 57
3.6 The Image Understanding Architecture 59

4.1 The Warwick Pyramid Machine 63
4.2 A Cluster, the modular component of the WPM 64
4.3 A 16x16 DAP SIMD array, showing the detail of 4 PEs 66
4.4 The DAP processing element 67
4.5 A 64 input count 71
4.6 A 256 input count from four 64 input counts 72
4.7 The internal components of a T800 and a T9000 Transputer 73
4.8 The main components of the Cluster controller 74

4.9 Operation of the Cluster Bus 76
4.10 Interface between the SIMD array and the controller 77
4.11 The controllers scalar ALU (the AMD29116) 78
4.12 The Cluster controllers intruction word 80
4.13 The controllers sequencer showing Transputer Load path 81
4.14 The shared memory between the Transputer and the controller 83
4.15 The interconnection of four Clusters 84
4.16 Interconnecting Cluster SIMD arrays 85
4.17 The hand-shaking mechanism between Clusters 87
4.18 Example synchronisation between six Clusters in a shift operation 89
4.19 The prototype Cluster boards 91
4.20 The Remote Procedure Call frame used within the Cluster 97

5.1 Number of regions that can be processed in parallel on an ideal M-SIMD architecture and on the WPM M-SIMD array 106
5.2 Efficiency of the WPM and SIMD array for varying region sizes 107
5.3 Comparison between the ideal M-SIMD architecture, the WPM M-SIMD array and an SIMD array for associative response operations 109
5.4 Relative performance of the WPM M-SIMD array, and an SIMD array, with the ideal M-SIMD architecture for associative operations 110
5.5 The time taken to count the number of bits set in a single bit plane on both an SIMD array and on the WPM 112
5.6 The time taken to shift data across the M-SIMD array within the WPM, and an SIMD array 113
5.7 Sheet and Crinkled data mappings on an array processor 115
5.8 Convolution paths for a 5x5 mask on an array processor 118
5.9 3x3 convolution masks for smoothing and high-pass filtering 120
5.10 The horizontal Sobel gradient operator 120
5.11 The Sobel operator decomposed into neighbour additions 121
5.12 The Faddeev formulation of matrix inversion 133
5.13 Example iteration of the Faddeev inversion algorithm on a 3x3 matrix 134
5.14 Transposing a matrix on an array processor 134
5.15 An example of sixteen matrices mapped onto a processor array 136
5.16 Time taken to perform a 16x16 matrix addition for a varying number of matrices 138
5.17 16x16 matrix multiplication on a 128x128 DAP SIMD array 138
5.18 16x16 matrix multiplication on an 8x8 Cluster WPM 139
5.19 Comparison of the minimum processing time for matrix multiplication between the DAP and WPM arrays 139

5.20 Comparison between the WPM and the DAP SIMD array for matrix multiplication over a range of matrix sizes 140
5.21 Comparison of the minimum time for matrix inversion between the DAP and WPM arrays 141
5.22 Comparison of matrix transpose on the DAP and WPM arrays 142

6.1 The combination of two Gaussian measurements 147
6.2 Example of validation gating in two dimensions 156
6.3 Two images from a sequence approaching a car 159
6.4 Processing flow showing image processing, size and position tracking 160
6.5 The Imager showing relationship between the size of an object and its depth from a camera 161
6.6 Example detection of circles using the circular Hough transform 165
6.7 Edge points which contribute to a single parameter space location within the Spoke filter 165
6.8 Example of the Spoke filter 166
6.9 Example of the Convex hull algorithm 169
6.10 Example output from the object segmentation 169
6.11 Processing flow in a generic multiple-target tracking environment 170
6.12 Example image from a meteor sequence 171
6.13 The monotonic operator of Kories 172
6.14 Example image showing the categories of the Harris Corner detector 173
6.15 The parameters passed between the validation gating, PDAF or NNSF data association and the tracking Kalman filters 176
6.16 Validation gates mapped onto four Clusters within the WPM 177
6.17 Time taken to perform a number of Kalman filter operations, n=4, m=2, on a WPM Cluster and a T800 Transputer 180

7.1 The area of an aeroplane on an M-SIMD machine 185
7.2 An object across five Clusters showing the possible shift directions that can take place to free each Cluster of it 187
7.3 Example of a North-East shift on part of a region mapped across four Clusters 189
7.4 Example distribution of sub-images across the WPM, from four images 191
7.5 Example of a set of jobs mapped onto an array processor where each job requires (p_i, q_i) processors and t_i time 192
7.6 Example of the malleability of the mapping of sub-images 193
7.7 Example of shelf packing of eight blocks 195

7.8 The utilisation of the WPM using the FFDH Shelf and FPDS load-balancing algorithms 197
7.9 The number of sub-images assigned across the WPM using the FFDH Shelf and FPDS algorithms 198
7.10 The communications required using the FFDH shelf and FPDS algorithms 199
7.11 The time cost for the communications required within the FFDH Shelf and FPDS algorithms 199
7.12 Example of the MLG mapping on a set of 16 data elements 203
7.13 Further example of the MLG mapping resulting in poor locality between data elements 204
7.14 Originl vs. MLG separation on randomly distributed data 205
7.15 Original vs. PMLG separation on randomly distributed data 208
7.16 Mapping nine compressed data sets into a single Cluster........ 209
7.17 Example of unevenly distributed data........ 210

List of Tables

2.1 Trends in VLSI speed and density 8
2.2 Comparison of some network topologies 16
4.1 The DAP instruction groups 69
4.2 Operations performed for inter-Cluster communications 88
4.3 Overhead for synchronised communications 90
4.4 Delays across the Cluster bus 95
4.5 Comparison of the use of macros and RPC calls 99
4.6 Cluster peak computational, memory and communication performances 102
5.1 Characteristics of Sheet and Crinkled mappings 116
5.2 Time taken for convolution operations on the WPM 119
5.3 Comparison of the time taken for the computation of the Sobel filter 121
5.4 Comparison of histogram generation algorithms on the WPM 123
5.5 Comparison of Rank order filter calculations on the WPM 125
5.6 Comparison of the mapping of matrices across a processor array 130
5.7 Time for the computation and communication in matrix operations 135
5.8 Multiplicative factors applied to the computation, broadcast and shifting times of the various data mappings 137
6.1 Image processing operations on the M-SIMD level of the WPM 175
6.2 Matrix/vector operations required on a single iteration of a Kalman filter with n states and m measurements 179
6.3 Timings for the image processing and tracking operations in the two tracking applications 181
7.1 Voting of the SIMD patches shown in Figure 7.3 190
7.2 Time taken for the load-balancing algorithms 201
7.3 Processor operations to find all data points within a distance of 16 pixels from each other 212
7.4 Time taken for the construction of a PMLG block on the SIMD array within a WPM Cluster 213
7.5 Time taken, and advantages of, the load-balancing techniques on the WPM 213

Acknowledgements

There are many people to whom grateful thanks are due. Firstly, I would like to thank Professor Graham Nudd, my supervisor, for enabling this work to take place and for his guidance throughout it. I would also like to thank other members of the VLSI group, both past and present, for their help and support. Special thanks must be given to Tim Atherton for his constant enthusiasm to many aspects of this work, and to Roger Packwood for his support in the early stages.

I would also like to thank Tim Atherton and Kay Garbett for their valued work in proof-reading this thesis.

Finally, I would like to give very special thanks to Kay Garbett, for her support, love, understanding, and patience during the writing of this thesis.

Declaration

This thesis is presented in accordance with the regulations for the degree of Doctor of Philosophy. It has been composed by myself and has not been submitted in any previous application for any degree. The work described in this thesis has been undertaken by myself except where otherwise stated.

Overviews of the Warwick Pyramid Machine have been published in [Nudd88, Vaudin89, Nudd89, Atherton90, Nudd91, Nudd92a]. The use of the Warwick Pyramid Machine for tracking operations has been published in [Kerbyson92, Nudd92b, Nudd92c].

Chapter 1

Introduction

1.1 Background

There has been rapid progress in increasing the performance of conventional uni-processor computer systems over recent years [Hennessy90]. Two factors account for much of the performance gained, namely the increase in clock speeds and the increase in the density of components achievable in VLSI. Further, studies of the efficient utilisation of components, within a processor, lead to the development of the Reduced Instruction Set Computer (RISC) approach [Patterson80]. The resulting effect on processors has been an increase in instructional throughput and overall performance.

There are application areas, which have sufficient demands upon computational resources to make uni-processor systems in-feasible options. Parallel processing has become associated with such areas. These application areas include meteorology, oceanography, medical imagery, fuel combustion, and computer vision [Rattner91]. Current research programmes aim to achieve the processing performances required for these applications, such as that initiated by the U.S. government - The Grand Challenges in High Performance Computing [Grand93]. The aim of this is to achieve Tera-Flop (10^{12} floating point operations) performances for a range of applications.

The requirements posed by each of these applications is ever increasing. For instance, the size of the data sets involved in the computation is often a function of the sampling resolution. The sampling resolution can relate to the smallest particle allowed in a many-body physical simulation, or to the spatial resolution of an image, or to the temporal resolution between images. Factors such as these enables processing with increasingly fine resolution, as and when the computational power permits.

The factors which increase the amount of computation in image analysis include the spatial resolution of the image sensors, the number of frames processed per second, and the algorithms used. The resolution of the images are increasing, for example, this can be seen with the current shift towards High Definition Television (HDTV) standards [Harris92]. Another factor is the increase in frame rate, e.g. Kodak has recently demonstrated a camera which can capture video at a rate of 1000 frames per second [Kodak90].

Image analysis has received much attention in terms of the algorithms required for specific operations [Suetens92]. However, little work has been done to produce processors able to achieve real-time performance. This thesis is concerned with the computational requirements of image analysis. A major contribution of the work presented here is the analysis of a programmable parallel architecture which has been optimised for use on a range of algorithms found in image analysis.

1.1.1 Image analysis

Image analysis is the process of taking in an image and extracting some high level measurements from it. In a computer vision environment, one would like to use such a system to mimic the functionality of the human visual system. This requires the processing of low-level pixel based information and the processing of high level information after it is extracted from the input images. This processing flow is commonly referred to as a bottom-up approach.

Several levels of processing are commonly associated with image analysis - low, intermediate and high [Weems91, Duff88, Leviadli88]. The low level involves the processing of image pixels either globally, where the operations are done for each pixel, or locally where regions of interest are processed in an object dependent way. The high level processing deals with symbolic representations of the images, possibly re-enforcing hypotheses through the use of known 'world' information. The intermediate level processing forms a transitionary level between the low and high processing levels. An image tracking application is used, within this work, to illustrate these different levels of processing.

A generic tracking operation also requires several levels of processing [Kolbe90], as in image analysis. At the lowest level, sensor data is processed to produce possible locations of objects being tracked. These are commonly incorporated into optimal tracking filters, requiring numeric processing on small data sets. Further operations may use the output from the tracking filters for high level decision making processes, such as that for resource allocation.

1.1.2 Parallel architectures

For the efficient utilisation of a parallel machine, the architecture should match the required computation, communication and structure of the data. In image analysis, the computation is such as to warrant a parallel solution. The question remains however, as to which type of architecture is most efficient. Parallel architectures are normally classified into two categories, according to their control structure, either as Single Instruction Multiple Data (SIMD) or Multiple Instruction Multiple Data (MIMD) [Flynn66]. Each type of architecture has their own advantages and disadvantages.

- SIMD architectures are synchronous with each of the processors operating from a single instruction stream [Hord90]. Thus, they act like a sequential processor in that only a single program needs to be written. An SIMD machine can be implemented by the replication of a single processing element. However, a major disadvantage of an SIMD processor is its instruction bottleneck. If the data set being processed, for any one instruction, is not as large as the SIMD processor array then poor utilisation, and performance, will result. Scaling the size of an SIMD architecture further increases the efficiency problems.

- MIMD architectures do not suffer from the same instruction bottleneck as an SIMD architecture. They enable individual processes to be executed concurrently on each processor. However, due to the asynchronous operation, programming complications can occur such as deadlock. Additionally, the replicated components required for the provision of multiple instruction streams adds to the overall component costs.

The different processing levels, within image analysis operations, are suited to the different control structures within parallel architectures. The low-level operations are suited to an SIMD processor array and the higher-level operations are suited to an MIMD processor array. Thus, a heterogeneous architectural solution might be appropriate. That is, a machine which combines some of the features of both SIMD and MIMD architectures.

The main thesis of this work is the analysis of a novel architecture which contains both SIMD and MIMD processor arrays. This machine is termed the Warwick Pyramid Machine (WPM) [Nudd89, Nudd91, Nudd92a, Nudd92b]. It combines a fine grain massively parallel SIMD array, partitioned into smaller Multiple-SIMD arrays, with a course grained MIMD array. This machine was designed from a study of the requirements of image analysis, and is thus aimed at the combined processing of the low and high levels of image analysis.

1.2 Outline of this Thesis

This thesis contains a detailed description of the design and implementation of the Warwick Pyramid Machine. The performance advantages achievable on this architecture are examined. In addition, a number of image processing and tracking operations are examined and the performance analysed on this architecture. Load-balancing techniques, for the efficient utilisation of this architecture, are also described. The thesis is organised in the following way :

Chapter 2 contains a review of parallel architectures. It classifies architectures according to their memory, interconnection and control structures. The ways in which parallel architectures have exploited these factors are described through a review of existing single array machines. The machines are classified according to their control structure, that is to either SIMD or MIMD control paradigms.

Chapter 3 reviews the operations involved in image analysis. The computational requirements of the different processing levels are examined. Architectures, which have been designed to efficiently exploit the different levels of processing within image analysis, are examined.

Chapter 4 contains a detailed description of the Warwick Pyramid Machine. This machine is a massively parallel heterogeneous architecture combining several processor types. The design of the Warwick Pyramid Machine, the implementation of a prototype, and also the issues concerned with the programming of a dual paradigm architecture, are described.

Chapter 5 contains an analysis of the advantages that this architecture can achieve over conventional parallel architectures. In particular, the capabilities of the Multiple-SIMD array are examined. The mapping of data onto the architecture is illustrated using both image based data and numeric (matrix based) data.

Chapter 6 examines two tracking applications and illustrates how different parts of the processing can be mapped onto the different levels of the Warwick Pyramid Machine. The first is a low density tracking example, from an image analysis application, requiring image operations and the tracking of the size and position of a small number of objects through an image sequence. The second is a higher density situation, treated as a generic target tracking application, where the image pre-processing poses less of a requirement then the actual tracking operations.

Chapter 7 considers the dynamic mapping of data across the Warwick Pyramid Machine, at run-time, to increase the machines utilisation. Load-balancing techniques are considered for both the dynamic mapping of image regions across the machine and for the mapping of sparse data sets.

Chapter 8 draws some conclusions from this work and makes some suggestions for further work.

The contributions to the fields of parallel processing and image analysis contained within this thesis include :

- The design and implementation of an heterogeneous architecture which contains the capability of simultaneously processing low-level and high-level information.

- The novel partitioning of the SIMD array, into a Multiple-SIMD array, within the architecture enabling the instruction bottleneck of conventional SIMD arrays to be overcome. This enables SIMD arrays to be scaled in size without performance inefficiencies.
- The use of the Multiple-SIMD for the novel formulation of some commonly used operations in low-level image analysis.
- The partitioning of tracking applications across the multiple-levels within the Warwick Pyramid Machine.
- The novel formulation of an object size tracking operation that exploits a linear relationship between range and inverse size. This enables a set of size based measurements to be used to estimate the depth of objects from the image plane.
- Novel dynamic load-balancing algorithms which enable the quick re-arrangement of data, both dense iconic data and sparse data sets, across the Warwick Pyramid Machine at run-time.

Chapter 2

Parallel Architectures

2.1 Introduction

Applications, such as image analysis and image generation, require vast amounts of computing power if they are to achieve the real-time operation required in robotics and simulation environments. Other areas such as climatic modelling, fluid dynamics, and vehicle dynamics amongst others, also require Tera-Flop (10^{12} floating point operations) levels of computational power [Rattner91]. The performance of conventional uniprocessor systems cannot currently achieve this level of performance, thus a parallel solution has to be sought.

In this chapter, the options available for the construction of general purpose parallel architectures are described and surveyed. The options include the structure of the memory, the interconnection between processors, and the control structure of multiple processors. The parallel architectures considered here are homogeneous, i.e. those that are constructed from a single type of processor. Later, in Chapter 3, the computational requirements for image analysis are reviewed which leads to a discussion on heterogeneous architectures.

The performance trends within microprocessors are reflected both by the trends in clock speed and in the density of semiconductor technology. These trends have been described by Procter [Procter91] and can be seen in Table 2.1 for ECL and CMOS semiconductor families. In CMOS the clock speed is doubling approximately every 4 years and the density of components is doubling every 1.5 years.

	ECL	CMOS	DRAM
Chip level speed (trend p.a.)	+20%	+20%	+12%
Chip density (trend p.a.)	+35%	+55%	+60%

Table 2.1 - Trends in VLSI clock speed and density.

The difference between the trends in the clock rate and the density can be accounted for by the relationship between them and the minimum feature size. The clock speed scales approximately linearly with the minimum feature size, whereas the density scales as the square of it. If the clock speed is solely relied upon to increase present performances, and the current trends continue, then TeraFlop performances will not be achieved within the next 60 years (assuming a current uniprocessor can achieve 30Mflops).

Performance improvements may also be achieved by the use of processors working concurrently, with increasing semiconductor densities as well as increasing the numbers of processors within a system. However it is not usually a simple task to connect together an ever increasing number of processors so that they can co-operate on a single task. In some cases it has been shown that adding more processors into a system can actually increase the amount of time taken for a given task [Hwang85].

Many parallel architectures have been proposed, some have been built and a few have become commercial products. Several surveys on parallel architectures have been carried out, including that of Duncan [Duncan90] which gives an overview of the options available for parallel processing, and those of Gehringer [Gehinger88] and Lim and Binford [Lim87] who survey existing commercial machines. A survey of current (1991) parallel machines, their costs, availability and configurations, is given by Trew and Wilson [Trew91]. In addition, several books are devoted to parallel processing such as that by Hwang [Hwang85] and that by Almasi and Gottlieb [Almasi89].

This chapter details the options available for the use of processors within a parallel processing system, and surveys existing parallel architectures. In Section 2.2, parallel architectures are described in terms of their memory structure, interconnection schemes between processors, and control structure. In Section 2.3, existing parallel architectures are described and classified according to their control structures.

2.2 Classification of Parallel Architectures

There are several levels in which a processor can be classed as being parallel. At a high level, a processor may be deemed to be parallel if it belongs to an array of such processors, or at a low level parallelism can be seen through bit-parallel operations within an ALU. Duncan [Ducan90] states that processors which employ only low-level parallel mechanisms should be excluded from any parallel classification and viewed only as sequential processors (often referred to as a von Neumann architecture). Low-level mechanisms within a uniprocessor include :

- Multi-bit (word) operation
- Instruction pipelining, allowing overlapping of various components of a single operations such as fetching, execute, and store
- Separate CPU and I/O capabilities
- Super-scalar, where multiple functional units exist such as in the Motorola 88110 [Diefendorff92]. This contains several integer functional units in addition to separate units for floating point addition, multiplication and division.

However the present generation of Reduced Instruction Set Computers (RISC) have ever increasing functionality, and increased performance in the commercial market place. For instance, the Motorola 88110 includes an additional graphics processing unit with the capability of performing integer addition or subtraction on either 8-, 16-, 32-bit words within its single 64-bit ALU. Each word-length is packed so as to fill the 64-bit ALU. Thus the processing of eight 8-bit data words can take place in parallel within a single cycle. One can see that classifying parallelism is not as simple as it may seem.

The main factors which distinguish architecture types are their memory structures, the interconnection topologies between processors and/or between processors and memory, and the control structure of the processors. These are described below.

2.2.1 Memory Structure

An ideal configuration, encompassing both processors and memory, is the Parallel - RAM (P-RAM) model, widely used within theoretical Computer Science, e.g. [Gibbons88]. In this model, it is assumed that a number of processors can simultaneously access any piece of data, from any memory location within the machine, with no overheads or conflicts. Although this is a useful model in the research of parallel algorithms it is not a feasible proposition for implementation. The hardware required to implement n-ported memory increases with the number of processors. In reality, any architecture may be used to simulate a P-RAM model, but with a time penalty due to the control and accessibility of the data within the memory structure.

The P-RAM model is feasible, to some extent, when the number of processors in the network is small. This class of machines is commonly referred to as shared memory architectures. In such architectures all processors have access to a common memory structure. An alternative structure, which is somewhat easier to implement, is that of a distributed memory architecture, in which each processor has its own local memory and communicates to other processors or memories through a separate inter-connection network.

2.2.1.1 Shared memory

Shared memory architectures commonly consist of a number of processors connected to a number of memory banks via a switching network - as shown in Figure 2.1. Each processor can simultaneously access all parts of the memory with equal latency. The switching network is responsible for connecting any of the processors to any of the memories.

Although shared memory systems have no overheads associated with data locality or access, they do have synchronisation problems - those associated with the updating of information in the correct sequence across processors. Techniques such as semaphores must be used. Semaphores enable software synchronisation of the update of data, but result with time overheads in the software.

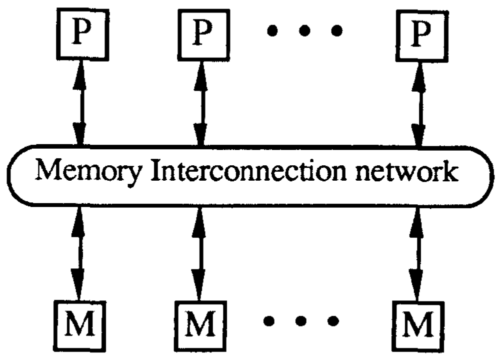


Figure 2.1 - A shared memory system.

However as the number of processors increases, the requirements of a memory switching network results in the system being un-economic to implement. In addition, as the number of processors increases, the chances that more than one processor will access the same memory location at any one time also increases, causing memory contention problems.

2.2.1.2 Distributed shared memory

In distributed shared memory systems each processor has its own local memory, which may be treated as a cache, and forms part of the overall memory structure. Data which is frequently accessed by the processors is stored within these local memories. Other data can be accessed from the local memories of the other processors, but with increased latency time. The structure of a distributed shared memory system is shown in Figure 2.2.

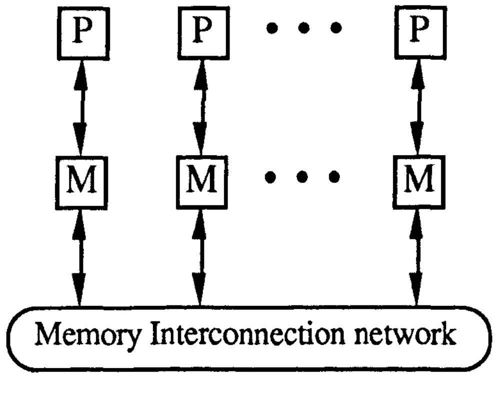


Figure 2.2 - A distributed shared memory system.

The memory network does not need as high a bandwidth as that for shared memory systems since most of the memory access is now between the processors and their local memory. Thus a hierarchical memory system is formed, having a high bandwidth between a processor and its local memory, and a lower bandwidth between a processor and the rest of the shared memory. This system works well as long as the data is efficiently distributed amongst the local memories.

The design of the memory network is not as critical within the distributed system as in the shared memory system since its utilisation is less. This results in a greater potential for scalability with increasing numbers of processors. Problems arise however, when two processors need to process the same data. One solution is to make multiple copies of the data within the processors local memories. This can cause significant cache coherency problems between processors but can be solved for small numbers of processors through a hardware implementation.

2.2.1.3 Distributed memory

In a distributed memory system, each processor is connected only to its own local memory, with no hardware support for global memory access. Instead the processors themselves are interconnected, allowing data to be passed between processors, using either message passing operations in asynchronous systems (e.g. MIMD) or in lock-step in synchronous systems (e.g. SIMD). The structure of a distributed memory system is shown in Figure 2.3.

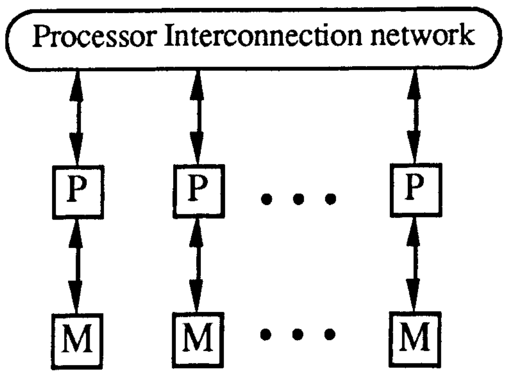


Figure 2.3 - A distributed memory system.

The inter-processor network within a distributed processing system is typically slower than that of the two shared memory systems. Thus the distribution of the data, across a network of distributed memory processors, is more critical than within shared memory systems. Message passing, for inter-processor communication in asynchronous systems, has the potential to hide communication latencies by scheduling further processes during the communication time.

Distributed memory systems scale readily. The required communication networks are not as intrinsically inbuilt as those of shared memory systems. The latter places restrictions on the number of processors. The disadvantage of the distributed system is the increase in latency time for the communication of data between processors. This necessitates optimisation of the data distribution so as to minimise the required communications.

2.2.2 Inter-connection networks

The interconnection network between processors in a distributed system, or between memories within shared memory systems, forms an important feature of any parallel architecture. Various interconnection networks have been proposed and generally fall into two categories - static and dynamic [Feng81]. Static networks result in a fixed interconnection scheme and are typically used for inter-processor communication. Dynamic networks have the capability to rearrange the connections between nodes within the network and are used within shared memory systems.

It is advantageous to use a network topology which matches the communication patterns, in the processing of the data, for which the system will be used. However, for a general purpose system, such information about the data can not be assumed in advance. Restricting the system to image processing reduces the available options as will be discussed in Chapter 3.

2.2.2.1 Bus based systems

The simplest interconnection network is a common bus to which each processor within the network is connected. Bus networks have a low minimum latency for communications between any two processors since no intermediate processors are involved. They are easily constructed requiring few resources, growing linearly with the number of processors.

The major disadvantage of a bus based system is that its throughput is fixed. Only one processor communication can take place at any one time. Thus as the number of processors increases, increasing possible communication traffic, bus contention can become a problem. The effect of this is an increase in average latency times.

2.2.2.2 Static interconnection networks

The performance of a bus system can be improved by increasing the number of connections from each processor. The resulting processor topology affects the maximum latency time for communication between two processors. The maximum latency is proportional to the diameter of the processor network (maximum distance between any two processors). Some static interconnection networks are shown in Figure 2.4 and described below. Reviews of interconnection networks have been given by Feng [Feng81] and by Broomell [Broomell83].

Binary tree - In a binary tree, the maximum number of connections per processor is 3: leaf nodes have 2 and the root has one. Binary trees are particularly useful in supporting divide and conquer algorithms for searching and sorting operations [Duncan90]. Other trees have been suggested but the binary is the most analysed. It is also suited to VLSI implementation. The maximum distance between any two nodes in a binary tree is $2\log_2(N+1) - 2$.

Quadtree - The quadtree can be thought of as a two-dimensional binary tree, with several levels of processors, forming a pyramid and having a single processor on the first level or an apex. Each processor is connected to a single processor in the level above it, and to four processors in the level below it. Additionally the processors can be connected to their four nearest neighbours on the same level. Five connections from each processor are thus required and the maximum distance between any two nodes is $2\log_2(N+1)-2$.

Star - In a star the central processor is connected to all other processors requiring (N-1) connections in an N processor system. The number of inter-connections for the central node makes a star topology impractical for large numbers of processors.

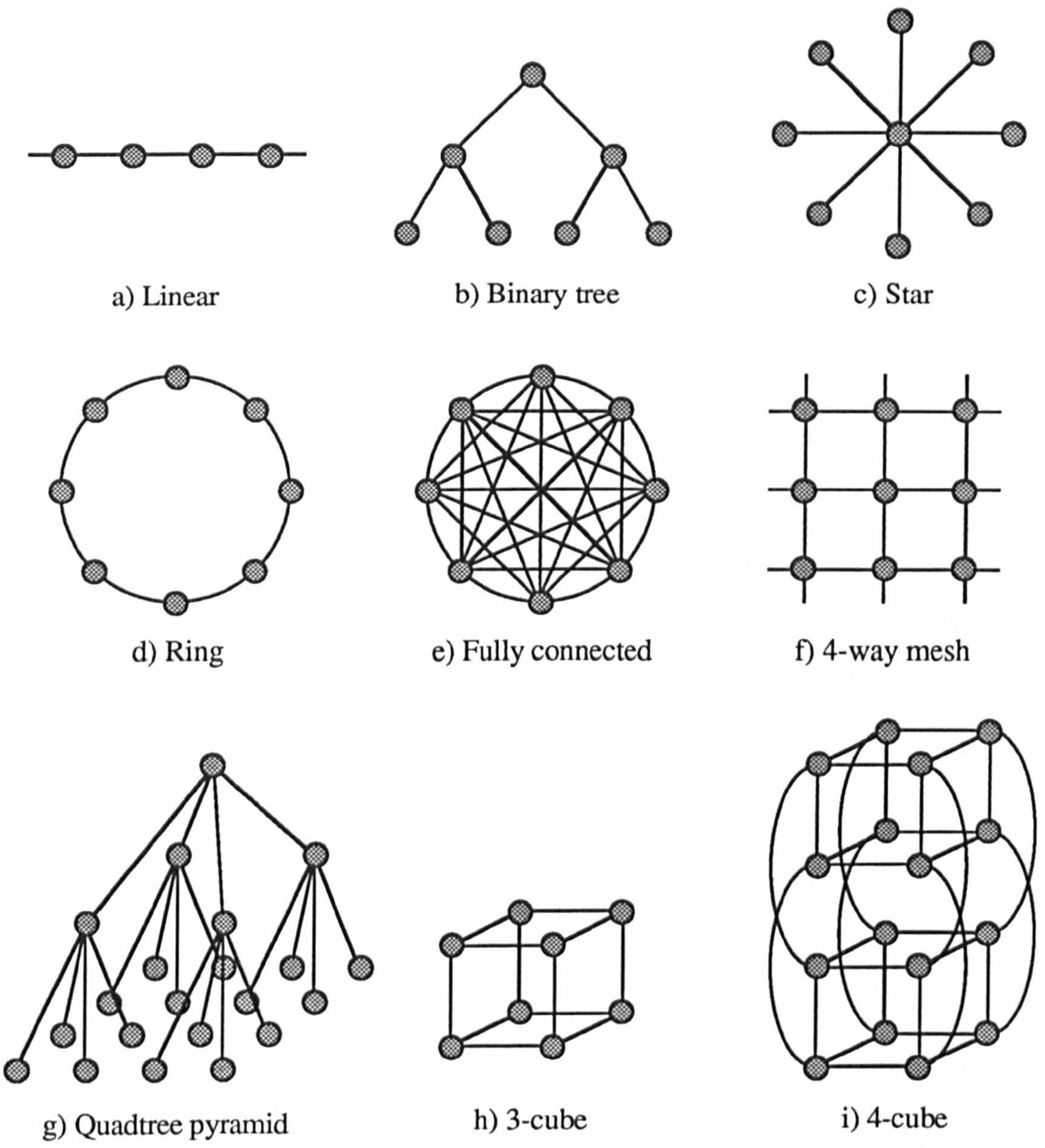


Figure 2.4 - Some static interconnection network topologies.

Meshes (1D) - A one dimensional nearest neighbour network forms a single linear array of processors, each requiring two connections. The maximum distance between any two processors is (N-1) in an N processor system. This distance can be halved by joining the end two processors together forming a ring topology.

Meshes (2D) - Two dimensional processor meshes have been popular in both SIMD and MIMD processor arrays because of their simplicity and scalability. Typically each processor is 4-way connected to its nearest neighbours, forming a grid, and is sometimes referred to as a NEWS (North, East, West, South) network. Other mesh networks have also been implemented: the 6-way mesh which forms a hexagonal grid; and the 8-way mesh in which each processor is also connected to its four nearest diagonal neighbours. The diameter in a 4-way mesh is $2\sqrt{N}$ and in an 8-way mesh is $\sqrt{N}$. These distances can be halved by wrapping around the edges of the array to form a torus network (the top edge connected to the bottom edge and the left edge to the right edge).

Hypercube - Each processor within a hypercube requires $\log_2 N$ inter-connections in an N processor system. The maximum distance between any two processors is $\log_2 N$. The network topology forms a k-dimensional cube ($k = \log_2 N$), three and four dimensional cubes are shown in Figure 2.4. Each processor is addressed by a k-bit binary number. Adjacent processors differ in their addresses by a single bit, determined by which one of the k-dimensions which differs. This leads to simple and elegant communications between processors.

Fully connected - Each processor is connected to each other processor in a fully connected network topology. This requires N connections from each processor with a total of $\frac{1}{2}N(N-1)$ inter-connections in the network for an N processor network. The implementation of such a network becomes unrealistic for large N.

Network	Total number of connections	N=64	Connections per processor	N=64	Diameter	N=64
bus	1	1	1	1	1	1
binary tree	N-1	63	3	3	$2\log_2(N+1)-2$	12
quadtree	N-1	63	5	5	$2\log_2(N+1)-2$	12
star	N-1	63	N-1	63	2	2
linear	N-1	63	2	2	N-1	63
ring	N	64	2	2	$\frac{1}{2}N$	32
2D grid (4-way)	$2(N-\sqrt{N})$	112	4	4	$2\sqrt{N}-2$	14
2D torus (4-way)	2N	128	4	4	$\sqrt{N}$	8
hypercube	$\frac{1}{2}N \log_2 N$	192	$\log_2 N$	8	$\log_2 N$	6
fully connected	$\frac{1}{2}N(N-1)$	2016	N	64	1	1

Table 2.2 - Comparison of some network topologies.

A summary of the connections per processor, the total number of connections within the network, and network diameters is given in Table 2.2 for the networks described above. Also included is an example case for N = 64 processors. The use of the wrap-around on the 1D and 2D meshes can be seen be halve the network diameter while having a minimal effect on the total number of connections. The hypercube networks require more inter-connections than the 2D meshes, but their network diameter is proportional to $\log_2 N$ compared with $\sqrt{N}$ for the mesh. The fully connected system requires a large number of connections (proportional to N^2), and in the example case of N = 64, is ten times greater than any other network.

2.2.2.3 Dynamic interconnection networks

In a dynamic interconnection network, the paths between any two processors is not fixed and connections are made dynamically through the use of switching networks. Their advantage is that the communication paths are set up only when they are required and do not exist permanently. They require less resources than the fully connected static network but potentially have similar communication bandwidths. Such a dynamic network enables each processor to connect with any other processor or, in the case of a shared memory architecture, can be used to connect a processor with any memory bank. Some dynamic crossbar switching networks and multi-stage networks are described below and shown in Figure 2.5.

Crossbar networks

One of the simplest dynamic interconnection networks is the crossbar. The crossbar consists of a set of switches which can directly connect any of the N input nodes to any of the M outputs nodes (a node can be a processor or a memory). Once the switches have been set, data can flow unhindered through the network, experiencing only the time-delay through a single switch. A crossbar switch interconnecting four nodes is shown in Figure 2.5a. The number of switches scales with the product of the number of input and output nodes. The limitations of the crossbar is that a connection can only be made to a node which is not busy i.e. not already connected to another node within the crossbar.

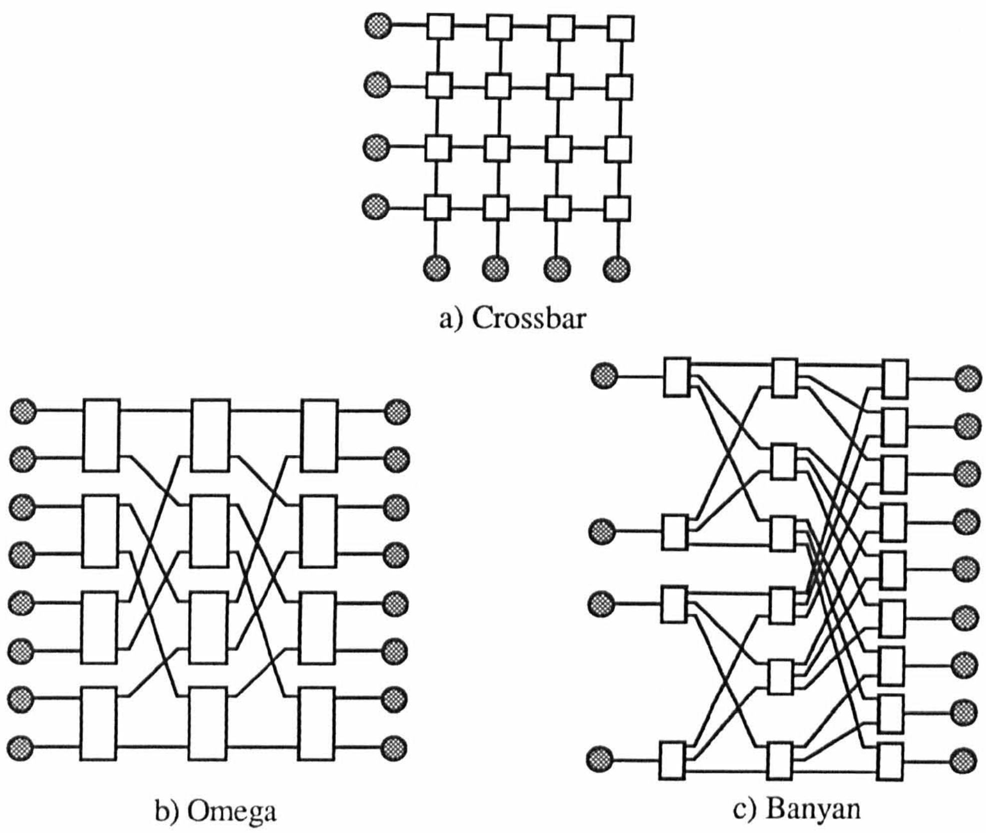


Figure 2.5 - Three dynamic interconnection topologies.

Multi-stage switching networks

The complexity of a crossbar network can be reduced by the use of multi-stage switching networks. An example is that of the Omega network, Figure 2.5b [Lawrie75]. Each switching element has four modes of operation: either passing the inputs straight through; crossing them over; or allowing the upper/lower inputs to be connected to both outputs. Thus each of the input nodes may be connected to any of the output nodes. A total of $N\log_2 N$ switching elements are required in comparison to N^2 elements in a crossbar. However the latency of the communication has now increased by a factor of $\log_2 N$ due to the extra switching stages. As the number of processors increases, the delay across the switching network can become a limiting factor to the use of multi-stage networks.

A multi-stage network may also be used for connection between differing numbers of inputs and outputs as can be seen with the Banyan network in Figure 2.5c. Here the number of inputs to each switching element is two while the number of outputs is three, enabling four input nodes to be connected to 9 output nodes.

Multi-stage networks, including that of the Omega and the Banyan, provide nearly as low a latency as a common bus system, making them ideally suited to shared memory architectures. However contention can arise in a similar manner to the crossbar network if the desired output path to a node is blocked by a path between other input and output nodes.

2.2.3 The control of parallel processors

A classification of parallel architectures in terms of their control structure was described by Flynn in 1966 [Flynn66]. There have been many more attempts since then for such a parallel architecture classification scheme but only Flynn's has obtained wide spread use. Flynn categorised all architectures into one of four groups :

SISD	- Single Instruction Single Data
SIMD	- Single Instruction Multiple Data
MISD	- Multiple Instruction Single Data
MIMD	- Multiple Instruction Multiple Data

These categories are concerned with only two features of an architecture, the number of instruction streams and the number of data streams. The SISD category conforms to a conventional von Neumann architecture which has a single instruction and single data path thus leading to a bottleneck between processor and memories. This is shown in Figure 2.6.

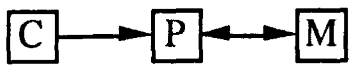


Figure 2.6 - The organisation of an SISD architecture.

The SIMD category again has only a single instructional stream but has multiple data paths between processors and memory. The processors are required to perform the same operation but use their own operands. This is shown in Figure 2.7. The bottleneck in the SIMD architectures is not the interface between processors and memory, but is in the single instruction path giving the processors little or no operational autonomy.

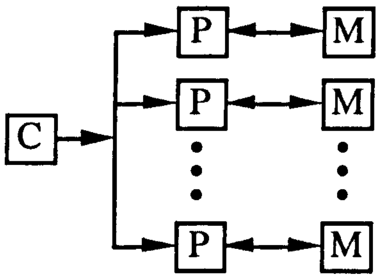


Figure 2.7 - The organisation of an SIMD architecture.

The third category, MISD, with multiple instruction streams and only a single data stream is not well defined - how can several operations be performed on the same data at the same time? Flynn suggests that this category suits machines in which either each processor has an instruction stream and all data is stored within a shared memory, or a pipelined machine in which the processors perform their own operations on data which is then passed down a pipeline. The organisation of the MISD architecture is shown in Figure 2.8.

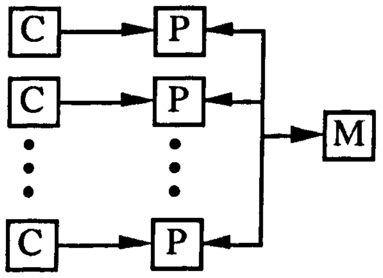


Figure 2.8- The organisation of an MISD architecture.

The final category of architecture is that of MIMD which has both multiple instruction and multiple data streams. One such configuration of an MIMD architecture is that of multiple-SISD (M-SISD) processors. The configuration of an MIMD architecture is shown in Figure 2.9.

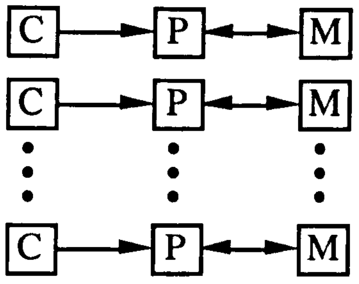


Figure 2.9 - The organisation of an MIMD architecture.

In Flynns classification, only two important categories enable parallel processing to take place - namely that of SIMD and MIMD. Both architectures enable operations to take place on multiple pieces of data at any one time through the provision of multiple data streams and additionally MIMD has multiple instruction streams. A considerable number of parallel machines have been designed, some of which conform to SIMD control and some to MIMD. Several architectures have progressed into commercial products, examples of which are described in Section 2.3.

The majority of the machines which have been implemented have been designed in the form of two-dimensional arrays, although some incorporate a hyper-cube interconnection topology. Array topologies offer scalability in that further processors may be added, forming a larger array, without altering the existing processors to a great extent. One common question which arises in the design of an array is whether the control structure should be SIMD or MIMD?

2.2.4 SIMD vs. MIMD arrays

SIMD and MIMD arrays differ fundamentally in their control structure. An SIMD array has a single instruction stream, whereas an MIMD array has an instruction stream for every processor in the array. The instruction stream is provided by a sequencer, which addresses the instruction memory, and providing added functionality such as looping and branching etc. Such a functional unit represents a control overhead, in terms of gates or silicon area, which could be used to implement further arithmetic functionality. The control overhead is far greater in an MIMD array than in an SIMD array. Thus in an SIMD array, a greater proportion of the active components can be used for the functionality within the processors.

2.2.4.1 Active Resources

A comparison between SIMD and MIMD arrays using a measure of active resources (e.g. gate counts), was given by Uhr in 1982 [Uhr82]. Uhr calculated that 10% of the total gate count in a serial processor (as used in forming Multiple SISD arrays) were active in comparison to 50-66% for an SIMD array. Although the comparison used technologies and machines in existence at that time, the use of active resources to compare architectures remains mostly valid today.

A comparison made using active resources is only valid if those parts forming the functional units of the processors within the comparison are being fully utilised. This can be viewed at one of two levels: at an operation level where the size of the data effects the utilisation of each processing unit; and at a higher level where the mapping of data elements across the array will effect the overall array utilisation.

An analysis of the utilisation of the functional unit of the processors, when considering different word-length of data, has lead to the conclusion that a set of single-bit processors is more efficient than any other type of processing unit [Hillis85, Uhr82]. Such a processor, commonly termed a single bit Processing Element (PE), can perform any operation that a larger word-length processor can do, but in increased time due to their bit-serial operation.

The utilisation of a processor on different data sizes can be seen through the example of a 1-bit and a 32-bit integer addition. A 32-bit addition takes a single cycle on a 32-bit processor, but 32 cycles bit-serially. If it is assumed that 32 bit-serial processors can be implemented using the same amount of resources as a single 32-bit processor then 32 such bit-serial additions can be performed in parallel. A one-bit addition on a 32-bit processor and on a bit-serial processor takes a single cycle to perform. However the bit-serial processors can do 32 such operations in parallel - an improvement by a factor of 32 over the 32-bit processor. In fact, a bit-serial processor may perform an operation on arbitrary sized data with no wastage of cycles.

Operations which require the multiple access of operands, such as that for the alignment of the mantissa in floating-point operations, results in poor performance on bit-serial processors in comparison with an integral floating point unit within a uniprocessor. However a floating point unit within a uniprocessor lies idle on all non-floating point operations. This can lead to poor utilisation of the resources. The resources given to any arithmetic unit, such as a floating point one, may be used to provide an enlarged processor array having an increased performance.

The number of processors within an array is limited, it is affected by the amount of data one wishes to process in a data parallel fashion. This results in a complex trade-off between the number of possible processors in an array and the array utilisation. For instance, if the amount of data to be mapped across the array is less than the number of processors then some processors will lie idle, resulting in poor utilisation.

2.2.4.2 Technology

Another issue in comparing SIMD and MIMD processing systems is the technology available for VLSI implementation. The numbers of processors which can be implemented within a single IC package are severely limited by the package pin count. The number of components that can be placed on a piece of silicon scales inversely to the square of the feature size. The number of pins on an IC depends upon bonding techniques on the edge of the IC, these have scaled more slowly over the past two decades [Hennessy91]. The package pin-count is of prime importance for a bit-serial processor array where, if it is assumed that memory for each of the processors is implemented using current memory technology off chip, a separate IC pin is required from each bit-serial processor to its memory. Thus the number of pins required scales linearly with the number of processors in the IC.

The pin-count limitation on current IC packing is about 500 pins [Asthana89], less control and power requirements. Therefore it is feasible for a 16x16 bit-serial SIMD array to be implemented on a single chip, with each PE requiring a single pin. Due to this limitation it may be more efficient to implement processors, with larger word-lengths than bit-serial, to utilise available on-chip resources which may not require as many external connections. Several SIMD processor designs have appeared which use enhanced functional units for their processors. These include support to enhance the performance of floating point operations e.g. the DAP co-processor [AMT90], the paper design of the GPFP SIMD array [Beal90] and the μPA [Jesshope89].

A further technology consideration is the clocking system of the processor array. A bit-serial processor has the potential for a higher clocking rate than a larger grain processor due to the non-existence of carry propagation delays within the processor ALU. However such a clock rate can rarely be achieved if the array uses a large wordlength ALU for its sequencing.

The distribution of the clock can also become a limiting factor in the scaling of an SIMD processor array [Jesshope89]. It becomes increasingly difficult to ensure that all processors will be synchronised, as the array increases in size, due to clock and instruction stream distribution problems. MIMD processors do not suffer in the same way, having both have local clocks and instruction streams.

The possibility of using wafer-scale integration for parallel computers is also being investigated. Wafer-scale integration has been primarily suggested for use in SIMD machines, where the processors may be easily replicated across the silicon, but could be equally be applied to MIMD machines. Example wafer-scale SIMD processors include the WASP bit-serial associative SIMD processor [Lea88], and the 3-dimensional wafer stack from Hughes [Grinberg84]. The use of this technology promises to alleviate any pin-count problems while increasing the density of components in a unit volume.

2.2.4.3 Programming

An MIMD processor array requires the implementation of message passing operations to be used for the communication of data between processors, unless suitable software is available to hide the available parallelism. Problems which may arise from this situation include deadlock.

An SIMD array is considerably easier to program than an MIMD array, acting like a sequential processor in the way in which code is written, requiring only a single instruction stream. To ensure maximum utilisation of the processors, where possible, additional software complexity can arise for the manipulation and mapping of data across an SIMD array.

SIMD arrays can be efficiently utilised in data parallel problems - problems which contain sufficient data so as to require the use of all the processors in the same way across the array. They can suffer limitations if the operation being performed does not require data parallel computation but instead control parallel computation.

MIMD processor arrays do not suffer the same inefficiencies, in that a data parallel operation may be performed on the MIMD array as if it were a control parallel one, using the message passing operations rather than synchronous communications (Single Program Multiple Data - SPMD operation). However, problems can arise in the debugging of asynchronous programs since the errors can be non-deterministic.

2.2.4.4 Local autonomy

The disadvantage of an SIMD array, and its single instruction stream, may be improved upon by adding local autonomous functions to the processing elements. A discussion on the autonomy options available for PEs was given by Fountain [Fountain88a] along with the cost in terms of percentage increase in hardware complexity. Local autonomy options include :

- Activity control. At its simplest level, this involves a single-bit flag in each PE which indicates if the result of the operations should be written back to memory thus effectively disabling a set of PEs.
- Data control. This enables the address of data within local memory or the local connectivity functions to be calculated locally which otherwise would be calculated globally.
- Function control. This allows the processor to locally control functional units such as barrel shifters or multipliers. This can be useful for the alignment of mantissas in floating point operations.
- Operation control. This provides each PE with a local instruction store, which is sequenced globally, allowing sets of processors to perform different operations.
- Sequencing control. This provides each processor with its own instruction stream thus forming an MIMD processor array, e.g. the Transputer.

The gradual addition of these autonomous functions to an SIMD processor transforms it into an MIMD one. The percentage overhead on the activity, data, function, operation and sequencing control was calculated by Fountain to be 1%, 6%, 1%, 30%

and between 25-600% respectively. It was assumed that each of the processors contained a 16-bit ALU for this calculation.

The local autonomy considered by Fountain was limited to the functions that could be added to each individual processor. Maresca [Maresca88] suggests that a more useful and practical level of autonomy is the logical division of processors into clusters, with each cluster having local autonomous control. One example of this is that of a Multiple-SIMD (M-SIMD) architecture which provides local autonomy in the form of instruction sequencing across sets of PEs. This approach has been used in the PASM processor array [Siegel81] - a configurable MIMD/SIMD system.

A discussion of M-SIMD architectures was given by Hwang [Hwang80, Hwang85]. Hwang used a model containing r PEs which could take their instruction stream from any of m control units using a dynamic reconfigurable switching network as shown in Figure 2.10. On an M-SIMD architecture the resources required for any operation, in terms of PE numbers, can be matched to the amount of data parallelism within it.

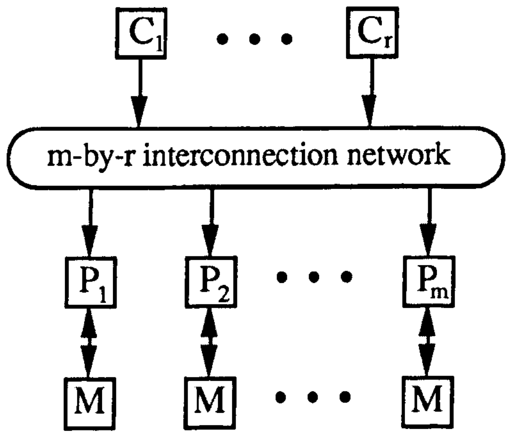


Figure 2.10 - The organisation of an M-SIMD architecture.

Local autonomy across sets of processors in the form of a pyramid M-SIMD architecture has been used by Cantoni et. al. [Cantoni87]. Cantoni used bit-serial processors to form a quadtree pyramid of processors. Each processor layer within the quadtree was controlled by separate controllers, thus giving each layer the ability to perform different operations.

2.3 Survey of Existing Parallel Machines

Existing single array parallel machines are described below. They are categorised according to their control structure, i.e. either to SIMD or MIMD. SIMD architectures are further classified according to the granularity of their processing elements. Many SIMD processors have been built using single-bit PEs but more recently incorporate multiple-bit PEs as a result of the increased integration possible in VLSI. MIMD architectures are classified according to their memory structure, either shared memory, distributed shared memory, or distributed memory.

2.3.1 SIMD architectures

The configuration of SIMD machines, designed to date, have been similar and can be characterised by an array of PEs each with its own memory and controlled by one or sometimes more array controllers. A characteristic SIMD array is shown in Figure 2.11 where the interconnection scheme is a two-dimensional mesh. The functionality of the PEs, coupled with their interconnection schemes and the technology in which they were implemented, gives rise to the performance of each SIMD system.

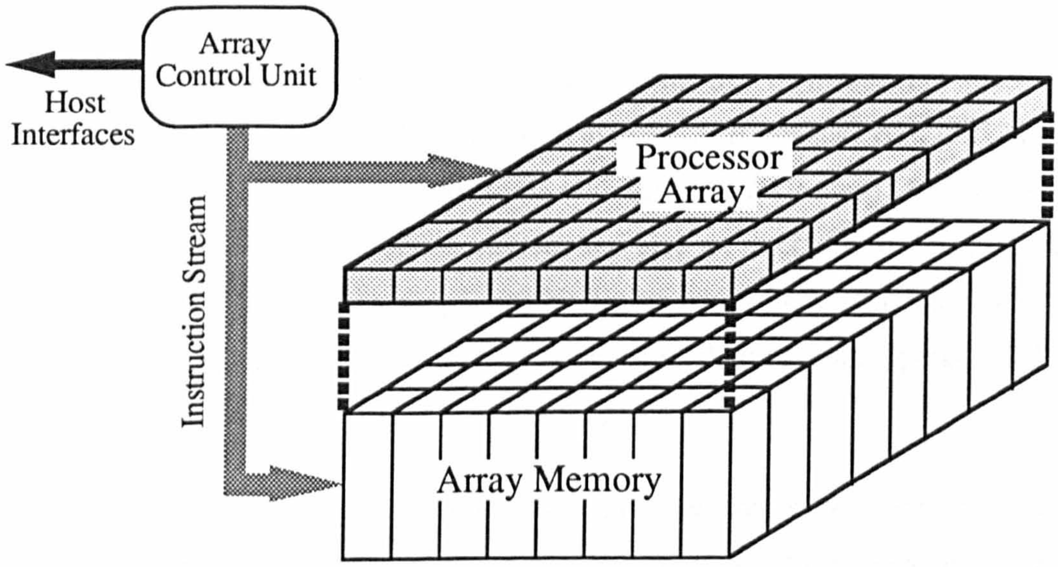


Figure 2.11 - The configuration of a two-dimensional SIMD array.

A pioneer project to implement the first large scale SIMD machine, known as ILLIAC IV, started in 1966 at the University of Illinios [Barnes68]. The original design incorporated 256 64-bit PEs, each with 2Kwords of memory, and a controller for each set of 64 PEs. Each PE had hardware support for floating point operations and could perform a 64-bit multiplication in 400ns. The prototype machine, built in 1972, contained only 64 PEs and was operational from 1975 to 1981. The time lag before the machine became operational was due to the debugging of the hardware, being implemented in the available technology at that time. Each PE used 210 printed circuit boards with the entire machine containing a staggering six million discrete components [Almasi89].

Since the implementation of ILLIAC IV several other SIMD machines have been proposed and built. They can be classified in terms of the granularity of their PEs. Many SIMD arrays use bit-serial PEs, including the Massively Parallel Processor (MPP), the Distributed Array Processor (DAP), the Connection Machine (CM). More recent SIMD designs have used multi-bit processors. These include the Massively Parallel machine (MasPar) and a co-processor to the DAP. The main characteristics of these architectures are described below.

2.3.1.1 Bit-serial SIMD processor arrays.

The Massively Parallel Processor (MPP).

The MPP [Batcher80] was built by Goodyear and was first operational in 1983. It is to date one of the largest SIMD processors ever built. It consists of 128x128 PEs forming a two dimensional mesh (or torus with array wrap-around). A single array control unit provides an instruction stream, and also an interface to a host (VAX) computer. Each of the PEs is a simple one bit design with additional functionality, in the form of a shift register, to aid the performance of floating point operations. The MPP processing element is shown in Figure 2.12.

The single-bit PE contains a single full bit adder, two general purpose registers, A and P, and a further two registers to store the sum & carry output from the adder, B & C. The shift register has a programmable length of between 2 and 30 bits and can be used to supply data to the adder. The P register is used for neighbourhood (4-way) communications and is associated with logic which can perform any logical function on two variables. Additionally the PEs are connected from West to East via a set of shift registers, S, for fast data I/O. Activity control is provided by the G register. Each PE can address 1Kbits of local memory. A custom designed IC, containing 2x4 PEs, is used to build up the whole MPP array.

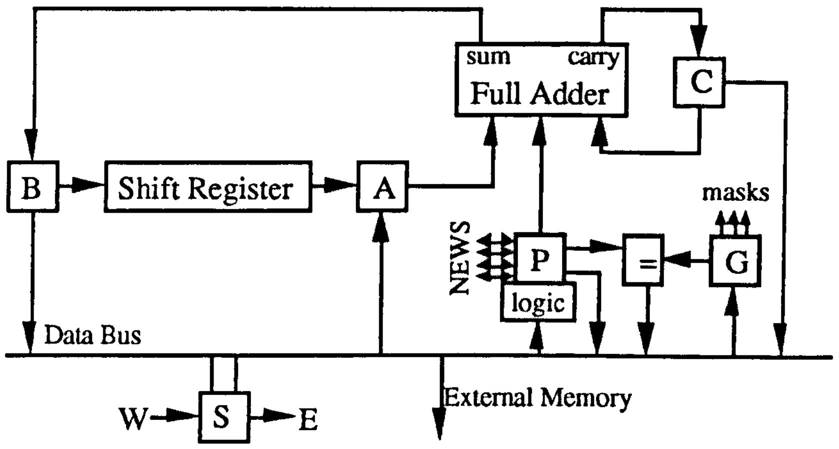


Figure 2.12 - The MPP processing element.

More recently, an enhanced version of the MPP processing element has been used within the design of a custom IC. This accommodates 8x16 PEs, using recent technology. The resulting machine has been called BLITZEN, the details of which have been described by Blevins et. al. [Blevins90]. The major differences between the MPP and BLITZEN include 1Kbits per PE on chip, reduced access to off-chip memory (4pins per 16 PEs), 8-way connectivity and an additional local control function of on-chip memory address modification.

The Distributed Array Processor (DAP).

The DAP is a further example of a bit-serial SIMD processing array similar, but with simpler PEs, to that of the MPP. The design of the DAP PE has remained virtually the same since the its conception in 1972, pre-dating the MPP. The design of the PE was constrained by the technology available at that time. Thus a main design requirement was to minimise the number of ICs required for each PE. The first DAP PE, as described by Reddaway [Reddaway73], required ~3 discrete IC's for each PE (excluding memory devices). It consisted of a one-bit full adder, 7 one-bit latches, associated routing circuitry, and 4Kbits memory.

The present PE design dates from the mid 70's, having fewer registers per PE. In 1979 four of these PEs could be implemented on a single IC using 200 gate equivalents [Reddaway79]. The commercial DAP product uses a custom IC containing an array of 8x8 of these bit-serial PEs using ~3,000 gates. The current DAP machine consists of an array of either 32x32 or 64x64 PEs.

Each PE consists of a full adder, with 3 inputs and 2 outputs, three general purpose registers and two multiplexors to route the inputs to, and outputs from, the adder. The PEs are four way connected in a NEWS communications mesh and also connected to row and column highways for associative response operations. Additionally the PEs are connected from South to North for fast autonomous data I/O. A PE can address up to 64Kbits of external memory. Each DAP instruction effectively combines the states of the internal registers with a memory value within a single instruction cycle.

The functionality of the DAP is further explored in Chapter 4.

The CLIP architecture

The Cellular Logic Image Processor (CLIP) series of architectures were designed and built by Fountain and Duff at University College London. The design of the CLIP-4 processor took place over the ten years to 1980 and is described by Fountain [Fountain87]. The prototype CLIP-4 consisted of a 96x96 processor array containing bit-serial PEs like the DAP and MPP. A commercial product consisting of a 32x32 PE array was also marketed.

Each CLIP-4 PE contained 4 registers, on-chip 32-bit memory, and a Boolean functional unit able to perform any logical combination on two single bit operand inputs. A network of adders was included in the CLIP-4 array to generate the sum of the number of bits set in a binary image. The PEs were arranged in an eight connected two-dimensional network. Eight PEs were implemented on a single custom IC.

The Reconfigurable Processor Array (RPA)

The RPA was designed by Jesshope et. al. [Jesshope87] at Southampton University. The goal of the RPA was not to improve the peak computational power of each PE but instead to allow dynamic configuration of the processors to match the parallelism within an application. The flexibility was thus aimed at increasing the overall utilisation of the array by reducing the number of idle processors at any one time.

The PEs within the RPA were again bit-serial processors arranged in a 32x32 mesh but also included local communication configuration control. This control enables a chain of PEs to be constructed to form an arbitrary carry propagation path, resulting in multi-bit Processing Units (PUs). The reconfiguration of the array is performed through local registers within each PE. Examples of this reconfiguration are shown in Figure 2.13 [O'Gorman89] for PUs of size 2, 4, 6, and 8-bits. Each PE also included 64-bits of memory, an 8-bit deep activity control stack, and support for floating point operations including a barrel shifter.

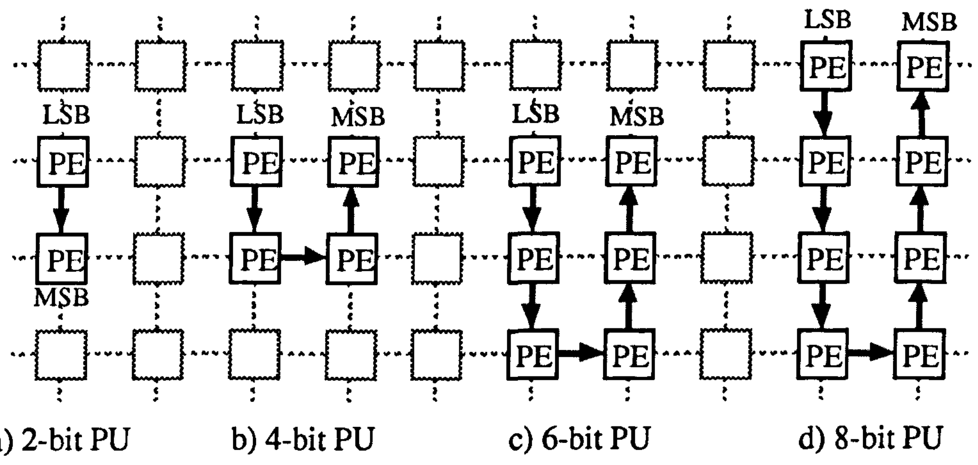


Figure 2.13 - Example reconfiguration of the RPA PEs.

The performance of the added functionality within the RPA PE was found to be minimal and in some cases (e.g. floating point multiplication) even worse than the simpler DAP PE [Rushton89].

The Connection Machine

The Connection Machine (CM) was originally designed by Hillis at MIT [Hillis85] from a study of the needs of AI. From 1983, the CM was manufactured by Thinking Machine Corporation (TMC) [TMC89]. Three different machines have appeared - the CM1, CM2 and CM5. The CM1 and CM2 series are both SIMD machines having a maximum of 65536 bit-serial PEs. The recently announced CM5 is an MIMD architecture. The configuration of the CM2 machine is described here.

Each of the PEs within a CM2 has a simple bit-serial design, consisting of five 1-bit registers, two 1 of 8 multiplexors and an activity flag. The multiplexors can perform any function of three input values. A total of 65536 PEs are used within the largest CM2, connected in a 12 dimensional hypercube with 4096 nodes. Each node within the hypercube consists of sixteen PEs, with communication support, implemented on a custom VLSI chip. Two nodes are shown in Figure 2.14.

A hardware router to support packet switching communication, between any PEs across the hypercube network, exists at each node. The hypercube network can also be used for inter-PE communications, using various network topologies, such as the two dimensional mesh, offering increased bandwidth over the packet router. Note that each node has only one communication connection with its adjacent nodes in the hypercube and so inter-PE communications across node boundaries take place sequentially for each PE within the node.

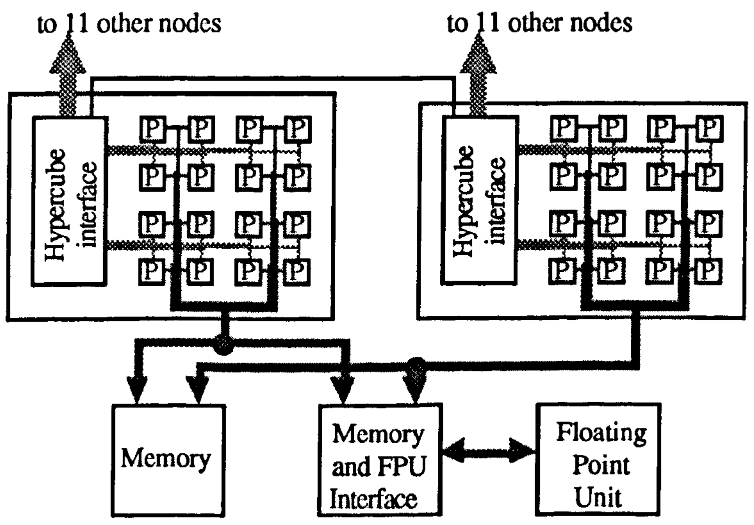


Figure 2.14 - Two CM2 Nodes with associated memory and floating point unit.

A floating point unit, with associated interface circuitry, is an optional part to each set of two nodes within the CM2, as shown in Figure 2.14. This unit considerably enhances the floating point capabilities of the CM2 bit-serial processors. The interface unit is used to convert (corner turn) data words from the bit-serial oriented format to a word oriented one, suitable for use in the floating point unit. A total of 2048 floating point units exist in the full sized CM2 machine.

The PEs within the CM2 are controlled by a set of four sequencers, allowing up to four separate programs to be executed concurrently. Each sequencer can control a quarter of the array or alternatively work in unison on a single program across the entire PE array.

Associative SIMD arrays

A further type of SIMD processor array that has been investigated is associative SIMD arrays. They are distinguished from other SIMD arrays by the type of local memory each processor has - Content Addressable Memory (CAM). The CAM enables PEs within the array to be identified by their memory contents rather than their position within an array. Typically a qualifying instruction is broadcast by the SIMD controller, part of which is passed to the CAM for matching, which sets the activity control within the PEs to determine which will perform the subsequent operations.

Two examples of an associative processor array are the GLiTCH [Duller89] from the University of Bristol and the ASP [Lea91] from Brunel University. The two approaches are similar. Both use single-bit PEs, each with between 64-70 bits of CAM, and local activity control can be set from the output of a CAM match. The PEs are arranged in a one-dimensional array but have support for communications over greater distances between PEs on-chip. A total of 64 PEs have been implemented on a single IC for both the GLiTCH and ASP. Recently, the ASP has been proposed as a suitable Fault Tolerant processor array suited to wafer-scale integration [Lea88].

2.3.1.2 Multi-bit SIMD processor arrays.

The increase available in VLSI integration has been utilised within SIMD processor array designs by enhancing the performance of the PEs. Improved performance has been achieved by both increasing the wordsize of the PEs, from single-bit, and by adding circuitry to increase local autonomy, typically optimised for floating point operations.

DAP with CP8 co-processor.

Recently, each of the DAP PEs has been complimented by the addition of a co-processor unit - the CP8 [AMT90]. The CP8 contains an 8-bit ALU capable of an 8-bit addition in a single cycle, a 32-bit shift register able to shift by bytes or bits under local data control, and a serial I/O interface to external memory. It works concurrently with the existing 1-bit DAP PE, improving floating point performance by an order of magnitude. Neighbourhood communications and Boolean operations are still carried out by the 1-bit DAP processors.

The CLIP-7 processor

The experience gained in the design and use of the CLIP-4 was used in the design of a multi-bit CLIP-7 processor [Fountain88b]. Its goal was to examine the various options available in the transformation from an SIMD processor to an MIMD one by the addition of different degrees of local autonomous functions. Each CLIP-7 PE consists of a 16-bit ALU having both activity and function control along with the local generation of memory address. The instruction stream remains that of the conventional SIMD processing system, utilising a single array controller. The first system to be constructed from the CLIP-7 chips was a 256 PE linear array.

The μPA processor

The design of the RPA has been improved with the design of the μPA, detailed by Jesshope et. al. [Jesshope89]. The PE in the μPA contains an 8-bit ALU, an 8-bit multiplier, 1Kbytes internal memory, which can be locally addressed, and a communications co-processor. Support for floating point operations, and a bit-serial interface to external memory, is also included. It is also proposed that each μPA chip (currently estimated to contain 4 μPA PEs [Jesshope89]) would contain a local sequencer and microcode which itself is under the control of a central array controller.

The communications co-processor handles all inter-PE communications either in a packet switching node, using a hardware router, or in NEWS mode allowing for global SIMD shifting operations over the two-dimensional grid.

The MasPar MP-1 Architecture

The Massively Parallel (MasPar) MP-1 is one of the latest SIMD machines (1989) to appear as a commercial product [Nickolls90, Christy90]. Its design is a departure from the bit-serial approach of the MPP, DAP and connection machine. The available level of integration has been used so as to enhance the performance of each PE, making them multi-bit (4-bit) ALUs with added floating point support circuitry. The MasPar MP-1 contains between 1K and 16K PEs implemented in custom ICs, each containing two clusters of 16 PEs, operating with a cycle time of 70ns.

Each PE consists of a 4-bit ALU, a 1-bit logic unit, a 16-bit exponent unit, a 64-bit mantissa unit with local control optimised for floating point operations, and 40 32-bit on-chip registers. An additional 16Kbytes of memory per PE exist off-chip which can be locally addressed by the PEs, or globally addressed by the array controller. Local addressing to external memory is a factor of three slower than global addressing.

The PEs are interconnected via two communications networks. The first is a static local neighbour network, the X-Net, enabling a PE to communicate with any of its 8 neighbours. The second network is a three stage crossbar router which enables any cluster of 16 PEs to communicate with any other cluster of PEs. Each of the routers contains 1024 ports enabling a total of 1024 simultaneous connections to be available at any one time. Communications across this router requires the originating PE to transmit the destination PE address first, followed by the data. A connection takes 40 cycles to establish if the destination cluster is not busy, and 10 cycles to close. Since the router is effectively multiplexed between 16 PEs within each cluster, its communication bandwidth is a factor of 16 less than that of the X-Net.

Recently, the second range of MasPar machines has been announced, the MP-2 [Nass92]. This machine uses 32-bit processors and boasts a performance increase of a factor of five over the MP-1. A custom VLSI chip contains a total of thirty-two 32-bit processors.

The General Purpose Floating Point (GPFP) support PE

Bit-serial floating point operations have been examined by Beal and Lambrinoudakis [Beal90, Lambrinoudakis91], and compared with that of the DAP PEs with and without the CP8 co-processor. The outcome of this was the design of the GFPF which retains the flexibility to act like a conventional bit-serial processor but includes support specific to floating point operations.

The performance of floating point operations has been greatly improved in the GFPF PE by the use of a 4-bit ALU, a 4-bit multiplier unit, an alignment/normalisation unit and on-chip 256bit memory with local address modification circuitry. These units reduce the cycle counts for the floating point operations to a bare minimum. It is estimated that on the GFPF single precision floating point addition would take 20 cycles and multiplication 61 cycles [Beal90] - an improvement by a factor of 50 over the DAP and by a factor of 2-3 over the DAP with CP-8 co-processor.

2.3.2 MIMD Architectures

There has been a large proliferation of MIMD machines compared with SIMD machines. This can mainly be accounted for by the construction of many MIMD machines from existing commercial uniprocessors, built in a Multiple-SISD fashion, and having additional support for inter-processor communications. Thus the development of custom VLSI processors is not usually a problem, and performance advances can be achieved by the use of faster uni-processors.

Some common MIMD architectures are described below and classified by their memory structure. The most common memory structure is that of distributed memory, due to its ease of design (typically in a Multiple-SISD form). The use of the Transputer has also lead to an expansion in distributed memory machines.

2.3.2.1 Shared Memory machines

The Encore Multimax

The Encore Multimax is constructed from up to 20 National Semiconductor 32332 processors, each with an associated Weitek floating point unit. Each processor has a 64Kbyte cache and is connected to a maximum of 320 Mbytes of shared memory [Gehinger88]. The processors and memory are connected via three common buses, having a total memory bandwidth of 100 Mbyte/s. This bandwidth is shared by all processors in the system and forms a bottleneck to the addition of further processors.

A recently announced range from Encore (the Encore 91) [Trew91] uses an enhanced bus between its memory and processors. It is hierarchical in structure, with each set of four processors sharing a single bus with a bandwidth of 100Mbytes/s, and each set of four processors connected to a lower capacity bus (26.6Mbyte/s). A total of eight sets of four processors can be interconnected in this way. The processor used in this machine is the Motorola 88100, having improved floating point performance over the Weitek FPU, and has either two or four Cache memory management units each with 16Kbytes of associative cache memory.

DASH

DASH (Directory Architecture for SHared memory) is a prototype machine, aimed at investigating cache-coherence problems in large-scale machines, being developed at Stanford University [Lenoski92]. It will eventually consist of 64 processors, 16 of which are currently implemented, arranged into clusters. Each cluster contains four MIPS R3000 processors, each with an R3010 co-processor. Connected to each of the processors within a cluster is a memory hierarchy consisting of a two level cache and shared main memory. The memory within other clusters can be accessed using a two-dimensional mesh interconnection scheme. The machine can thus be classified as shared memory within a cluster and distributed shared memory between clusters. Four clusters of the DASH prototype are shown in Figure 2.15.

The first level cache consists of two 64Kbyte caches used for instruction and data caching, and operates synchronously with the processor. The second level cache is a 256Kbyte data cache connected to the processor with a bandwidth of 16Mbytes/s. A common bus, with a maximum bandwidth of 64Mbytes/s, within the cluster connects the processors to the local shared memory. Inter-cluster communications are performed using a pair of wormhole routed meshes, one for request messages and the other for replies. This network has a peak bandwidth of 120Mbytes/s.

Early results have shown that near-linear speedups can be obtained on DASH using real applications, when implemented using load balancing techniques [Lenoski92, Singh92]. It remains to be seen if such hierarchical shared memory interconnection schemes can be extended to systems with 100's to 1000's of processors.

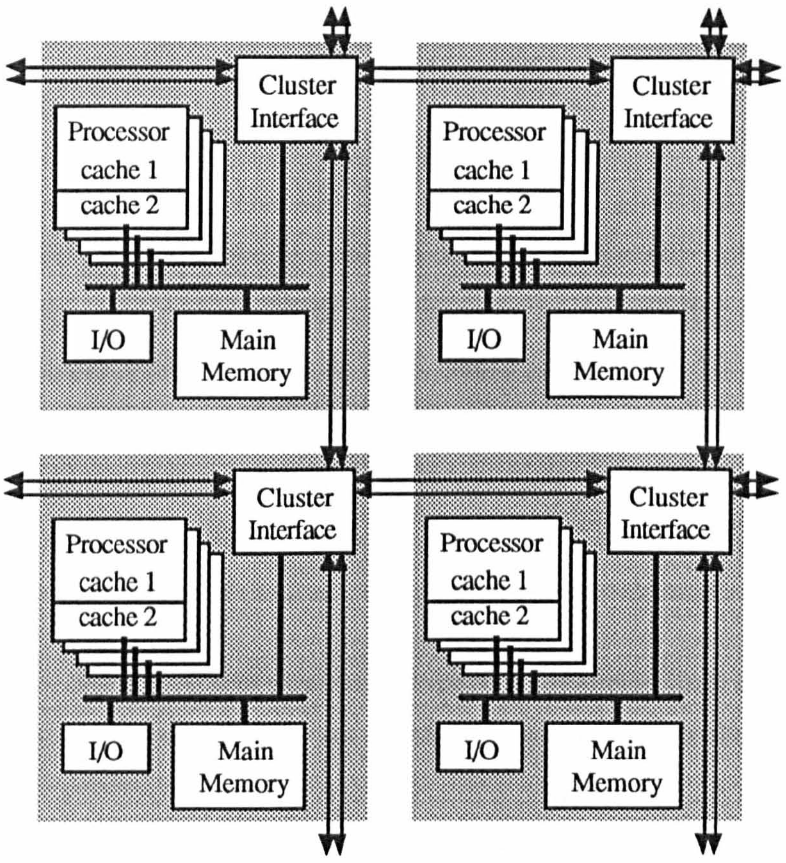


Figure 2.15 - Four clusters from the DASH prototype.

2.3.2.2 Distributed shared memory

IBM RP3

An example of a distributed shared memory machine is that of the IBM RP3 (Research Parallel Processor Prototype) [Pfister87]. Each node within the RP3 consists of a proprietary 32-bit RISC processor with a 32Kbyte cache, an FPU and up to 8Mbytes of memory. A total of 512 nodes can be contained within the machine. A memory management unit enables non-local memory access to be performed over an Omega network, interconnecting all nodes with an total bandwidth of 12.8Gbytes/s.

Local memory references take 100ns while non-local references take 500ns across this network, assuming that their is no contention. This is typical of a distributed shared memory system in which the memory access latency increases as the memory accessed moves further from the processor. These machines are scalable to a greater extent than shared memory machines, allowing a larger number of processors to be supported.

2.3.2.3 Distributed memory

The Transputer

The Transputer [Inmos85] was the first conventional processor incorporating hardware communication, allowing a parallel network to be easily constructed. It was first introduced in 1985 and has appeared in several forms, including a 16-bit version and a 32-bit version with a FPU - the T800. More recently, a 64-bit floating point version has been announced, the T9000, and includes enhanced inter-processor communication facilities [Inmos91]. Several independent companies including Meiko, Parsys and Parsytec, were created around the Transputer product. These companies now hold a large percentage of the European parallel computing market [Trew91].

The performance of a Transputer machine is typically calculated from the performance of a single processor multiplied by the number of processors in the machine. The 30MHz of the T800 Transputer can operate with a peak instruction rate of 30MIPS and 4.3Mflops. It has four bi-directional communication links which can operate asynchronously at a speed of 20Mbits/s. The machines offered by the different manufacturers differ mainly in their interconnection networks, I/O support and software support.

Meiko - The Meiko computing surface is, in theory, expandable indefinitely with the largest system produced containing 1024 processing nodes [Barr89]. A board in the computing surface can contain 16 processors, which are fully dynamically interconnected, each with up to 4Mbytes of memory. The boards fit into a common back-plane, over which non-board level communications can be performed. Dynamic connection between all processors cannot be guarantied in this system.

Parsys - The Parsys SN1000 series machines originated from the ESPRIT Supernode project [Nicole88]. It can contain up to 1024 processors arranged into modules of 16 with associated control and I/O support (a SuperNode). Full reconfiguration between the Transputers is possible using two levels of switching. The first switching level uses 72-way crossbar chips, supporting up to 72 Transputers, and the second connects SuperNodes together.

Parsytec - The Parsytec SuperCluster [Parsytec88] series of machines consists of a 16-node cluster with a 96 by 96 communications matrix. This matrix provides full connectivity between all nodes within a cluster, and the remaining 32 links interconnect with four further cluster boards within the SuperCluster and also to further SuperClusters. The largest machine installed by Parsytec is a 400 node machine.

Parsytec has also recently announced a new range of machines, the GC [Parsytec91], which will use the T9000 product. Other manufacturers such as, Meiko, have incorporated more powerful processors (e.g. the Intel i860) into their machines but continue to use Transputers to provide a communications network.

NCube

The Ncube range of machines originated from Caltech, where a prototype hypercube machine was built - the Cosmic cube [Seitz85]. Two ranges of commercial machines have so far been produced. The most recent, the Ncube/2, contains up to 8192 processors arranged in a 13-dimensional hypercube. The processors are of a custom built design containing a 64-bit CPU with floating point unit, 14 serial DMA channel pairs and routing hardware. Between 1Mbyte and 64Mbytes of memory can be attached to each processor. A high integration is achieved through the use of a small number of components, with 64 nodes implemented on a single board. The hypercube network uses the serial-DMA channels operating at a rate of 4.44Mbits/s.

The Intel iWarp

The Intel iWarp was produced from a joint project between Intel and Carnegie Mellon University [Peterson91]. Each processor is a custom designed 32-bit RISC with floating point circuitry operating at 20MHz. Four-way connectivity is provided with on-chip communication support using worm-hole routing. Each iWarp processor claims to achieve 100 MOPS by exploiting the parallelism available within the processor through the use of a 96-bit instruction word. The communication takes place across eight byte-wide buses (four input and four output) providing a total communication bandwidth from each node of 320Mbytes/s. On chip buffering provides 20 logical data-paths between processors which are time shared over the four physical communication channels. This allows connections to be made between non-adjacent processors without affecting intermediate processors.

The Connection Machine 5 (CM5)

The CM5 is the latest range of machines from Thinking Machines Corporation [TMC92]. It is an MIMD machine, with support for SIMD programming, and represents a departure from the CM1 and CM2 SIMD range of machines. There are three types of node within the CM5: a processing node; a control node; and an I/O node. Each processing node contains a SPARC RISC processor, with up to 32 Mbytes of memory partitioned across four optional vector-processing units. Each processing node can deliver a peak of 128 Mflops (double precision).

A control node is similar to a processing node, but lacks the vector units, and includes added I/O connections. Each control node is responsible for a partition within the whole machine and one or more control nodes may exist within a machine. All processing nodes are interconnected by two networks, a data network for message passing and a control network which supports data parallel operation.

The CM5 was one of the first machines to boast scalable performance up to Teraflops. A total of 16K nodes can be used within a CM5 giving a 2 Teraflop performance. The cost of a machine of this size is currently estimated at $300 million and has not yet been built. It remains to be seen whether a machine with this number of processors could be reliably built and sustain the claimed performance.

2.4 Summary

The options available for the memory structures, inter-processor connection schemes and control structures for parallel machines have been described and compared. The ideal P-RAM model of computation is an infeasible option for a parallel machine, except when only a few processors are required, in which case a shared memory architecture with cache coherency support can be used. As the number of processors increases, a distributed memory architecture is required. The processor interconnection schemes have shown that there is a trade off between the complexity of their implementation and the time taken for communication between any two nodes.

The survey of existing parallel architectures has illustrated that two main approaches are commonly taken in the design of a parallel machine, namely that of SIMD and MIMD. Each type of architecture has its own advantages.

The SIMD architecture can be easily implemented in VLSI, requiring only one functional unit to be replicated. It requires only one instruction stream, thus making it easy to program. However, the same single instruction stream can prove to be the bottleneck in applications which do not exhibit a high degree of data parallelism, leading to poor performance. Commonly, early SIMD machines such as the MPP, DAP and CLIP used single-bit PEs arranged in two-dimensional meshes. More recently, the available increase in VLSI density has been used to increase the functionality of each PE and increase the network topology. For example, the PEs in the MasPar MP-1 consist of 4-bit ALUs, with floating point support circuitry, and communicate via a nearest neighbour network or a 3-stage routing network.

The MIMD architectures are usually constructed from the latest high performance uniprocessor available, and built in a Multiple-SISD fashion. This can be less efficient than SIMD processors, with the replication of instruction paths and memories taking place. For example, the Stanford DASH uses the MIPS R3000 processor and the CM5 the SPARC processor. However, sometimes custom designed processors, such as in the Ncube/2 series, are used. The Transputer was the first processor to incorporate communication support for parallel processing on chip and has spawned a number of machines. Communications are performed bit-serially in the Transputer and Ncube machines, where as others, such as the iWarp, allow byte-wide communication with increased bandwidths. The inter-processor communication networks commonly form hyper-cubes (e.g. Ncube) or can be dynamically reconfigured to match the processing requirements (e.g. the Transputer machines from Parsytec and Parsys).

In the following chapter, the computational requirements for image analysis applications are examined, and the formulation of image analysis architectures discussed. It will be seen that such an application can benefit from both SIMD (for data parallel operations) and MIMD architectures (for control parallel operations).

Chapter 3

Architectural Options for Image Analysis

3.1 Introduction

In this chapter, the architectural options available for the operation of image analysis are described. These options differ from those for a general purpose parallel architecture, described in Chapter 2, in that several levels of processing can arise. Architectures which combine several levels of processors, or those that consist of a number of types of processor, might be suitable to this processing.

Image analysis, sometimes referred to as Image Understanding or computer vision, is concerned with the extraction of high level information from the contents of an image. An image analysis architecture needs to efficiently match both the processing and the data on which it is performed. Image analysis is commonly divided into several processing levels, each with its own representation of an image [Weems91, Levialdi88, Duff88]. At the lowest level, the image is a two-dimensional data array, initially containing intensity values, and at the highest level can be a relational graph containing the symbolic information about the image. Each level contains different data types and requires different forms of processing.

The different levels of processing required are suited to different processor types. Typically, the low-level, or iconic processing, is best suited to SIMD processors. High-level, or symbolic processing, is best suited to MIMD processors [Choauldry91]. Architectures which combine SIMD and MIMD processors have been proposed for use in image analysis.

The contents of this chapter are as follows: image analysis processing is described in Section 3.2; the processor requirements for the different levels of processing are discussed in Section 3.3; architectural options for image analysis are described in Section 3.4, and are illustrated through examples of current experimental machines.

3.2 Image Analysis Processing

Image analysis is concerned with the transformation of information contained within an image, from a spatial intensity representation, into a meaningful descriptive form. The resulting description can then be used for high level interpretations of the world being viewed, for example in intelligent autonomous vehicle environments.

An example of an image analysis processing flow is shown in Figure 3.1. This is a common processing flow used in many applications [WSTL90, Schalkoff89, Cantoni86]. The early processing steps typically identifies regions of interest within the input image which then require further (localised) processing. In the processing flow shown, the object detection phase can result in tentative object locations, forming regions of interest, which are further processed in an object dependent way, e.g. to segment each detected object.

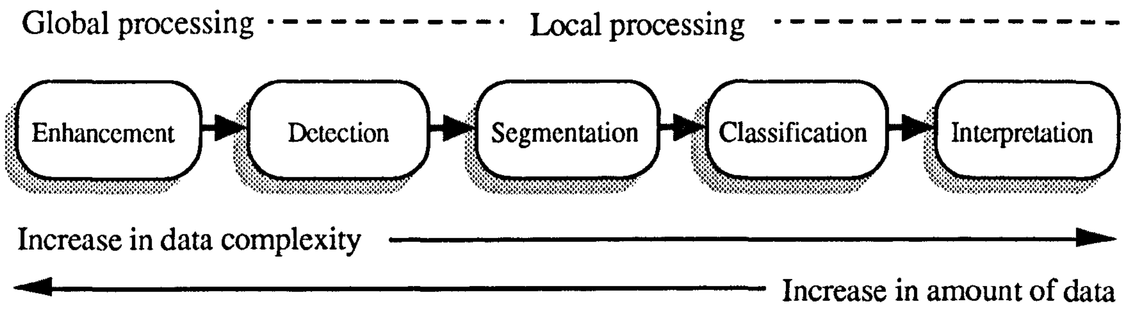


Figure 3.1 - Example image analysis processing flow.

The size of the data set during this processing flow generally decreases [Cantoni86] from image data arrays to lists of features and then to further representations of the contents of the image. However, the complexity of the data generally increases, from image data consisting of 8-bit words to linked lists, or arrays, of features and through to symbolic relationships.

Two general approaches can be taken within an image analysis system - namely those of top-down and bottom-up approaches [Ballard82]. A bottom-up, or data-directed approach, uses no a-priori knowledge, extracting low-level features from the input images which are then processed at increasingly higher levels. Alternatively a top-down approach uses a high-level model of hypothesised information which may be contained within the input image. This is processed at increasingly lower levels, down to the raw input data, looking for evidence to support initial hypothesis at each stage.

Marr [Marr82] was one of the first to propose a computational approach to computer vision. He considered three representations of images to be important. These are the Primal sketch, the $2\frac{1}{2}$-D sketch and the 3-D model representation. The Primal sketch makes the information contained with an image (and its distribution) explicit, e.g. zero-crossings, blobs, edge segments, groups and boundaries. The $2\frac{1}{2}$-D sketch is a viewer centred representation of the image, containing information about surface orientation, depth and their discontinuities. The 3-D sketch uses the $2\frac{1}{2}$-D sketch to produce a three dimensional, object centred, representation dealing with volumetric, skeletal and surface measures.

Image analysis thus encompasses many techniques on different representations of images. It includes the operations of image processing, pattern recognition, model-based vision and higher level relational approaches which have a strong link with Artificial Intelligence. The low-level operations use image-based information, whereas the high-level operations use non-image related knowledge, either extracted from the images or matched to existing models.

Many techniques are being developed for image analysis, including operations for region segmentation, motion interpretation, the use of colour and range information, shape from shading, structure from motion, and multi-sensor fusion including stereopsis [Suetens92, Nevatia82]. Such a broad range of techniques cannot be individually reviewed within the context of this thesis. However, in reviewing the requirements for parallel processing, the structure of the data and the form of the processing to be performed will be discussed.

The different levels of processing associated with image analysis have been commonly classified into three broad levels [Weems91, Leviadi88, Duff88, Hwang91, Maresca88, Levitan87, Tanimoto85]. These are - iconic, where the data remains in the form of images (2D data arrays); intermediate, consisting of representations of the images in non-iconic form; and symbolic, where the data consists of descriptions or abstract representations of the images.

The three levels of processing in image analysis are shown schematically in Figure 3.2. The arrows in Figure 3.2 indicate the interactions between the levels. The downward arrows indicate that further processing can take place between the levels, dependent upon earlier processing, consistent with a top-down approach.

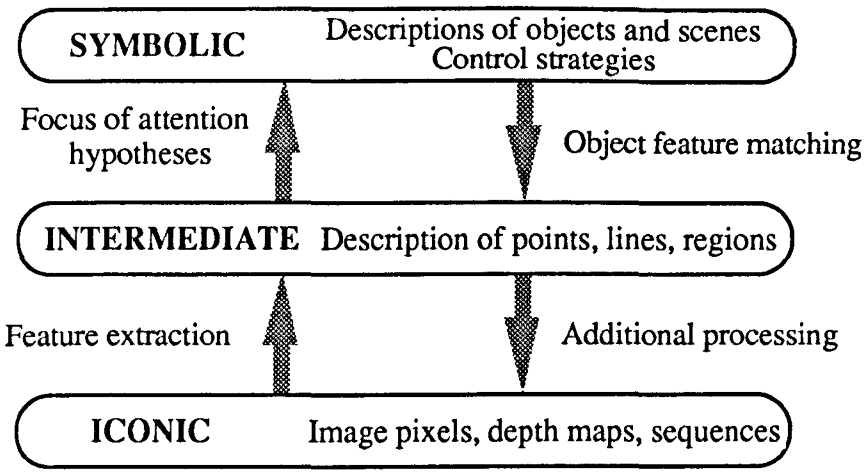

SYMBOLIC
Descriptions of objects and scenes
Control strategies
Focus of attention hypotheses
Object feature matching
INTERMEDIATE
Description of points, lines, regions
Feature extraction
Additional processing
ICONIC
Image pixels, depth maps, sequences

Figure 3.2 - The levels of processing within image analysis.

3.2.1 Iconic processing

This level of processing can be characterised by operations which take images as input and produce images as output. These operations commonly form the early stages of processing in an image analysis system and are often referred to as low-level operations. An example is a local neighbourhood operation, where each output pixel is a function of the pixels within a local window centred about itself.

The data in the iconic processing stage remains in the form of a two-dimensional array. Commonly, the input images have 256 grey-levels (8-bits precision) per pixel which are expanded upon by summation or multiplication operations. Other iconic representations include:

- Labelled images, in terms of regions, edges, corners, texture etc.
- Transforms, including Fourier, Hough or feature-spaces
- Multi-resolution, consisting of images containing different spatial resolutions
- Depth maps, from binocular vision systems or from range sensors
- Motion flow-fields, from image sequences

Rosenfeld [Rosenfeld82] considers four classes of operations for low-level image analysis. These are distinguished by their structure :

- Point operations, only a single pixel is involved in the computation. An example is a simple threshold operation resulting in a binary image.
- Neighbourhood operations, which require a local window of pixels for the calculation of each output pixel. Many operations in low-level image analysis have this form including convolutions for enhancement and edge detection, rank order filters, and local histogram operations.
- Global operations, where each output is a function of most or all of the input image. This includes transformation operations such as the Fourier Transform and the Hough transform.
- Statistical operations, e.g. histogram generation. Although this is strictly classed as an intermediate operation, it can be used to produce an iconic output in operations such as histogram equalisation.

Low-level iconic processing is usually uniform across the whole image. Little or no information is available about what parts of the image are of interest and thus will require specific processing.

3.2.2 Intermediate processing

The intermediate level is not as well defined as the iconic level. It is often thought of as a transitionary level, taking image data as input and producing non-image data as output [Tanimoto85]. It is used to interface sensor information and information about the world. Thus, the intermediate level forms an interface between the iconic and the symbolic processing levels.

The processing involved at this level often takes the form of extracting features from images, through grouping, splitting and labelling to form symbolic representations.

The resulting symbolic representations should also be integrated so that the different representations can be related, e.g. a region is related to its boundary lines.

Most symbolic representations can be built from three types of image features: points, lines, and regions [Tanimoto85]. Points in an image can represent end-points of lines, maximum or minimum intensity locations, centres of regions, or corners. Lines can represent object boundaries - the position of changes of regions e.g. texture. Regions are a collated set of pixels, possibly representing an area of the same texture, or the output from a segmentation process.

Weems [Weems91] considers the intermediate level to also include the processes of model transformations, and model matching. The model transformations (such as rotation, scaling and translation) take place on the extracted image features and are matched to existing models.

3.2.3 Symbolic processing

Symbolic level processing aims to generate descriptions of objects in the 3-dimensional world, from the image data through the use of intermediate representations. Further processing may then take place to interpret the symbol representation, such as that for manoeuvre planning in a robotic environment, or for resource allocation.

The form of the processing that should take place in this high level stage is not well understood [Duff88]. Operations that have been carried out in the symbolic level have, in general, been specific to the problems in which they have been applied.

Weems [Weems91] considers the symbolic processing to also be responsible for maintaining the knowledge of the world. This can be achieved through the matching of models, the production of hypotheses about the contents of the image, comparing the hypothesis and resolving any conflicts through further processing.

3.3 Computational Requirements for Image Analysis

The structure of the data, contained within the three processing levels of an image analysis application, exhibit different degrees of parallelism and complexity. The structure of the data, and the operations performed, results in the possible utilisation of parallel processors. The efficient matching (or mapping) of the data and operations onto a parallel processor affects the overall utilisation and system performance. When real-time operation is required, the choice of operations performed is often restricted by the performance of the system.

Image analysis requires a large amount of processing and data throughput. For instance, the number of multiply-accumulate operations per second required for the operation of a single spatial filter, on an image in real-time, is :

$$m^2 * N^2 * \frac{1}{T}$$

where m^2 is the number of elements in the filter, N^2 is the number of pixels in the image and T is the frame time (in seconds). Even for a modest filter of size 7x7, on an image of size 512x512, a total of $321*10^6$ multiply-accumulate operations are required per second with a typical frame time of 40ms. Uni-processors are inadequate even for a single task of this complexity.

The structure and types of data for image analysis operations are described below. Each is considered within the three levels of processing outlined in Section 3.2.

Iconic level requirements

The processing at the iconic level involves the processing of regular, 2-dimensional, arrays of data. The high degree of implicit data parallelism may be utilised so long as the operations to be performed can take place on all the elements of the array in parallel. Such a situation arises in early image analysis operations. However, in later operations, local areas or regions of interest may be required to be processed independently. The main features of iconic processing are :

- The operations typically involve the use of local data values, as with spatial filters, requiring simple (local) patterns of data addressing.
- The word size of the data is typically low, starting at 8-bits from an image sensor, and increasing as a result of processing.

Exceptions to these include transformations, e.g. the FFT and Hough transforms, which require global communications and can also result in enlarged word lengths with the need for floating point arithmetic [Rosenfeld88].

The operations involved in the iconic processing level are suited to the SIMD processing paradigm when the operation is global. Local region processing is also suited to SIMD. However, poor utilisation of an SIMD array will result if the region sizes are less than the size of the processor array.

Intermediate level requirements

The intermediate level requires the extraction of features from the iconic data while retaining spatial relationships. The operations which are performed at this level on the extracted features will typically involve a large area of the image, e.g. for the collation of edges. Thus more complicated communications are required than for the iconic level. The extraction of the features from the image data can be performed in parallel if multiple data-paths exist between the iconic and intermediate processing levels.

The structure of the data that is processed in this level typically consists of lists, or arrays of features. The processing is usually irregular and data dependent [Choudhary92], dealing with the collation of features to build a symbolic representation. The extraction and collation of the data is local to image regions and can thus be performed in parallel. These operations are suited to the MIMD parallel processing paradigm.

Symbolic level requirements

Since the processing in this level is object/scene dependent, the processor requirements cannot be accurately determined. The techniques that can be used at this level include Artificial Intelligence and database interrogation for model matching. However, it is generally accepted that the processors in this layer should be as flexible and general purpose as possible [Duff88]. The parallelism contained within the operations at this level is also unclear, but will probably require the dynamic scheduling of processes [Choudhary92].

The flexibility required at this level is suited to the MIMD processor paradigm. The unstructured control, communications and lower granularity, in an MIMD processor array in comparison to an SIMD array, provides a better match to the required flexibility.

It is clear from the above discussion that the three levels of iconic, intermediate and symbolic processing have differing computational requirements :

- the iconic processing is best suited to an SIMD array
- the symbolic processing is best suited to an MIMD array
- the intermediate level forms the interface between the iconic and symbolic levels in bottom-up processing, and vica-versa for top-down processing

The apparent conflict in the processing requirements for image analysis result in a question of which type of processor should be used?

Several proposed image analysis systems have recognised the need for a dual-paradigm architecture. A number of experimental systems use separate processor levels for the image analysis processing levels described above. The architectural options that have been applied in experimental image analysis systems are described in Section 3.4 below.

3.4 Image Analysis Architectures

There have been several architectures proposed which consist of a number of processor arrays, each applied to a different level of processing within image analysis. In 1984, Reeves [Reeves84] suggested a system which combines an SIMD array with an MIMD array, where the iconic processing would be mapped to the SIMD array and the symbolic processing mapped to the MIMD array. This general approach has also been suggested by others and forms the basis of many proposed image analysis systems [Siegel81, Nudd89, Segal90, Hwang91].

Many options arise in the construction of an architecture from several processor arrays, including which processors to use. Also the connectivity, and control of the processors within, and between, the arrays is an important factor. The options used within experimental image analysis systems have conformed to the following four main categories :

1) Dedicated hardware
   - those that use specific purpose components for certain operations
2) Homogeneous pyramids -
   - those that use a single processor type within several processor arrays
3) Reconfigurable
   - those that can change their control structure dynamically
4) Heterogeneous arrays
   - those that use different processor types in multiple processor arrays

3.4.1 Dedicated hardware

Dedicated chips for specific operations typically offer increased performance over a general purpose programmable system. This is due to the hardware being dedicated to the computation of a specific operation. Available operations, that can be performed on single chips include the FFT and Hough Transforms, FIR filters, histograms, rank order filters and convolutions [LSI89, Slorach88]. However, systems built solely from dedicated components are limited to performing specific operations.

A system, which incorporates a range of dedicated chips with an array of general purpose MIMD processors, has been proposed and built at ETH in Switzerland. This system is called the SYDAMA-2 [Stokar92]. Special attention was given to the design of a bus interconnection scheme, between the dedicated processing chips, to enable real-time operation, both in terms of the required computation and bandwidth. A schematic of the SYDAMA-2 is shown in Figure 3.3.

The dedicated processors are connected together in a ring topology, via a pipelined ring bus. This bus consists of 48 communication lines which are time multiplexed to provide 96 logical lines. This enables twelve 8-bit video data streams to be transmitted concurrently. Each stage within this bus consists of a 64x64 cross-bar switch, enabling the transmitted data from the previous stage in the ring to be transmitted to the next stage, or used within the processing of the current stage. The input and output video data are connected directly to this bus.

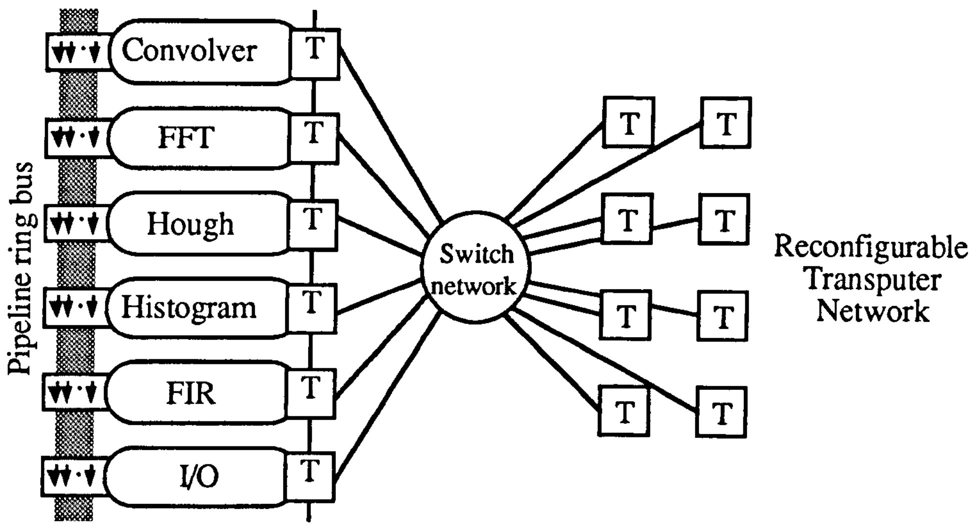


Figure 3.3 - The configuration of the SYDAMA-2 system.

Each dedicated processor is under the control of a single Transputer. Further Transputers are connected to form a dynamically reconfigurable MIMD network, available for general purpose computing. The MIMD network enables an operation to be performed on the image data that could not be performed using the dedicated hardware components. However, such operations are limited by the computation and communication capabilities of the Transputer network.

3.4.2 Homogeneous pyramids

A number of architectures have been suggested which form homogeneous pyramids of processors. These architectures consist of multiple arrays (levels) of a single type of processing element. Each level contains a different number of processors, the largest at the bottom, with progressively smaller arrays towards the top. These architectures therefore form a pyramid of processing elements. The basic architecture of a homogeneous pyramid is shown in Figure 3.4.

Homogeneous pyramids include the GAM pyramid [Schaefer87], the PAPIA [Cantoni87], and the SPHINX [Clermont87]. Much of the algorithmic work on multiple-layer pyramids has been carried out by Tanimoto [Tanimoto86, Tanimoto87]. The design of these pyramids have much in common. They are all designed for use as a hierarchical image analysis system. They all use bit-serial processors within their multiple levels. They all have connections between processors within the same pyramid levels and to levels above and below that level. Typically a quadtree structure is formed, i.e. each processor is connected to four processors in the level below and to one processor in the level above.

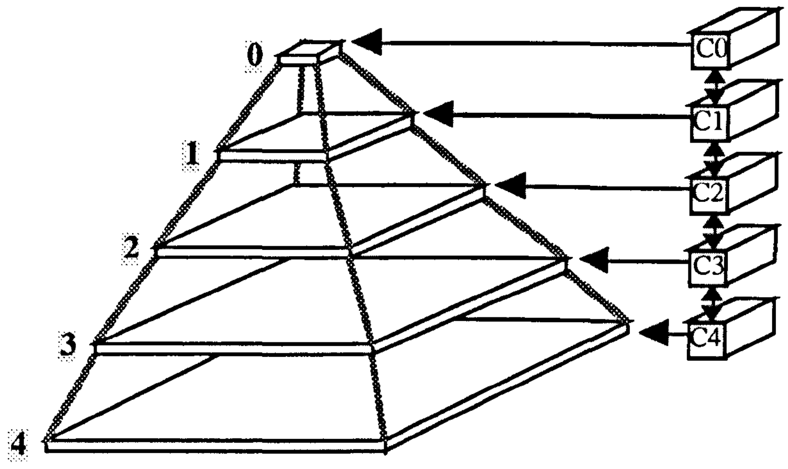


Figure 3.4 - The structure of a multiple-layer homogeneous pyramid architecture.

The GAM pyramid has five processing levels, with each level a factor of four smaller than the one beneath it. A total of 341 processors were used, implemented by the available Massively Parallel Processor (MPP) PE. All processors operate under the control of a single controller, i.e. operate as an SIMD machine except for communications between levels. Here, local autonomy enables one level to perform a 'send' operation while another performs a 'receive' operation. A camera interface provides input 16x16 images at the lowest array level.

The PAPIA pyramid was proposed as an eight level machine, containing a total of 21,845 processors, with the bottom level consisting of an array of 128x128 processors. The bit-serial processing element was custom designed, and a proposed second generation IC design would contain a mini-pyramid with 3 levels. Three controllers where included in the design: one to control the upper four levels; one for the 5th and 6th levels; and the third for the lower two levels. Thus, a full sized PAPIA machine forms a Multiple-SIMD architecture.

The SPHINX pyramid has a very similar form to that of the PAPIA pyramid. It uses a custom designed processing element, but has individual controllers for each processor level. The synchronisation of these controllers, for communication between levels, is discussed by Clermont [Clermont87].

All of these pyramids form multiple levels of SIMD processors, either controlled on a per level basis or, in the case of the GAM pyramid, under the common control of a single controller. Although these architectures are very well suited to multi-resolution algorithms, they are not suited to the mixture of iconic and symbolic operations involved in image analysis. Cantoni [Cantoni87] envisaged that low level operations would be performed on the PAPIA pyramid, and that high level vision operations would be relegated to a host computer. This however, makes the assumption that little computation is required in the high level operations.

3.4.3 Reconfigurable

Some architectures enable the dynamic reconfiguration of their control structure between SIMD and MIMD modes of operation. An example is the PArtitionable Simd/Mimd (PASM) machine [Seigel81]. PASM contains a resource of up to 1024 processors which can be partitioned to operate as many independent SIMD and MIMD machines as required for an application. The reconfiguration can be performed dynamically. A prototype, the PASM-1, has been constructed containing a total of 30 Motorola MC68000-series processors [Kuehn85].

PASM consists of four main components; the Parallel Computation Unit (PCU), a set of Micro-Controllers (MCs), a memory management system and a system controller. The PCU contains the processors, each with their own memory, and an interconnection network. Each processor may take its instruction stream from its own memory, in MIMD mode, or from an MC, in SIMD mode. Each MC is a processor (a MC68000 in the prototype) that acts as a control unit for a group of processors in SIMD mode. Each MC is responsible for a fixed groups of processors, in the prototype each MC can control four processors.

The interconnection network between processors is capable of operating both synchronously and asynchronously. When a set of processors are operating in SIMD mode, the relevant part of the network operates synchronously supporting nearest neighbour communications. When the processors operate in MIMD mode, the relevant parts of the network operate asynchronously, supporting point to point communications. However, this reconfiguration capability results in complex control and communication overheads.

3.4.4 Heterogeneous architectures

Heterogeneous architectures contain two, or more, types of processor. Typically, individual arrays are formed from each processor type and interconnected to form a multiple-level heterogeneous architecture. An application using an heterogeneous architecture can be split into constituent components such that the components best suited to one type of processor can be mapped onto that processor. Thus, in an image analysis application, the different levels of processing can each be performed on the most suitable processor array.

Several heterogeneous architectures have been proposed for use in image analysis, including the Display Array (DisArray) from Oxford University [Page89], the Orthogonal Multiprocessor with an Enhanced Mesh by Hwang at University of Southern California [Hwang91], and the Image Understanding Architecture (IUA) a joint project by Hughes Research Laboratories and the University of Massachusetts [Weems89].

The DisArray combines an array of SIMD processors with an array of MIMD processors. The design of the DisArray resulted from a study of vision and computer graphic operations. Low level operations are performed on the SIMD array and higher-level operations are performed on the MIMD array. The SIMD array consists of 16x16 bit-serial processing elements. The MIMD array contains ten T414 Transputers, inter-connected using a crossbar switch, with a further thirty processors arranged in a fixed 5x6 grid.

The SIMD array has a dedicated controller, based on a single T800 Transputer, having the benefits associated with the Transputer software support. The controller is connected to the crossbar switch and thus enables data transfers to take place between the SIMD and MIMD arrays. However, the available bandwidth is limited to a single Transputer link, at a maximum of 20Mbits/s. Although this system is small, some graphic and vision algorithms have exploited the individual capabilities of both the SIMD and MIMD arrays.

The Orthogonal Multiprocessor with an Enhanced Mesh has recently been proposed by Hwang [Hwang91]. It has the potential to support both low and high level image analysis tasks through the use of an SIMD and an MIMD processor array. The MIMD array is termed the Orthogonal Multiprocessor (OMP) and the SIMD array is termed the Enhanced Mesh (EM). Interaction between the two arrays is provided by a common shared memory. The OMP and the EM are both shown in Figure 3.5 along with the shared memory.

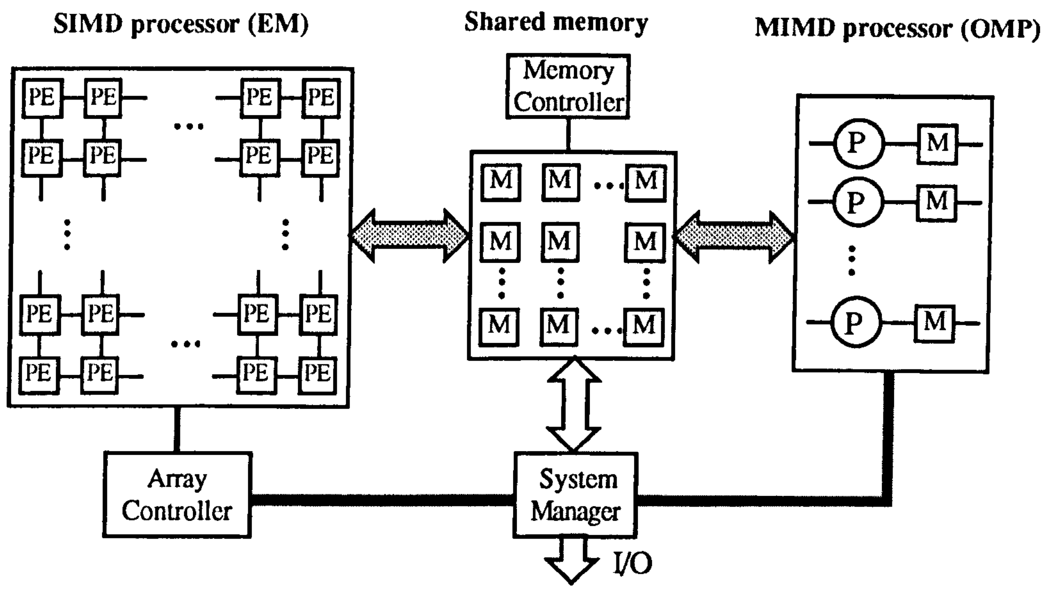


Figure 3.5 - The Orthogonal Multi-Processor with an Enhanced mesh.

The shared memory forms an integral part of the architecture. It enables data transfers to be made transparently between the SIMD and MIMD arrays in either direction. The shared memory consists of n^2 memory modules. Each module consists of three memory sub-modules, interconnected via a 3x3 crossbar switch, and interfaces to the SIMD array, the MIMD array, and the system manager for I/O. Each memory module contains the memory for a total of k_1k_2 SIMD processing elements, and part of the memory for a single OMP (MIMD) processor. The crossbar switch enables SIMD, MIMD, and I/O memory operations to take place alternatively on data contained within the three memory sub-modules.

The SIMD array consists of NxN processing elements with four way nearest neighbour connectivity. The processors are controlled via a single array controller and can communicate to the MIMD array via the shared memory. The MIMD array is based upon the OMP multiprocessor [Hwang90]. It consists of n Intel i860 processors. Each processor is connected to a pair of dedicated busses, running horizontally and vertically through a square grid of the n^2 memory modules. Thus, each processor can access a total of 2n-1 memory modules. Local memory connected to each OMP processor stores local data and code. All OMP processors are connected to a common communications bus which can be used for interprocessor communication and synchronisation.

A range of low and high level image analysis tasks have been shown to be efficiently executed on the combined OMP and EM [Hwang91]. A prototype of the combined OMP and EM has not been constructed, but Hwang suggests that a suitable size would be 64x64 SIMD PEs, 16x16 MIMD processors and 16x16 shared memory modules. However, the single controller on the SIMD array can result with in-efficient local processing for image regions.

The Image Understanding Architecture (IUA) also consists of several processor arrays. It has been designed to match the processing requirements of image analysis [Weems89]. It consists of three layers of processors: one for low level processing; one for the intermediate level; and one for the high (symbolic) level. These layers form a pyramid of processors. However, unlike the homogeneous pyramids described above, the three layers form a heterogeneous pyramid. Interleaving the processor layers is shared memory, enabling communications between the layers to take place. The IUA is shown in Figure 3.6.

The low level array is implemented using a custom designed, bit-serial Content Addressable Array Parallel Processor (CAAPP). The CAAPP is used to form a 512x512 array, intended to perform low-level image processing operations. The mechanisms of a global Some/None and COUNT operations are included in this array to aid associative operations. The CAAPP array operates as a conventional SIMD array, under the control of a single Array Control Unit (ACU). Each CAAPP processor has access to 32Kbits of external memory, which is also shared by the intermediate level.

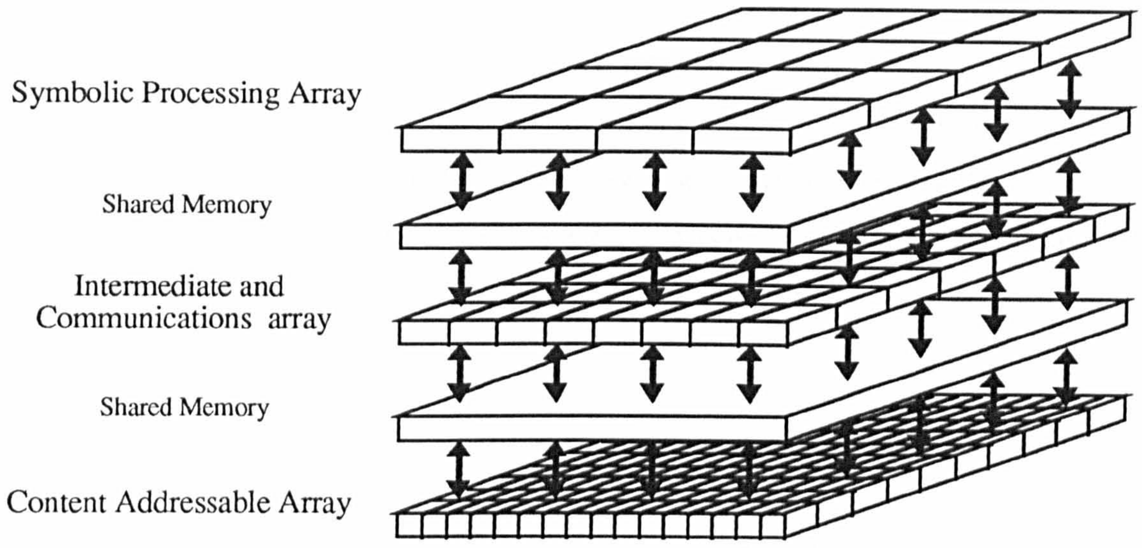

Symbolic Processing Array
Shared Memory
Intermediate and Communications array
Shared Memory
Content Addressable Array

Figure 3.6 - The Image Understanding Architecture.

The SIMD array also incorporates a reconfigurable bus termed the Coterie network by Weems [Weems89] and the Gated Connection Network (GCN) by Hughes [Shu88]. It consists of an array of eight transmission gates per PE. It enables a broadcast bus to be set up between sets of processors, which need not be adjacent, using the dynamic switching of the transmission gates. The performance of image analysis operations such as maximum / minimum calculations, labelling connected components, and minimum spanning trees are enhanced as a result of this network [Shu88]. The calculation of the minimum spanning tree occurs in applications such as the unwrapping of fringe interferograms [Judge92].

The Intermediate level consists of an array of 64x64 Intermediate Communications Associative Processors (ICAP). Each ICAP consists of a DSP processor with local and shared memories. The ICAP processors are interconnected, enabling intermediate grouping operations to be performed. The ICAP has two modes of operation, either synchronous, where all ICAPs operate on the same instruction stream, or asynchronously, operating with independent instruction streams. Control for the ICAP processors is provided by the ACU when operating synchronously, or by the symbolic level processors when operating asynchronously.

The Symbolic Processor Array (SPA) contains general purpose MIMD processors, with co-processor support for symbolic operations. The SPA communicates to the ICAP processors through the second layer of shared memory. The detailed specification for the SPA has not yet been carried out, although it is envisaged that it will consist of 64 processors running a LISP based black-board system [Weems89]. Using the Motorola M68020 as a single symbolic processor, a prototype IUA has been constructed which consists of a 1/64th of the whole machine.

3.5 Summary

The operations required for image analysis conform to three levels of processing, namely: iconic, intermediate, and symbolic. These three levels have different computational requirements, resulting in a system requiring multiple-levels of processors. Each processor level is dedicated to a part of the overall image analysis task. The iconic level requires SIMD processing, which can be global across the whole image, or local to specific regions within the image. The intermediate and symbolic levels require the flexibility of MIMD processing.

Several architectures have been proposed for use in image analysis. The SYDAMA-2 uses dedicated components which are suited only to the calculation of specific operations. The proposed homogeneous pyramids of SPHINX, PAPIA, and GAM contain multiple layers of processors. These are suited to multiple-resolution processing and not to combined iconic and symbolic processing. The dual processing paradigms of PASM, the combined OMP / EM, and the IUA contain the capability of the dual processing required for image analysis.

However, the IUA and the combined OMP / EM, enable only global SIMD processing with MIMD processing. Thus, local SIMD processing can not be performed efficiently on either of these machines. PASM promises the capability of reconfigurable SIMD /

MIMD processing including local SIMD processing. However, the reconfigurability results in complex control and communication systems.

In the next chapter the design of a dual-paradigm architecture is described, which encompasses both local and global SIMD processing, with MIMD processing. The machine is called the Warwick Pyramid Machine (WPM), and is designed for use in image analysis applications. Its design and implementation is described in Chapter 4. The mapping of data onto the WPM is described in Chapter 5, and a study of its use for tracking operations is contained in Chapter 6. Finally load-balancing considerations, for efficient utilisation of the WPM, are discussed in Chapter 7.

Chapter 4

The Warwick Pyramid Machine

4.1 Introduction

This chapter details the design and implementation of the Warwick Pyramid Machine (WPM) [Nudd89, Nudd91, Nudd92a]. The WPM incorporates several types of processors for the simultaneous processing of iconic and symbolic data types. It consists of several processing levels : at the lowest level is a massively parallel SIMD array; the next level consists of an array of controllers, each of which controls a square sub-section of the SIMD array; and an array of MIMD processors. The apex of the pyramid consists of a single Host node such as a SUN workstation.

The partitioning of the SIMD array, into smaller Multiple-SIMD arrays each under local autonomous control, enables the WPM to process many data sub-regions simultaneously. This approach differs from other architectures, such as the IUA [Weems89] and the OMP with EM [Hwang91] (see Section 3.4.4), although they also combine SIMD and MIMD processor arrays. The SIMD arrays in these architectures contain little or no autonomous control, except for activity controlled operations.

The description of the WPM within this chapter is divided in the following way. In Section 4.2, an overview of the WPM is given. In Section 4.3, the details of the prototype WPM are given. The prototype implementation is described in terms of its SIMD array, its MIMD array, and its controller array. A part of each of these arrays forms a modular and scalable unit (a Cluster), and this is also described. The programming issues associated with the WPM are described in Section 4.4. Finally, performance figures for each of the levels within the prototype WPM are given in Section 4.5.

4.2 Overview of the WPM

The WPM contains both MIMD and Multiple-SIMD (M-SIMD) processor arrays. The machine can be viewed as forming a pyramid of processors. At its base is a massively parallel two-dimensional SIMD array connected to a camera ouput, partitioned into smaller M-SIMD arrays, and capable of processing iconic data. Further up is a smaller two-dimensional array of coarser MIMD processors which are aimed at the processing of symbolic data. At the pyramids apex is a single host workstation providing both a user interface, to the lower levels, and storage facilities. These processor levels are shown in Figure 4.1.

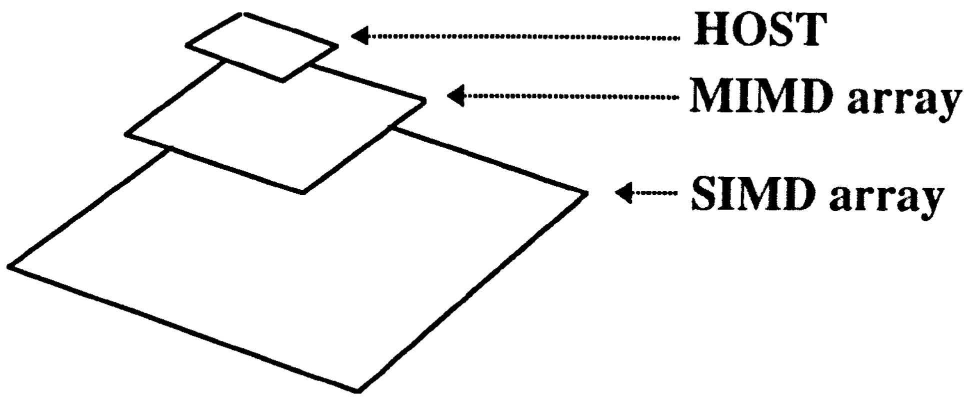


Figure 4.1 - The Warwick Pyramid Machine.

The WPM is configured such that each of the M-SIMD arrays has its own controller and each is associated with a single MIMD processor. This forms a modular and scalable functional unit which is termed a Cluster. The spatial arrangement of processors between levels is preserved, such that the top-left MIMD processor is associated with the top-left M-SIMD array. The configuration of a single Cluster is shown schematically in Figure 4.2.

Clusters are four-way interconnected at each of their three levels: at the SIMD level for synchronous communications; at the controller level for local instruction synchronisation; and at the MIMD level for asynchronous message passing communications. The interconnections at the MIMD level and the controller level are shown by the arrows in Figure 4.2.

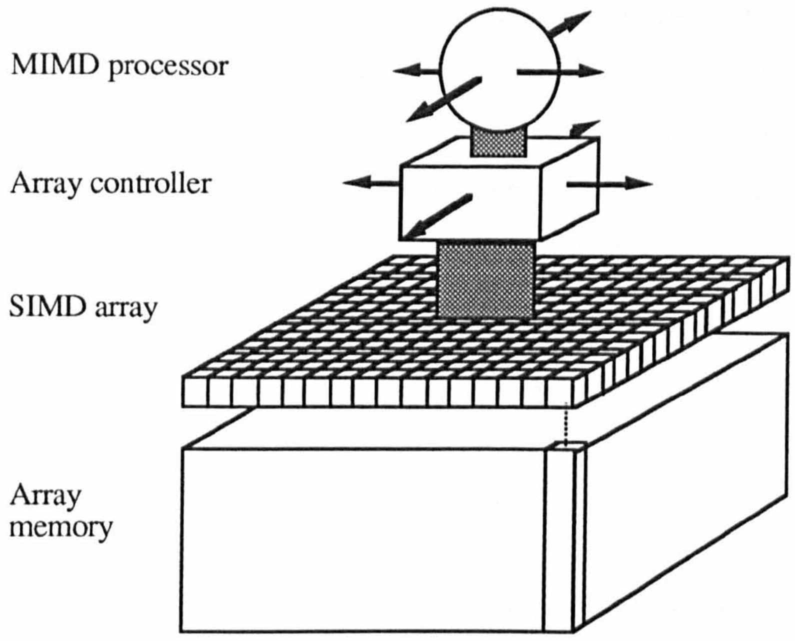


Figure 4.2 - A Cluster, the modular component of the WPM.

Shared memory exists between the MIMD processor and SIMD array within a Cluster, enabling communications between the two to take place. This interconnection scheme has a greater bandwidth over an architecture which simply interconnects an SIMD and MIMD array along a single axis. The shared memory is accessible by both the MIMD processor and the controller. The controller can pass data to the SIMD array, from the shared memory using broadcast operations, or data can be placed in the shared memory from the SIMD array, using its associate response mechanisms.

The SIMD array within a Cluster has two modes of operation; either Cluster mode, or Array mode, both of which are described below.

- Cluster mode enables each SIMD array to operate autonomously, communicating only at the MIMD level. This enables image regions, within each Cluster, to be processed independently, thus overcoming the single instruction bottleneck of a conventional SIMD array. In Cluster mode, the communication boundaries of the SIMD array are wrapped around, internal to the Cluster, forming a torus network.

- Array mode enables some or all of the Clusters to be synchronised. This allows communications between adjacent Clusters to take place at both the

SIMD and MIMD levels. When all Clusters are synchronised, the whole M-SIMD array acts like a conventional SIMD array. This feature is important in shifting operations in which communications are required between each PE and their neighbour, e.g. in a filtering operation. A global filtering operation, on the whole SIMD level, requires the synchronisation of all the SIMD Cluster arrays.

In the prototype WPM, the SIMD level of a Cluster is implemented using an array of 16x16 DAP processing elements. The MIMD level of the WPM is implemented using the Transputer (T800). The controller within each Cluster has been designed for general SIMD array usage, giving flexibility for expansion, or for the functional change of the SIMD array. However, the prototype controller incorporates specific facilities to support the functional capabilities of the DAP processors.

The modular design of the WPM enables its design to be specified in terms of a single Cluster, as this contains part of each of the three processing levels. Thus, the design of the Cluster enables a multiple-Cluster configuration of the machine to be constructed. An 8x8 array of Clusters is envisaged as a full sized Warwick Pyramid Machine.

The use of available processors has enabled the quick prototyping of the WPM to take place. Although this may not have produced an optimal match between the processing levels, in terms of their computational capabilities, or in the use of current VLSI technology, it has enabled a prototype WPM to be constructed, tested and analysed for use in various applications.

4.3 Implementation of the WPM

The implementation of the prototype WPM Cluster is described below in terms of its three levels. The prototype uses an array of 16x16 DAP processing elements for the SIMD level, a single Transputer for the MIMD level, and a bit-slice SIMD controller design. The controller provides instruction sequencing for the SIMD array, 16-bit scalar functionality, and shared memory between the SIMD and MIMD arrays within a

Cluster. The interconnections of Clusters between their three levels are also detailed below and the communication performance is discussed.

4.3.1 The SIMD array

The SIMD array, within each WPM Cluster, is implemented using the AMT DAP bit-serial processing element. The heart of the present commercial DAP machine is used to implement the WPM SIMD array. A set of four ICs are used from the DAP machine, each containing an array of 8x8 PEs implemented using approximately 3,000 gate equivalents.

The DAP PEs are arranged in a 4-way connected mesh network, NEWS (North, East, West, South). Figure 4.3 shows the arrangement of 16x16 DAP PEs as used within a WPM Cluster. The enlarged view of four PEs, on the right of Figure 4.3, shows the row and column highways, running horizontally and vertically through the PEs, used in PE associative operations.

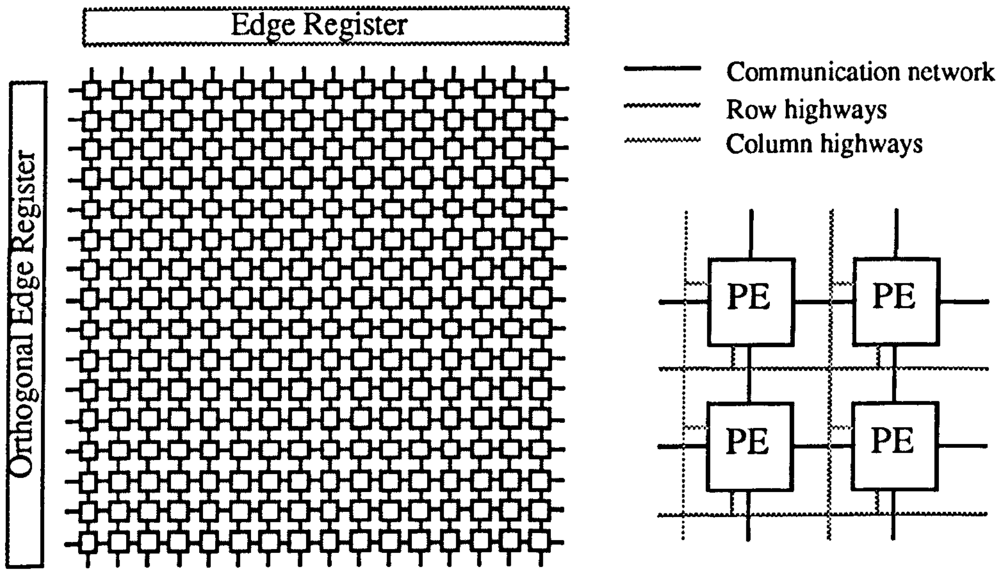


Figure 4.3 - A 16x16 DAP SIMD array, showing the detail of 4 PEs.

A 16-bit register known as the Edge Register is situated along two sides of the SIMD array. It is used to store the associative responses of the SIMD array and can also be used to broadcast values across the array. The associative response can take the form of extracting a specific row or column from the array, using the row and column highways, or it can extract the logical AND of all rows or all columns. The Edge Register can broadcast a value across the rows of the SIMD array or down the columns of the array. The value may be stored in specific rows or columns by the use of conditional store instructions.

The associative capabilities of the DAP PEs have been enhanced, within the WPM, by the addition of a 256-bit input count within each Cluster. This can count the number of bits set, one taken from each PE, to produce a 9-bit result. In addition, it provides a single bit Some/None response which indicates if any of the PEs have a bit set. Both of these responses can be used by the Cluster controller for conditional instruction sequencing.

4.3.1.1 Functionality of the DAP PE

Each DAP PE contains a full-adder with 3 inputs and 2 outputs, five single-bit registers and two multiplexors to route the inputs to, and outputs from the full-adder. A DAP PE is shown in Figure 4.4.

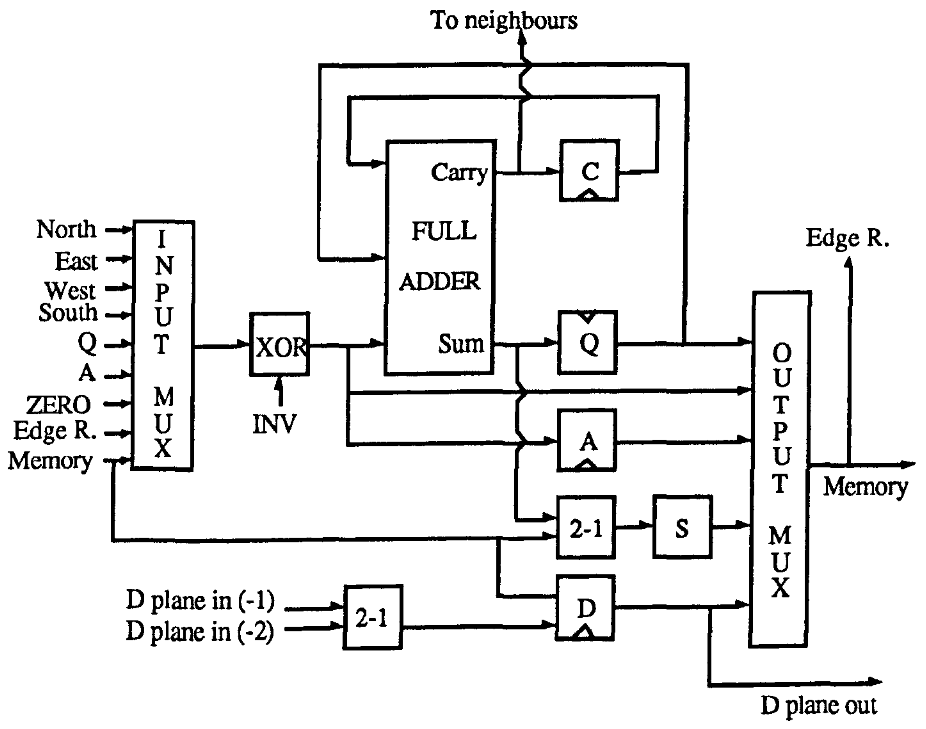


Figure 4.4 - The DAP processing element.

Each register can be viewed as a plane of registers when considering the SIMD array as a whole. The function of each PE is determined by an instruction word of 13-bits. Each DAP instruction effectively combines the states of the internal registers with a memory value. Most DAP operations take a single cycle, including a read from or write to the external memory. The performance of the DAP is limited by the bandwidth to its external memory.

The input to the full-adder can be taken from either: the PEs four nearest neighbours; the Q or A registers internal to the PE; external memory; a zero; or from the Edge Register. The output multiplexor can route data from the Q, A, or S registers, or from the input multiplexor, to external memory, or to the Edge Register using associative response operations. The S register takes its value from the memory input or from the Sum output of the full-adder dependent upon the value of the A register. This is used in conditional operations, with the A register being the activity control.

Four bits in the DAP instruction word determine the data-path through the PE (i.e. which one of the inputs is selected by each of the multiplexors). Each of the sixteen combinations of these four bits represent a different instruction group as shown in Table 4.1. These groups form the DAP assembly language instruction set - APAL [AMT88].

Each of the registers within the DAP PE have a specific use, or uses, depending upon the instruction group being executed. The C register is used to hold the carry output from the full adder, and the Q register the Sum. The registers Q, A and C can be updated during all instruction groups, except 2 and 6, using clock control lines which are part of the DAP instruction word. Shifting of a data bit, between PEs, is performed in a group 12 instruction. During this instruction, two of the inputs of the full-adder are taken from the Q register (the third is zeroed) producing the value of Q on the carry output. The carry output can be shifted in any direction to the four neighbouring PEs.

The A register acts as an accumulator in that its previous value can be ANDed with the value from the input multiplexor. Alternatively, it can take the value of the input multiplexor. It also acts as an activity control bit on a conditional store or addition instructions, in groups 10 and 11. The conditional operations are two cycle instructions. In the first cycle, a memory bit is read into the S register while also performing the addition or copy to the Q register. This is followed in the next cycle by writing either the S or Q registers to memory, using the A register as the selector.

Group	Description	Input MUX	Output MUX
0	memory ⇒ PE	memory	none
1	XOR memory with bit of Edge Register ⇒ PE	memory	none
2	memory ⇒ Edge Register	memory	input MUX
3	orthogonal Edge Register ⇒ PE	orthogonal R	none
4	Zero ⇒ PE	zero	input MUX
5	bit of Edge Register ⇒ PE	zero	none
6	Edge Register ⇒ memory (main store mode)	R	input MUX
7	Edge Register ⇒ PE	R	none
8	Q ⇒ PE/store/Edge Register	Q	Q
9	A ⇒ PE/store/Edge Register	A	input MUX
10	conditional add ⇒ store	memory	S
11	conditional write ⇒ store	zero	S
12	shift Q ⇒ nearest neighbour	neighbour	none
13	ripple add (in any direction)	neighbour	none
14/15	no-ops	-	-

Table 4.1 - The DAP instruction groups.

The D plane is independent of the normal operation of the PE, allowing autonomous shifting from south to north across the whole SIMD array. It can perform one of four operations - no shifting, shift even rows only, shift odd rows only, or shift all rows. This is designed specifically to be connected to a sensor output, allowing for possible frame interlace. The shifting takes place independently of the PEs operation, but the data from the D plane can be accessed by the PEs when the shifting is complete. The overhead of shifting in an input image into the array is minimal. The D plane has not as yet been used within the WPM.

All the instruction groups of Table 4.1 have been implemented on the DAP PEs within the WPM, except for Main Store Mode (DAP group 6 instructions). This mode allows the Edge Register to be written to the memory of a single row of the SIMD PEs. The memory is arranged in the DAP machine such that one row of PEs is connected to one set of memory chips. Hence, by making part of the memory address the row address, the memory may be selectively accessed. For the WPMs 16x16 array, four bits would be needed to specify a row address, thus limiting the addressable memory to 4kbits per PE when using a 16-bit address field. Instead, a 16-bit word is used to address a total of 64kbits of memory per PE. Individual writing to memory rows can still take place using the conditional write instructions, involving a slight overhead when compared to the DAP group 6 instructions.

4.3.1.2 The SIMD associative Count

The DAP PEs associative response capabilities, within the WPM, have been enhanced by a count device. This has a single input from each of the PEs, connected to each PEs external memory line within the 16x16 SIMD array of a Cluster. It performs the function of counting the number of inputs which are set, and outputs a 9-bit summation value. In addition, a check is made to see if any of the outputs are set, a Some/None operation, outputting a single bit result. Both outputs of this count can be used by the Cluster controller.

The count and Some/None can be used for a number of different functions. Consider the problem of determining if a feature exists within an image. The output of such processing could consist of a binary mask, the same size of the image, with a '1' indicating the presence of the feature and a '0' indicating its absence. The Some/None line can be used to quickly ascertain if there are any such features within the image that are contained within a Cluster. The count could give the area over which the features exists. Its use in image operations is described further in Section 5.4.2.

The use of a count for the number of responders is not novel in the context of array processors. In 1971 Foster [Foster71] showed that the design of an N input count could be implemented using a maximum of N full adders. His design was used within an array of associative processors. However, if the exact number of responders is not required, only an approximation, then a technique using a resistive summing network and an analog to digital converter may be used [Kaplan63].

The Grid array processor uses a shift register and an accumulator, to count the number of set bits within a word [Pass85]. This cycles at four times the speed of the processor array. The DAP machine has no support for counting bits. In applications such as histogram generation, the count of different pixel values can be performed in parallel across the array, using bit-serial operations [Reddaway90].

Within the WPM, a count design similar to Foster's is used, since in most image and numerical applications a precise result is required. However, the count is limited to the boundary of each Cluster to reduce hardware complexity. The design is implemented in a gate array with a 2-micron design process [Walton89]. At the time of design, no IC packaging was available with the required number of pins - one per PE (256) plus output, control, and power. Thus, the design was modularised into 4 ICs, each performing the count of 64 inputs, Figure 4.5. However, the count could be incorporated within the design of the PE on a single IC, since it requires only 8,000 gates (2,000 per 64 input count).

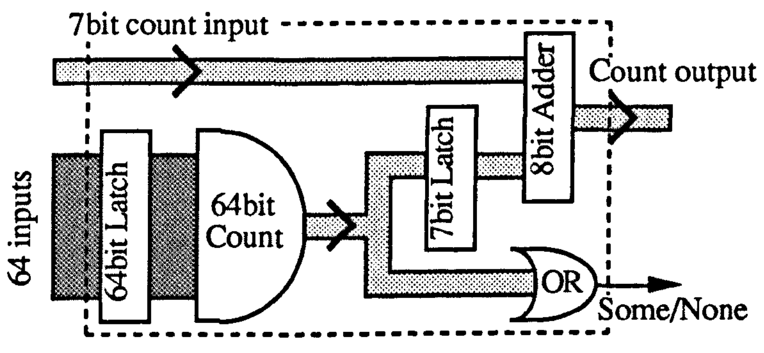


Figure 4.5 - A 64 input count.

To achieve the same clock speed as the DAP chips (10MHz), pipeline latches were incorporated into the design, two in the count data-path and one in the Some/None data-path. Thus, the output of the count is available two cycles after the data was active on the DAP memory lines, and the Some/None one cycle after. Four of these ICs can be connected together as shown in Figure 4.6. The outputs from the left hand count ICs are used as the inputs to the right hand count ICs. The outputs of the right hand counts are added externally to produce the total 256 count output which can be used by the Cluster controller. Similarly the Some/None outputs, from each count IC, are OR'ed externally to produce the Some/None output for the Cluster.

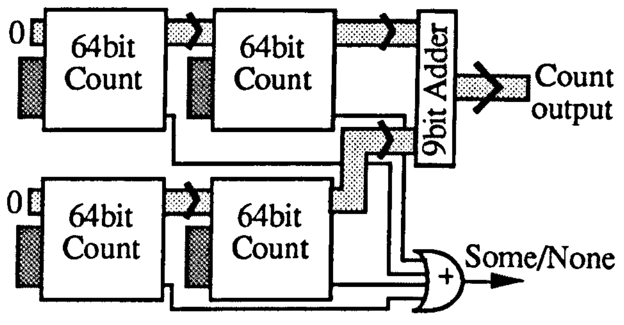


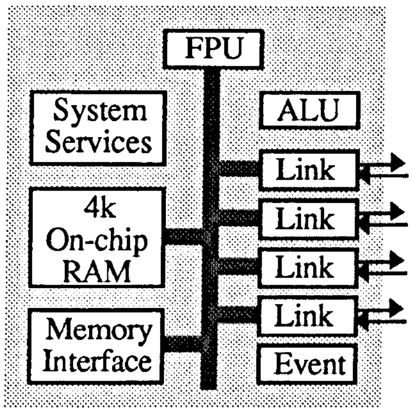
Figure 4.6 - A 256 input count from four 64 input counts.

4.3.2 The MIMD array

The Transputer is used to implement the MIMD array within the WPM. The Transputer is a conventional microprocessor, with on-chip communication capability, allowing several such devices to form an MIMD network. The T800 Transputer is used within the WPM and is described below. A future implementation of the WPM could utilise the T9000 Transputer, when available, with improved performance.

The Transputer consists of a single VLSI device requiring only the addition of a clock circuit, in its simplest form, and has on chip-memory which can be used for both program and data storage. Further, external memory can be added using the Transputers in-built memory interface, requiring only a minimum of ICs for buffering and latching. Communication connections between Transputers are made by simply connecting the relevant pin from one Transputer to the correct pin on another Transputer. These communication channels are called links.

The internal components of a Transputer can be seen in Figure 4.7 for both a T800 and T9000 Version. The 30MHz version of the T800 operates with a peak instruction rate of 30MIPs and 4.3MFLOPs. It contains an on-chip 4k memory cache which can be accessed at 3 times the speed of external memory. Each of the four links operate asynchronously with a capacity of 20Mbits/s in each direction, typically resulting in a 0.9MByte/s sustained communication rate.



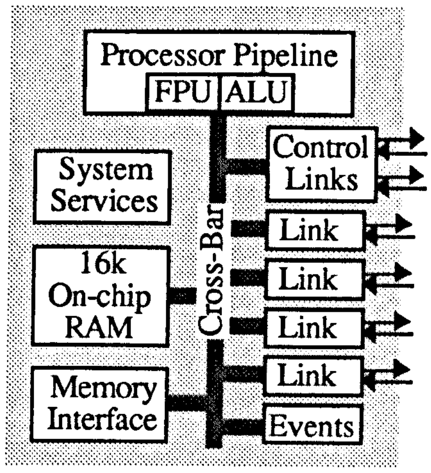


Figure 4.7 - The internal components of a T800 and a T9000 Transputer.

The T9000 operates at a clock speed of 50MHz and differs from the T800 in its computation and communication rates. It can execute instructions at a peak rate of 200MIPS (>70MIPS sustained) and 25MFLOPS (>15MFLOPS sustained) [Inmos91]. It uses super-scalar techniques such that the processor pipeline can issue and execute up to four instructions in each cycle. The T9000 also benefits from an internal cross-bar switch which can interchange four 32-bit data buses and four 32-bit address buses. Its links can operate at a rate of 10Mbytes/s using an extra handshaking line in each of the links directions. An additional two links exist, which can be daisy chained together across several processors, to aid the control, debugging and reliability within a network.

Each Cluster within the WPM uses a single T800 Transputer as its MIMD processor connected in a four-way mesh network. Each of the Transputers has an additional 2Mbyte of external memory and runs at 25MHz. The busing system of the Transputer is used extensively to interface with the Cluster controller, for both the loading of the instruction memory (to drive the SIMD array) and for data communication, the latter in the form of shared memory between the Cluster controller and the Transputer.

4.3.3 The Cluster Controller

The utilisation of the SIMD array is of prime importance for its overall performance. At the lowest level, it is important to keep the array cycling at its full speed while doing useful operations. This is the Cluster controller's primary purpose, to provide an instruction stream to the SIMD array at its maximum throughput. The Cluster controller incorporates a scalar computation capability for the manipulation of numeric data. In addition, it provides an interface between the MIMD processor and the SIMD array, for both high level instruction calls from the MIMD processor to the array and for information retrieval from the array to the MIMD processor.

The controller is similar to a conventional 16-bit single bus microprocessor. The main elements of the controller are: an instruction store to drive both itself and the SIMD array; a scalar ALU with storage capacity for the manipulation of data; and a shared memory interface between the MIMD processor and SIMD array within a Cluster. All of these components need to cycle at the same rate, i.e. at the rate at which the DAP array can execute instructions. The main components of the Cluster controller are shown in Figure 4.8.

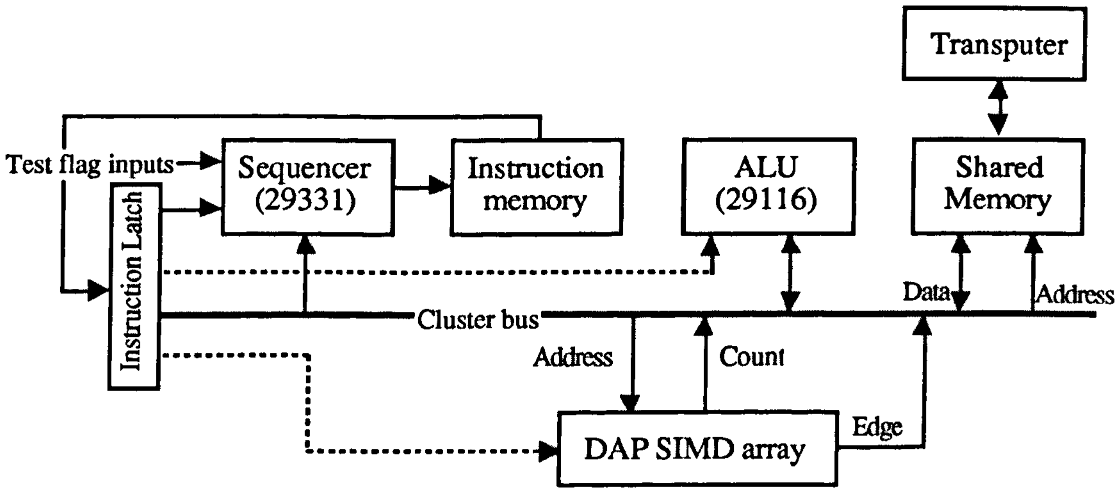


Figure 4.8 - The main components of the Cluster controller.

The main requirement of the controller is to keep the PE array busy, provide memory addressing within the SIMD array, and in loop control within multi-bit operations. A single library routine is used to provide the necessary instructions for a single operation, such as an addition, for an arbitrary data word size. This avoides replication of instruction code for each possible data word size. The requirements for such a routine involves a programmable loop control (looping over the data word size) and indexed addressing into each data operand.

Consider the case when two images are to be added together, both mapped onto the SIMD array such that each PE contains one image pixel. The addition is performed bit-wise, by first adding the LSB of each image and storing the result, followed by the second bit and so on. Indexed addressing of both the input and output images is required. The looping and indexing has to be performed in parallel with the SIMD array operations so as not to impair the arrays performance.

The description of the Cluster controller is split into the following components: the Cluster bus, the interface between the Cluster bus and the DAP SIMD array, the scalar ALU, the instruction sequencer, and the shared memory between the controller and the Transputer.

4.3.3.1 The Cluster Bus

There is a single internal bus that connects all the components of the controller together, termed the Cluster bus. It is used for data transfers between the components of the controller and for the addressing of both the SIMD array memory and the controller's local memory. Each of the components on this bus are treated as ports, with a unique port address. Ports which source data on the Cluster bus are considered separate to destination ports. Thus, each component on the Cluster bus may have one or more port addresses. Example ports already mentioned are the DAP PE's associative count (source) and the SIMD array's Edge Register which can be read from or written to (both a source and a destination).

Only one source port and one destination port may be active in any one cycle (as with a conventional single bus microprocessor). The active ports are specified by a 10-bit field within each controller micro-code instruction, 5-bits for the source port and 5-bits for the destination. A special case is made for the indexed addressing of the SIMD PEs memory, explained in more detail in Section 4.3.3.3 along with the operation of the scalar ALU. A total of 32 source and 32 destination ports are available within a Cluster. A list of the all ports on the Cluster bus is listed in Appendix A.

A 16-bit immediate operand is also incorporated as a separate source port on the Cluster bus, existing as part of the controller instruction. This enables values to be passed to any of the components on the Cluster bus, and can also specify the address of data within the SIMD array when it is known at compile time.

The Cluster bus is shown schematically in Figure 4.9. The source port becomes active when the input to the source decoder from the controller instruction is stable, just after the start of an instruction cycle, and the destination latch is clocked at the end of a cycle, using the controller clock gated with the destination decoder.

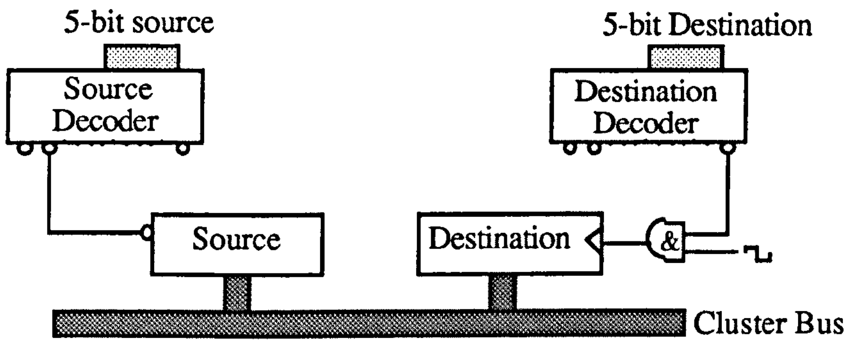


Figure 4.9 - Operation of the Cluster Bus.

4.3.3.2 The interface between the DAP SIMD array and the controller

The DAP SIMD array has several connections to the Cluster bus within the controller. These are: *EDGE* - the Edge Register (source or destination port) to pass data to and from the array; *COUNT* - the SIMD array count output (source), *PEADDR* - the PE memory address (destination) and *PEINV* (destination) - a one-bit broadcast facility across the SIMD array. *PEINV* uses the lowest bit from the Cluster bus and optionally inverts the value from the input multiplexor within each DAP PE. The interface between the Cluster bus and the DAP SIMD array is shown in Figure 4.10.

The Edge Register is, in effect, a two way latch in that it can be written to, or read from, by the Cluster bus and the SIMD array. Logic around the Edge Register interprets the *EDGE* source and destination Cluster bus controls, along with the DAP instruction group, for the enabling and clocking of this register. A run-time error can occur when this register is written to from both the Cluster bus and the DAP SIMD array at the same time. This sets a single bit within an error latch which can be subsequently read at the end of an instruction sequence when checking for run-time errors. Note that this error is a result of bad programming and should be picked up at program compilation time.

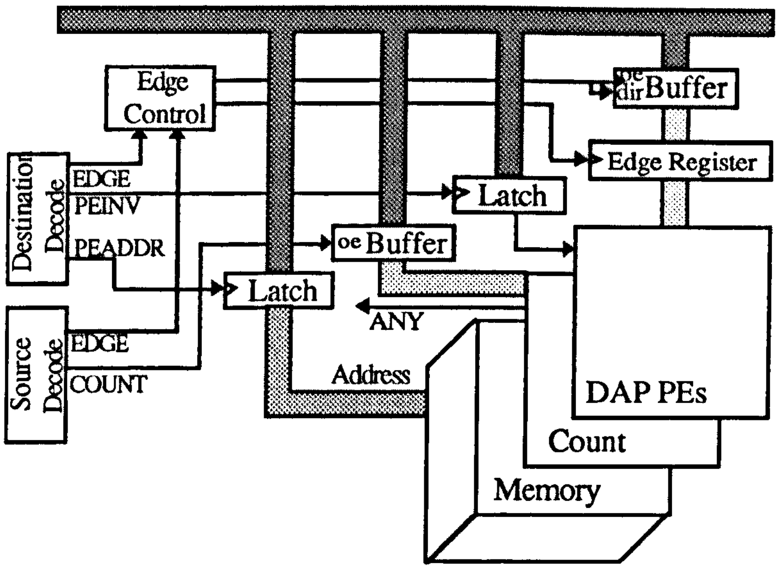


Figure 4.10 - Interface between the SIMD array and the controller.

The *ANY* line from the count device is not connected to a port on the Cluster bus but is instead connected to the controller's sequencer. This allows conditional operations to be performed on the result of a Some/None calculation.

4.3.3.3 The scalar ALU

The role of the ALU is to provide scalar manipulation of SIMD array responses and to provide indexed addressing for the SIMD array memory. It was originally intended that conventional bit-slice components would be used to build a 16-bit ALU. These components have the advantage of being flexible, readily available and can be incorporated into designs running at clock speeds in the 20MHz range. However the AMD29116 component was chosen, a modification of the conventional bit-slice architecture, containing a 16-bit ALU on a single device [AMD86]. The AMD29116 was also used in the prototype DisArray [Page89].

The AMD29116 can perform additions, subtractions, and logical operations between one, two and three operands. In addition, a barrel shifter enables word rotations on one of these operands. Operands may be taken from an external input, an internal accumulator or a internal register file (32 words). The results from the ALU may be stored in the accumulator, the register file, or placed on the external outputs. Flags representing a Carry (C), Zero (Z), Negative (N), and Overflow (O) as a result of an ALU operation are also produced.

A priority encoder is also incorporated within the AMD29116. It produces an output corresponding to the position of the highest bit set within the input operand. This is useful for finding the position of the first responder from the SIMD array when using its associative response mechanisms. All the constituents of the ALU are connected together via an internal bus.

The way in which the ALU is connected to the Cluster bus is shown in Figure 4.11. The ALU may source data onto the Cluster bus or be a destination port, taking data from it. The output flags C, Z, N, and O are latched at the end of each cycle, and are available to the instruction sequencer at the start of the next cycle. They may be used for conditional operations such as instruction branching.

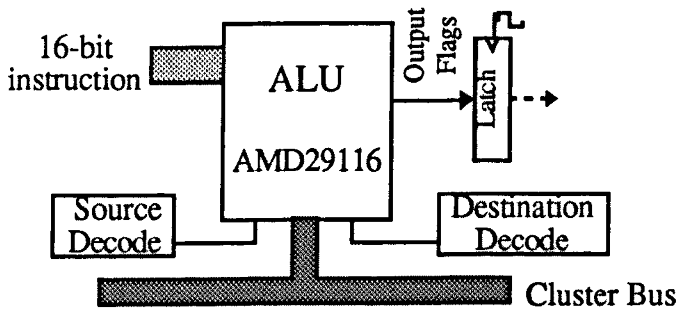


Figure 4.11 - The controller's scalar ALU (the AMD29116).

Indexed addressing is achieved by switching the ALUs internal bus from being an input to an output half way through a cycle. An operand, representing the base PE data memory address, is read into the ALU and latched at the mid-point of the cycle. The address value can then undergo an operation within the ALU, such as an addition with one of its internal registers, and then output the result towards the end of the cycle. The ALU register used for the addition can be thought of as an indexing register indicating which bit of the operands, within the SIMD memory, is currently being used in a multi-bit operation. The same ALU register can be used to index into one or more operands, and is incremented for each subsequent bit, up to and including the word-length of the data being used in the SIMD array.

The actual device used for the ALU is the AMD29116-A which cycles 25% faster than the standard part at 75ns. Switching around the ALUs internal bus at the half cycle point, from input to output, allows sufficient time for data to propagate externally through the Cluster bus to a destination latch.

Indexed addressing, as required by the SIMD array, is treated as a separate port on the Cluster bus, in addition to the normal ALU source and destination ports. The destination is specified by

$$\left(\text{Clock} \ \& \ \overline{\text{Dest}}_{\text{alu}}\right) + \left(\text{Clock} \ \& \ \overline{\text{Dest}}_{\text{immed\_alu}}\right)$$

and the source is specified by

$$\left(\overline{\text{Source}}_{\text{alu}}\right) + \left(\overline{\text{Clock}} \ \& \ \overline{\text{Source}}_{\text{immed\_alu}}\right)$$

where Source and Dest are available from the Cluster bus port decoding (active low) and Clock is the controller's system clock, which is high in the first half cycle.

4.3.3.4 Instruction sequencing

The instruction sequencer provides the instruction stream for all the elements within the controller: the DAP SIMD array; the ALU; the Cluster bus addressing; and the sequencer itself. The sequencer used is the AMD29331 - a 16-bit unit with a cycle time of 50ns. It contains an address register, a counter and a 33-word deep stack along with associated busing circuitry. The operation of the sequencer is decoded from a 6-bit instruction word supplied at the start of each cycle.

The instruction memory that the sequencer addresses is a horizontal micro-code, combining the separate instructions, for each of the controller's components. This results in a 61-bit format, as shown in Figure 4.12, and is implemented as a 64-bit wide instruction word using 4bit-wide memories with 3-bits spare. The 16-bit sequencer enables the addressing of a total of 64kwords, only 16k of which are implemented within the prototype Clusters.

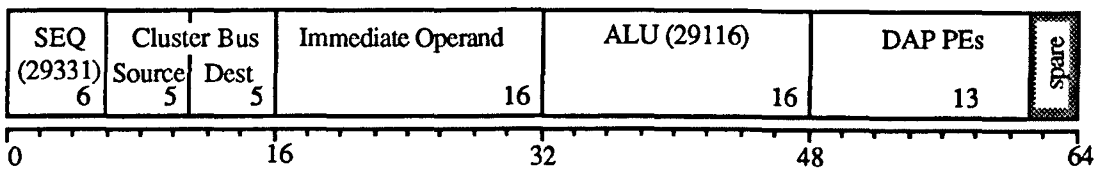


Figure 4.12 - The Cluster controller's instruction word.

The sequencer contains an address register which can be incremented in every cycle, outputting its value to the external instruction memory. The counter can be used within loops, set at the start with an appropriate value, and decremented on each loop iteration. Both the address and counter registers can be set from an external source, across the Cluster bus or from the integral 33-word deep stack. Thus, nested operations are allowed including subroutine calls and looping.

Operands, such as subroutine address calls from the Transputer, can be passed to the sequencer via the Cluster bus. The sequencer reads the Cluster bus as its default state (i.e. defaults to a bus destination) but can also source the bus allowing its internal address register or stack to be read. The sequencer is shown schematically in Figure 4.13 along with its interface with the Transputer.

At the start of each cycle the sequencer decodes and executes an instruction, outputting the resultant address to the instruction memory. The address output may be the current contents of the address register, an address from the internal stack or an address supplied from the Cluster bus. The instruction addressed in the instruction memory is then stored in the instruction latch at the end of the cycle and is executed in the next cycle. The memories used are 16k by 4bits with an access time of 45ns. The sequencer takes < 30ns to perform its instruction, leaving ample time for latch output and set-up delays within the 100ns cycle time.

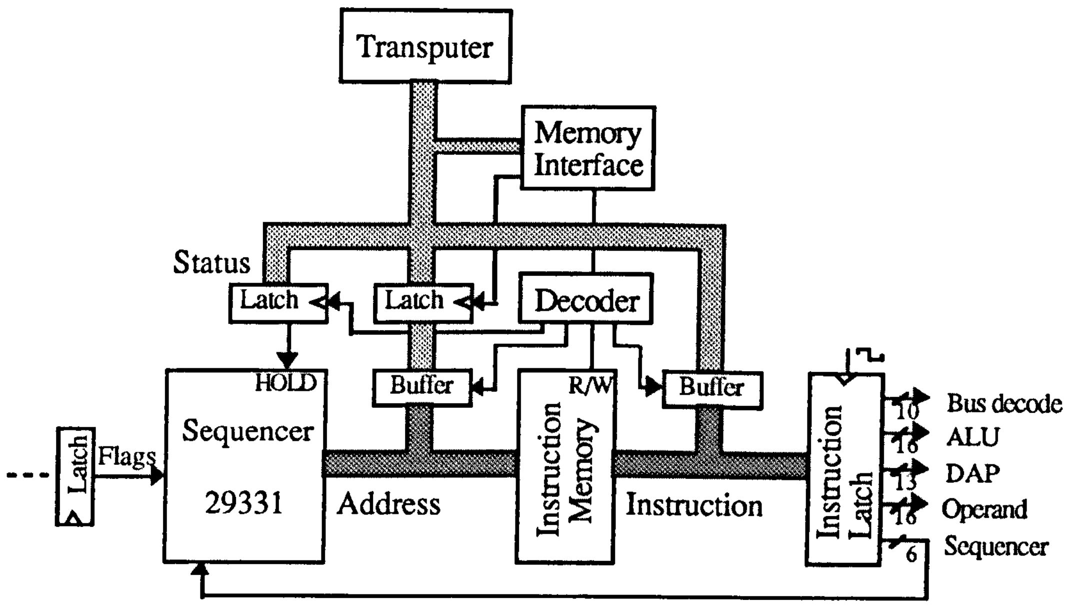


Figure 4.13 - The controller's sequencer showing Transputer Load Path.

The flag outputs from the ALU (C,Z,N,O) can be used for conditional branching operations, being latched from the ALU at the end of the previous cycle. The particular flag used within a conditional operation is specified by a 4 bit selector port, which can be written to from the Cluster bus. A total of twelve flags can be connected to the sequencer in this way. Further flag inputs are used by the controller, including the Some/None function from the SIMD count and local neighbour synchronisation flags (discussed in Section 4.3.4.2).

The Cluster instruction memory is loaded at boot-time from the Transputer. This is achieved by logic around the sequencer, asserting its *HOLD* input to three-state its output, and enabling the Transputer to read and write to the controller's instruction store. The addressing and data system of the Transputer is thus enabled to the controller's instruction memory, and treated as if it where mapped within the Transputers own memory. The instruction memory may also be read back to the Transputer for debugging purposes. During loading or debugging phases the clock to the rest of the controller is suspended.

4.3.3.5 The shared memory between the controller and the Transputer

Communications between the controller and the Transputer take place via shared memory. Typically, the Transputer will queue requests for the controller to perform computation on the array, which may subsequently generate results. Thus, a bi-directional data-flow is generated. The memory is also used as the controller's only local data store.

The shared memory is transparently accessible from both the Transputer and the controller. Conventional dual-ported memories have built-in logic which gives priority to the processor making the earliest access, and if necessary, delays the second processor's access. However, the timing of the controller's operation is more critical than that of the Transputer and hence conventional dual-port operation is not suitable for this shared memory. Instead, the shared memory is implemented using standard memories with additional arbitration logic.

It is important that the controller does not have to wait for access to the memory, so as not to decrease the utilisation of the array. More importantly, synchronisation with neighbouring Clusters should not be lost due to a shared memory access by a Transputer. There would be no way of guaranting that several controllers would be delayed, by the arbitration system in the dual-ported memories, by the same amount. Although the Transputers can execute the same program, they run asynchronously at the timing level. Thus, the Transputer can be made to wait by the shared memory arbitration logic for each shared memory access until it is known that the controller is not accessing the memory.

The controller, unlike a more conventional processor, is only a single bus system having no separate mechanism for memory addressing. The Cluster bus is used for both address and data transfers to and from the shared memory. The shared memory, along with the interface to the Transputer, is shown in Figure 4.14.

A write to the shared memory from the controller takes three cycles. This consists of a data write to the memory address latch, a data write to the data latch, and then the actual memory write operation. Note however, the last cycle can be overlapped with the start of a subsequent write operation, therefore taking only two cycles in a pipelined fashion. A read from the memory is very similar, again taking three cycles. The first cycle is used to write to the address latch, the second reads the data from the memory into the memory latch, and the last transfers the data across the Cluster bus to its destination.

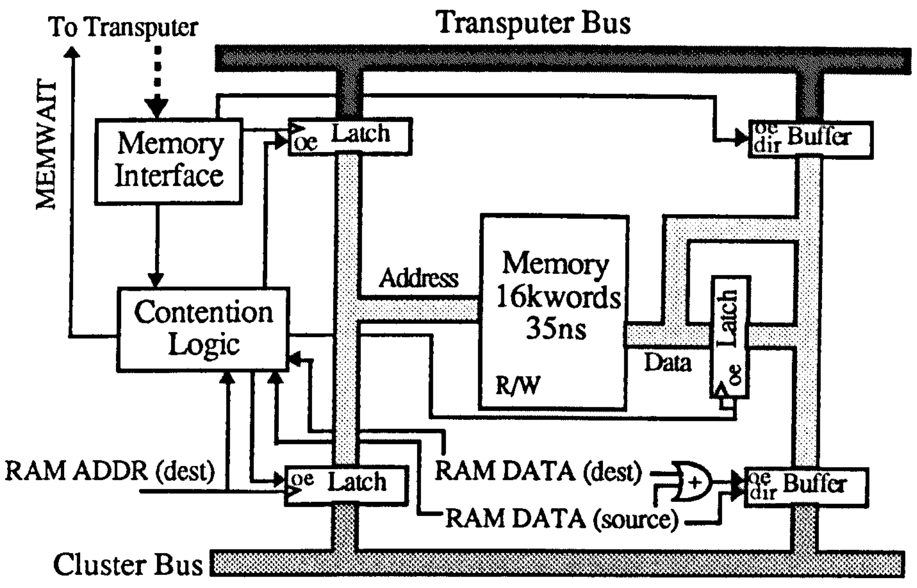


Figure 4.14 - The shared memory between the Transputer and the controller.

The arbitration logic in Figure 4.14 performs the function of locking the Transputer out if the controller is doing a memory access in the next cycle. The *MEMWAIT* signal is used to hold the Transputer in the middle of a memory access and is only asserted when the controller and Transputer are both accessing the shared memory. It is assumed that the first cycle of the controller's memory access is always a write to its address latch, followed a write to the data latch when writing to this memory. Thus, the Cluster bus destination ports, *RAMADDR* and *RAMDATA* are used to ascertain when the controller is accessing the shared memory.

4.3.4 Connecting Clusters together

Communication is possible between adjacent Clusters at both the Transputer and SIMD array levels. The Transputer level allows message passing, and can be used for accumulation of results between Clusters. At the SIMD level, groups of Clusters can perform filtering operations, on the iconic data, as if the Clusters within the group formed a conventional SIMD array. This is achieved by the synchronisation of controllers, such that adjacent controllers and SIMD arrays perform the same operations in parallel. The interconnection of four Clusters is shown schematically in Figure 4.15.

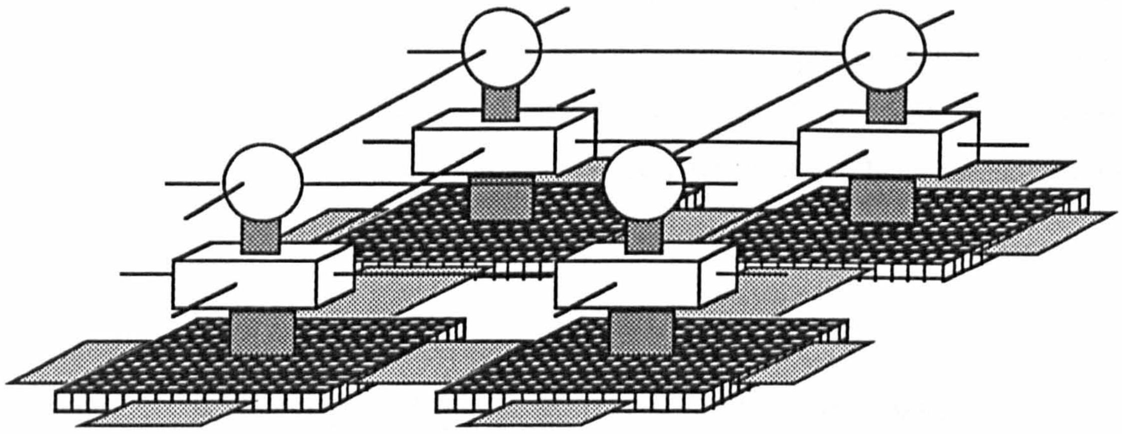


Figure 4.15 - The interconnection of four Clusters.

The SIMD arrays are connected together between Clusters by the 4-way NEWS network between the SIMD PEs. A total of 64 lines emerge from each Cluster - 16 from each edge of the SIMD array. Each set of 16 lines is connected to their neighbouring Clusters, forming a 2D mesh. SIMD PEs, on the edge of the Cluster array, are connected to their opposites on the other side of the SIMD array, forming a 2D torus network.

Each Cluster has its own clock generation circuitry operating at 10MHz. With the number of Clusters in a full sized WPM approaching 8x8, it might be unsafe to have a global clock for all Clusters without incurring severe distribution problems [Jesshope89]. Adjacent communications at the SIMD level use a synchronisation mechanism in the form of handshaking.

4.3.4.1 Communication at the SIMD level

The NEWS communication network of the SIMD array within the Cluster can operate in one of two modes, either in array mode or Cluster mode. Cluster mode wraps around the data between the North/South and East/West edges. Array mode allows shifting of data between Clusters (operating a group of Clusters as a larger array) and connects the left edge of the left-most Clusters to the right edge of the right-most Clusters to form a 2D Torus for the whole SIMD array.

In addition, any of the Cluster edges can be specified to shift in a line of zeros (using the *CYC* control line of the DAP PEs), and is used in array mode when the Cluster concerned is on the edge of the array. E.g. when the Cluster is on the north edge of the array, a south shift would shift a row of zeros into the top most row. This can also be used when a group of Clusters are synchronised - the Clusters on the edge of this group can have their communication inputs zeroed, which would have normally come from Clusters outside of the group.

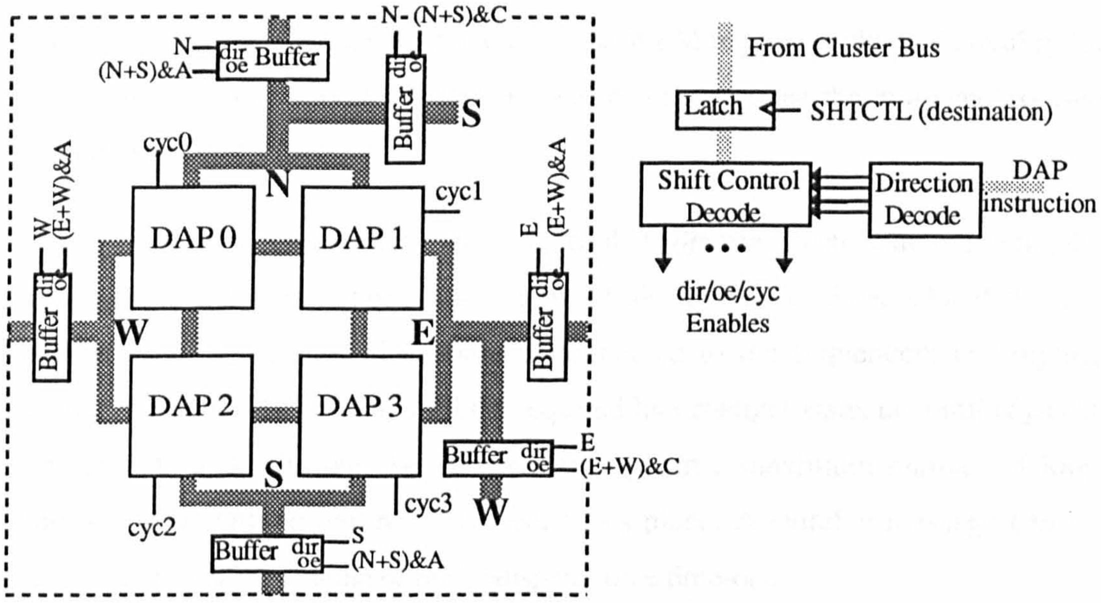


Figure 4.16 - Interconnecting Cluster SIMD arrays.

The communication connections within a Cluster are shown in Figure 4.16. These extend in all four directions to join with the neighbouring Clusters. All the buffers shown in Figure 4.16 are bi-directional and 16-bits wide. The enabling and direction controls for these buffers, along with the *CYC* DAP control lines, are decoded from the DAP instruction and the shift-control register. The shift control register is a destination port on the Cluster bus (*SHFTCTL*) and its operation is described in

Appendix A. The buffer enable lines are logical functions between the lines indicating a shift in each of the NEWS directions and the mode of operation - 'A' for array mode and 'C' for Cluster mode.

The two buffers which use the Cluster mode line perform the wrap-around within the Cluster, from North->South, from East->West, and vice versa. The remaining four buffers are used for array mode operation. In addition to the operation of these buffers, the synchronisation mechanism ensures that two adjacent Clusters do not shift to each other which could cause hardware shorts. The synchronisation mechanism is described below.

4.3.4.2 The synchronisation of Clusters

A shift within an SIMD array consists of a read, a shift, and a write operation taking three cycles. However, this operation only works if the whole SIMD array is operating in lock-step using the same clock. The WPM requires additional circuitry for the shifting of data between Cluster boundaries to overcome the non-synchronised clocking between Clusters.

The synchronisation mechanism used assumes that adjacent Clusters are executing the same operations but are out of clock step by a finite amount of time. A hand-shaking line, for each of the four directions, is connected to the sequencers test inputs, allowing the controller to loop until the required line changes state, i.e. until adjacent communication is possible. The sequencer is given a maximum number of loop iterations to undertake before a time-out takes place. A suitable message can be returned to the calling routine of the Transputer on a time-out.

The synchronisation mechanism is shown in Figure 4.17 for a Cluster B, illustrating its communication interface to the East and West. In each direction there are two latches, one for each of the two data communication directions, along with two single-bit flags indicating if there is any data in the latches. The flags are set by a Cluster shifting data into the respective latch and are reset by the adjacent Cluster shifting data out from the latch. The outputs from the flags are passed to the controller's sequencer, via the condition code latch (Figure 4.13), enabling conditional looping on the status of each flag.

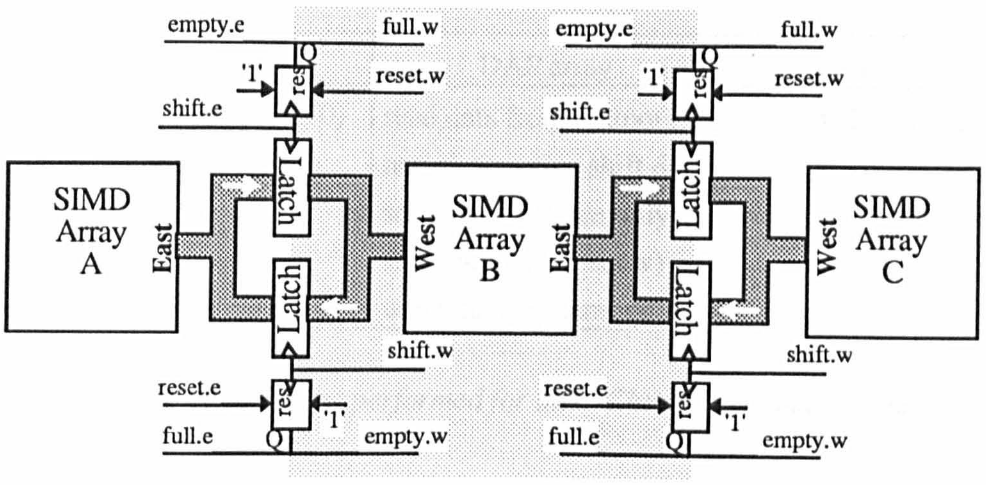


Figure 4.17 - The hand-shaking mechanism between Clusters.

The operations performed for a synchronised shift, from East to West for Cluster B (left to right in Figure 4.17), are as follows. Note that only the control lines within Cluster B's shaded area are used. It is assumed that all three Clusters, A, B and C are performing the same synchronised shift.

Firstly, a check is made to see if the East latch is empty (empty.e = 0) and repeated until it is, while the DAP PEs constantly reads data from memory. Cluster B performs a non-destructive shift East, shifting its right-most column into the East latch and causing a transition on shift.e and setting empty.e (indicating the presence of data). A non-destructive shift differs from a normal shift in that the data appears at its destination on the NEWS network but is not latched into the Q register within the DAP PEs (this is a modified DAP instruction). The non-destructive shift is repeated until full.w has been set by Cluster A. Note that the transition on shift.e occurs only on the first non-destructive shift, latching the data only once.

Once full.w is set, a normal (destructive) shift is performed, activating reset.w, which zeros full.w (no data now present in the West latch). The data is then written into the DAP memory. The whole process can be repeated for each subsequent bit of data being shifted. The whole operation can be seen in Table 4.2 which indicates the operations being performed by the sequencer, DAP array and ALU.

step	Sequencer	DAP array	ALU
1	repeat until (empty.e=0)	read data from memory	provide PE address
2	repeat until (full.w=1)	non-destructive shift & set empty.e (once only)	-
3	-	shift (activate reset.w)	-
4	-	write data to memory	-

Table 4.2 - Operations performed for inter-Cluster communications.

Each of the Clusters within a group, requiring synchronised SIMD communications, operate the loop described in Table 4.2 with the exception of those Clusters at the edge of the array or group. These only have a single neighbouring Cluster and so only need half of the checks performed above. Note that there is a dependency only on rows of Clusters for East->West communications, and only on columns for North->South communications. The time for a single synchronised shift is four cycles (cf. three cycles for a normal SIMD shift). This represents a 25% overhead for the required hand-shaking once each of the Clusters, within the group, have been initially synchronised.

A ripple effect is apparent in the direction of the communication. For a shift East, the neighbouring West Cluster will be at most one cycle ahead of the East Cluster. This is due to the West Cluster having to wait until after the East Cluster has performed its first non-destructive shift. This results in a latency delay, equal to the number of Clusters being synchronised, in the overall communication time.

There is also an initial synchronisation overhead, to ensure each of the Clusters start executing the loop, which should be added to the total time. This effect can be seen more clearly in the Figure 4.18 below, where the group of Clusters, A->F, are shifting towards Cluster F. The ripple effect can clearly be seen moving down the Clusters, in the direction of the shift.

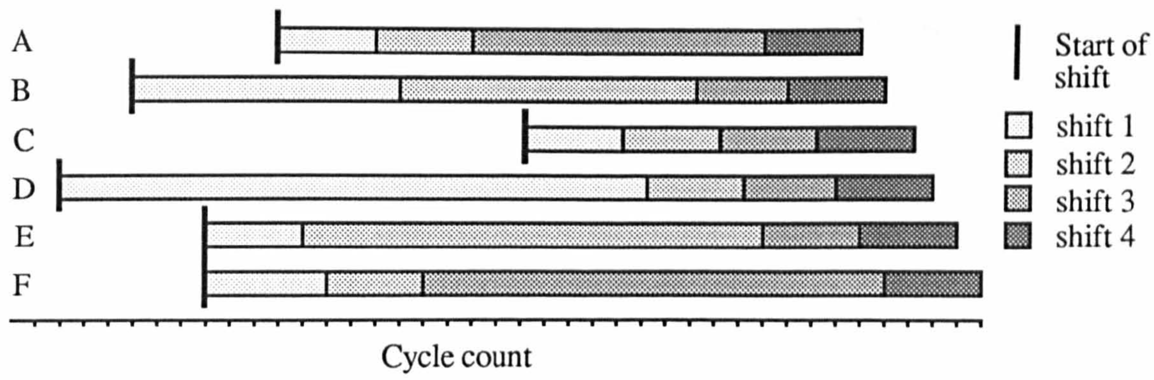


Figure 4.18 - Example synchronisation between six Clusters in a shift operation.

The total time for a shift is:

$$T_{sync} + nbits*4 + N_{cluster}$$

where $N_{cluster}$ is the number of Clusters being synchronised in the same row, nbits is the word-size of the data and T_{sync} is the number of cycles that the slowest Cluster is behind the quickest Cluster. Ignoring the time for initial synchronisation, T_{sync}, which would be apparent in any synchronisation mechanism within the WPM, the total overhead (compared with the normal shift time of nbits*3) is :

$$nbits + N_{cluster}$$

Subsequent synchronised communications between Clusters in the same direction will not be subject to the delay T_{sync}. However, subsequent communications in the opposite direction, (East->West in the above example) will have a latency time of $2*N_{cluster}$. This is double the first latency due to the dependency between Clusters being changed to the opposite direction, the West Cluster will now be at most one cycle behind the East Cluster. Any shifting in the orthogonal directions (North->South and South-> North) will undergo the same initial synchronisation, having the initial delay of T_{sync}.

The operation of the synchronisation mechanism resembles that of a FIFO but has only the capacity of a single word in each of the communication directions. The latches in each direction could be replaced with a FIFO, and the shifting operations changed so that the non-destructive shifts are done for all the bits of the data before the destructive shifts, thus eliminating the ripple effect in the direction of the shift.

However, there is an added hardware overhead in the use of FIFOs and there is always the possibility that they may fill and create the same waiting requirements as with a single latch.

The overhead in the WPM synchronisation mechanism is shown in Table 4.3 for 8 and 32 bit data. Note that the overhead asymptotically approaches 25% for larger data-words (due to the 4 cycle shift instead of 3). The use of FIFOs would result in a 25% overhead for all data words, assuming that the FIFO size is not exceeded. Although these figures may appear large at first glance, the time for shifting within an application is typically small when compared to the overall time.

Synchronised communication:	Time (cycles)	Overhead 8-bit data	For 32-bit data
1st communication	T_{sync}+4*nbits+$N_{cluster}$	40 %	29 %
2nd+ (same direction)	4*nbits+$N_{cluster}$	40 %	29 %
2nd+ (opposite direction)	4*nbits+2*$N_{cluster}$	50 %	33 %
2nd+ (orthogonal direction)	T_{sync}+4*nbits+$N_{cluster}$	40 %	29 %

Table 4.3 - Overhead for synchronised communications.

4.3.5 The Prototype system

Two prototype Clusters have been built within a rack system, each comprising of five 6U Eurocards connected together via two bus back-planes. The timing of the controller's clock was found to be the most critical item during the build of the prototypes. The controller's clock was buffered onto each of the system boards within the rack. Similar buffering was placed in the Cluster bus to reduce its loading.

The five prototype boards of a Cluster were:

- a single Transputer board, equipped with a 25MHz T800 and 2MBytes of RAM. The Transputer memory system drives the lower bus back-plane.

- a controller board comprising the sequencer, ALU, instruction memory and instruction latches. This drives the instruction to the DAP and Cluster bus port addressing on the upper bus back-plane.
- a shared memory board with interface to the Transputer and system clock
- a DAP PE board containing 4 DAP chips (16x16 PEs)
- a DAP memory board attached to the DAP PE board via four 96-way DIN connectors

The five Cluster boards are shown in Figure 4.19.

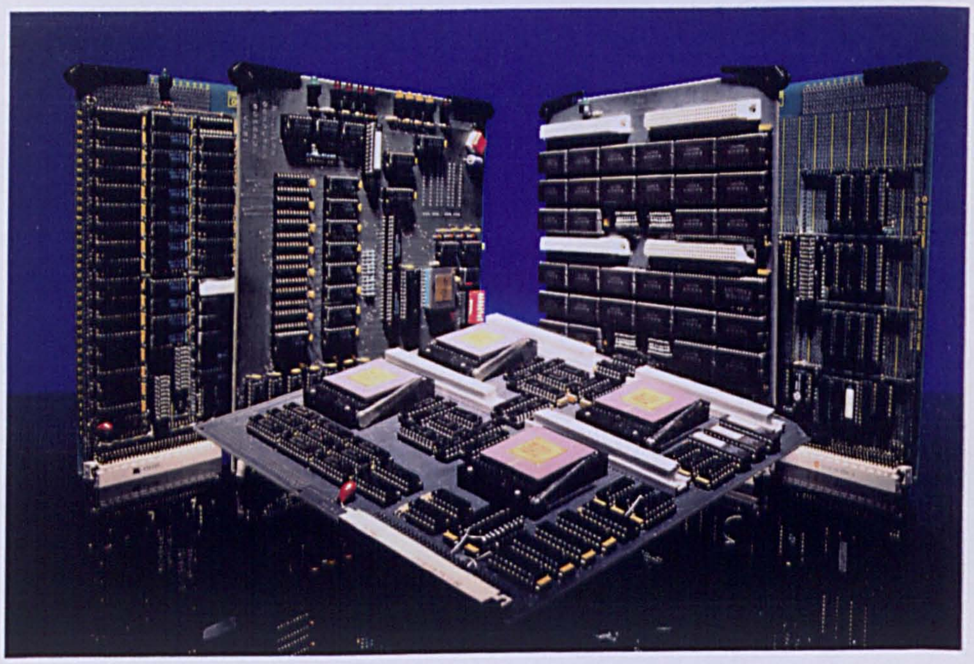


Figure 4.19 - The prototype Cluster boards.

The prototype Cluster was designed to be modular, resulting in some redundancy of circuitry between the boards. The integration of the Cluster to a greater extent would, for instance, eliminate the need for duplicate Transputer interfaces on the controller and shared memory boards. Duplicate Cluster bus decode logic, and the buffering between boards would also not be needed. These would be important considerations if the controller were to be made to cycle at a faster rate, necessary for faster SIMD processors.

Debugging capabilities were included in the form of being able to single step the system clock within the controller. This can be performed under the control of the Transputer using a bit from the status latch, shown in Figure 4.13. The sequencer address and Cluster bus data values, which indicate the controller's activity, can be monitored by the Transputer for tracing and breakpoint purposes. This is achieved by extra memory-mapped buffers between the address output of the sequencer and the Transputer bus (not shown in Figure 4.13) and also a latch between the Cluster bus and the Transputer bus (not shown in Figure 4.14).

It has not been necessary to single step parts of the controller outside of a debugging environment except in the case of booting the system. At boot time the system clock is disabled and the instruction store is loaded by the Transputer. The sequencer is single stepped, by itself without clocking the rest of the system, so as to fill the instruction latch with the first valid instruction of the controller. The clock can then be enabled to the whole controller to commence normal operation.

4.3.6 M-SIMD Operation of the WPM

The computational components within the WPM, in terms of its component Cluster, have been described above. Each of the Clusters operate on their own instruction stream, connected in a 4-way mesh at the SIMD and MIMD levels and have a synchronisation mechanism at the controller level enabling data movements at the SIMD level. Clusters thus operate autonomously, except when iconic data movements are required which produces operational dependencies between adjacent Clusters.

The associative facilities of the SIMD arrays have been cut into the Cluster sized boundaries, thus aiding local operations within the Clusters. However, it is not always the case that the processing required will fit within a Cluster (such as an object contained within the image being twice the Cluster size). In this case, the associative response will be needed across Cluster boundaries. This can be achieved by firstly performing the associative response on each of the object Clusters, with results accumulated with those of adjacent Clusters. The accumulation may be performed on either the Transputers or on the controllers using synchronised SIMD shifting.

When an object within the image could fit within one Cluster, but lies across Cluster boundaries, it can be shifted to fit fully within a single Cluster. This minimises the number of Clusters associated with the object thus increasing the maximum throughput. When the object can not fit fully within a Cluster boundary, the set of Clusters involved with the object should be minimised to reduce the overhead of the synchronisation mechanism for data movements and accumulation of results. These issues are discussed in Sections 7.2 and 7.3.

The local autonomy that is available within Clusters can be used for local iconic operations, or alternatively an additional type of parallelism may be exploited. That is, a form of algorithmic parallelism such that each Cluster, or set of Clusters, can perform their own sequence of operations on their own data. Either containing copies of a complete image or different images from different sensors, for multi-sensor fusion. In a multi-user system, groups of Clusters could be assigned to different users. This type of parallelism has not been exploited on the WPM.

4.4 Programming the WPM

The Warwick Pyramid Machine combines both SIMD and MIMD elements which have quite separate programming paradigms. An array of MIMD processors such as the Transputer is conventionally programmed in Occam or other high-level languages with parallel constructs and communication capabilities as extensions (e.g. C). SIMD arrays are again usually programmed in a high level language, such as Fortran with array extensions, or with embedded assembly instructions within the language.

These two paradigms were reflected in the implementation by the fact that the WPM was first programmed using two separate languages, using software tools to collate both at run time. The Transputers were programmed in a parallel version of C, and the DAP SIMD array in its native assembly language, APAL, with associated instructions for the controller forming the CLuster ASSembly language (CLASS). The format and use of CLASS is described below. Further details are given in Appendix B. A simulator of the Cluster SIMD array and controller, PSIM, has also been written which uses CLASS instructions [Francis89].

Investigative work has been performed providing a unified programming paradigm within an object oriented programming language, C++. This work is reported by Vaudin [Vaudin91] and is illustrated later. Both the programming approaches lead to the implementation of a set of library routines, implemented in CLASS, that can be used on the DAP SIMD array.

4.4.1 The Cluster assembler - CLASS

The Cluster assembler has a one to one mapping between an assembly instruction and the instructions executed on the Cluster. Each CLASS instruction contains the parallelism available within the Cluster enabling each of its components to be programmed within the same instruction. Thus, the programmer needs to be fully aware of the parallelism available within the Cluster.

A CLASS instruction is split into several fields, each representing the instruction to a component of the Cluster, and is compiled into the 64-bit horizontal instruction word described earlier. The fields, each separated by a semi-colon and in order, are: the 29331 sequencer, the 29116 ALU, the DAP SIMD array and the Cluster bus control along with the optional 16-bit immediate operand. An example CLASS instruction is shown below.

```
FOR_D ; MOVE SOZR R01   ; SF          ; SEQ = 0x10
```

The 'FOR_D' instructs the sequencer to initialise a loop with 0x10 iterations (indicated in the Cluster bus field at the end of the instruction), the ALU is instructed to move a zero to the register R01, and the DAP's instruction moves a zero to the store plane currently addressed by its memory register *PEADDR*. An understanding of this assembly language requires an understanding of each of the components within the instruction.

The instructions that the sequencer and ALU can perform, along with their respective mnemonics, are briefly described in Appendix B. Further details can be found by referring to their respective data sheets [AMD86, AMD87]. The instructions that can be performed on the DAP are defined by the APAL instruction set [AMT88].

The DAP executes its instruction the cycle after it is specified in the CLASS instruction. The placement of the DAP instruction the cycle before it is executed improves the readability of the CLASS code. The operations for the ALU, and subsequent writes to the *PEADDR* array memory address latch, occur on the same line as the DAP instruction that uses the memory. The contents of the address latch will be correct at the start of the next cycle when the DAP instruction is executed.

The Cluster bus is controlled by the final field in a CLASS instruction. This simply takes the form of 'destination port = source port'. When the immediate operand is the source only a value need be specified. The special case of passing the immediate operand through the ALU before writing the result to the *PEADDR* latch, for PE indexed addressing, is done by specifying the immediate operand before the destination (PEADDR) and source (ALU) ports. A list of the ports and their uses is listed in Appendix A.

All Cluster bus port data interchanges take a single cycle, although there are delays in a few cases. Some of the delays are due to the pipeline in the DAP instruction and others due to the single bus access to the controller's shared memory. These Cluster bus delays are listed in Table 4.4 below.

Operation	Delay
ALU calculation -> sequencer	1
ALU calculation -> sequencer use of condition codes	1
writing shared memory *RAMADDR* -> reading *RAMDATA*	2
PE operation -> reading the *EDGE* port (associative response)	2
PE operation -> reading the *COUNT* port	3

Table 4.4 - Delays across the Cluster bus.

4.4.1.1 The use of CLASS for Cluster operations

Example routines for the Cluster, programmed in CLASS, are given below. The first is a multi-bit copy routine and the second an addition. Both the routines are of similar structure, with a loop performed on the sequencer with the FOR_D and DJMP_S instructions, looping for a total of nbits. The constants src1, src2 and dest specify the locations of the input images and the output image. The registers R10-R12 in the ALU are used as SIMD memory addressing registers, incremented in each iteration of the loop, and take initial values of the image addresses-1. The minus one is due to the pre-incrementation, before output, of an ALU register.

```
/* Copy - copies data from PE memory src to PE memory dest */
#define copy(nbits,dest,src)                                \
      ; MOVE SODR R10   ;           ; ALU = src-1           \
      ; MOVE SODR R11   ;           ; ALU = dest-1          \
FOR_D ;                 ;           ; SEQ = nbits           \
      ; INC SORR R10    ; QS        ; PEADDR = ALU          \
DJMP_S; INC SORR R11    ; SQ        ; PEADDR = ALU


/* Add - adds data at PE src1 to src2 and puts in dest */
#define add(nbits,dest,src1,src2)                           \
      ; MOVE SODR R10   ;           ; ALU = src-1           \
      ; MOVE SODR R11   ;           ; ALU = src2-1          \
      ; MOVE SODR R12   ; QT        ; ALU = dest-1          \
FOR_D ;                 ; CQ        ; SEQ = nbits           \
      ; INC SORR R10    ; QS        ; PEADDR = ALU          \
      ; INC SORR R11    ; CQPCQS    ; PEADDR = ALU          \
DJMP_S; INC SORR R12    ; SQ        ; PEADDR = ALU
```

The routines above take a total of 2*nbits+3 and 3*nbits+4 cycles to execute respectively, where nbits is the word length of the data being processed. These times are typical of two and three operand operations, one input and one output in the copy, and 2 input and 1 output in the addition. The DAP PE requires a single cycle for each memory access both for a memory read and a write.

The routines written in CLASS were initially defined within a macro. That is, a macro heading is used to replicate the code contained within the body of the macro, replacing its parameter with different values for each use. Thus, the same macro can be used for an image of any word length. However, the code is replicated for each use of the macro. This could become a limitation on a large application or within an operational system when the amount of controller instruction memory is limited.

4.4.2 Remote procedure calls between C and CLASS

An alternative method defines each of the macros as a subroutine call, with associated parameters, initiated by the Transputer from its C program. A section of the shared memory is reserved for a remote procedure call (RPC) mechanism, for the calling of these routines. This enables the Transputer to provide the address of controller library routines, within the instruction memory (which is known at compile time), to be executed along with any necessary parameters.

The frame for each RPC call consists of: the address of the library routine within the instruction memory; the location of the start of the next RPC frame within the shared memory; and the library routines parameters. An example of the RPC frame is shown in Figure 4.20. The arguments typically define the location of the operands and provide scope for the passing of results back to the Transputer. A calling routine exists within the Transputer C program, to set up the RPC frame, and a server program runs on the Cluster to execute RPC calls.

Format	Call Address	Next Frame	Argument 1	Argument 2	Return Arguments	Call Address	
	100	101	102	103	104	105	RAM Address
Example	300	105	8	16		0	• • •

Figure 4.20 - The Remote Procedure Call frame used within a Cluster.

The controller executes a loop, polling the start of the RPC frame area in the shared memory, until an RPC frame is set. This is indicated by a non-zero at the start of the frame. The address of the next frame is put into a memory location within the shared memory so that it can be read back and used to indicate the location to poll for the start of subsequent RPC frames. A single ALU register is used as a pointer to the parameter list, and is incremented every time an argument is taken or put into an RPC frame. The ordering of the parameters is defined both within the C calling procedure, and within the CLASS routine.

The difference in code for the copy routine using an RPC frame is shown below. The main body of the code remains unchanged, only the initialisation of the image operands, within the ALU, has changed. The operations `function()`, `get_arg()`, and `function_end` are macro definitions which perform the function of setting a label (0 cycles), getting the next argument from the RPC frame (3 cycles per argument), and returning from a subroutine call (1 cycle) respectively.

```
function(copy)
get_arg(R10)
get_arg(R11)
FOR_D ;                ;           ; SEQ = nbits
      ; INC SORR R10   ; QS        ; PEADDR = ALU
DJMP_S; INC SORR R11   ; SQ        ; PEADDER = ALU
function_end
```

The overall result of using the RPC mechanism is that the addressing of variables is shifted from having to be known at compile time, and built into the macros, to that of the Transputer supplying them in the RPC parameter list. However, the functions of `get_arg()` and `function_end` add an overhead of 3n+1 cycles (n = the number of arguments) when compared with the macro definition. However, only 10 instruction words are required to store the subroutine, in comparison to 6 words per use of a macro. This is summarised in Table 4.5 for an n-bit addition operation.

	Macros	RPC
Number of instructions	3*nbit+3	3*nbits+11
Code size	(no. of uses) * (3*nbit+3)	3*nbits+11
Transputer overhead	Low	High

Table 4.5 - Comparison of the use of macros and RPC calls.

An additional consideration is the amount of time spent by the Transputer in providing the RPC calls. In order to minimise this overhead the subroutine size must be sufficiently large as to free the Transputer for a significant percentage of the total processing time available. To date, the functions performed on the DAP SIMD array have been defined in terms of a set of macros and called within a single RPC frame.

4.4.3 Pyramid C++

The problem of programming a parallel machine is enlarged when two separate programming languages are required for the programming of the separate parts (the SIMD and MIMD within the WPM). A solution using a single language was found by the use and slight modification of the C++ programming language, termed Pyramid C++. The implementation of this language has been described in detail by Vaudin [Vaudin91].

C++ is an object oriented language programmed through the definitions of classes (not to be confused with the CLuster ASSembler - CLASS). A class contains a list of variables which are to be used to store an object's data and a list of functions which are the operations which will act on the data. Only the functions defined in the class (called member functions) can access the data of the object, ensuring encapsulation of the object.

An Image class has been defined in Pyramid C++ which allows images to be created in the same way as other objects, but implemented in such a way as to distribute the data across the SIMD processor array. When an operation is performed on an Image object, an 'invisible' RPC call to the controller for the operation to be done on the

SIMD array takes place. The difficulties associated with the separate call for an SIMD operation has now been removed - it is performed automatically by the C++ operation.

The operations that can be performed on the Image object is limited to those specified in the definition of the Image class. As a default, all basic arithmetic and logical operations are provided in this class, in addition to data movements across the array. If a definitive list of operations for the SIMD array is available all could be implemented within the Pyramid C++ Image class requiring no further micro-coding on the SIMD array and controller. However, to date only a subset of operations are implemented. Therefore the use of the Pyramid C++ is limited to these basic operations. A simplified definition of the Image class is given below containing functions for arithmetic and shifting operations. The full Image class definition contains a larger list of operations.

```
class Image{
public:
        Image();
        Image& operator+(Image&);
        Image& operator-(Image&);
        Image& operator*(Image&);
        Image& operator/(Image&);
        Image& S();
        Image& N();
        Image& E();
        Image& W();
}
```

The use of the Image class is illustrated below. Four images are defined, each of size 128x128 pixels and with a word length of 8 bits (ByteImage). Note that the data contained within the Images is stored on the SIMD array but can be manipulated just like ordinary variables, their implementation being hidden from the programmer. The first two images are two input frames and the second two are temporary images. The operation that is performed on the first image, frame1, is a shift North. The second operation performs an arithmetic addition between the two images frame1 and frame2.

```
main()
{
        ByteImage frame1(128,128), frame2(128,128);
        ByteImage temp1(128,128), temp2(128,128);
        temp1 = frame1.N();
        temp2 = frame1 + frame2;
}
```

The Image class also contains constructors and destructors - routines that are called when an object is first declared and when the object is destroyed respectively. These two routines can provide dynamic storage capability for the SIMD processor array. However such a capability adds to the overhead of the RPC calls to the SIMD array but this must be offset by the ease in which the system may be programmed within a single language.

4.5 Performance of the WPM

The peak performance of a single Cluster is simply the summation of the peaks on each of its constituent levels. The Cluster has a peak execution rate of 2.5×10^9 logical operations per second on the DAP SIMD array, with the ALU within the controller performing a further 10 million 16-bit instructions per second. Table 4.6 shows the time taken and throughput of some basic arithmetic operations on a single Cluster. The performance figures increase linearly with the number of Clusters existing within the overall machine.

The timings for the arithmetic operations on the SIMD array are given for two data mappings; the first is where one data item is mapped to each PE (Matrix) and the second is where one data item is mapped across a row of 16 PEs (Vector). Vector mode is supported by a set of instructions allowing vector addition (DAP instruction group 13). The floating point operations have not been implemented on the WPM but the figures are taken from performances published for the DAP [Beal90]. This is assumed to be valid since the same functionality exists within the SIMD arrays of the WPM and of the AMT DAP machine (except for DAP group 6 instructions which are used in Vector mode).

Communication and memory bandwidths are also given in Table 4.6. It can be seen that there is a general reduction in both sets of figures when going from the SIMD array to the Transputer, reflecting the reduction in volume of the data in the transformation of an image to a symbolic representation.

Operation	Cycle Count	Single Cluster (16x16 SIMD array)
Arithmetic (Matrix)		MOPS
1 bit Logical	1	2560
8 bit Integer addition	24	107
8 bit integer Multiply/Accumulate	192	13
Controller 16-bit scalar	1	10
Arithmetic (Vector)		MOPS
8 bit addition	6	27
8 bit multiplication	83	2
Floating Point (Matrix)		MFLOPS
32 bit addition	860	3
32 bit multiplication	1280	2
32 bit division	1710	1.5
Associative (within Cluster)		MOPS
broadcast	2	1280
response	2	1280
Some/None	1	2560
count	3	853
Memory Access		MByte/sec
DAP memory	1	320
Controller (Dual Port)	2	10
Transputer (inc. Dual Port)	3	6.6
Communication		MByte/sec
DAP shifting	1	320
Transputer T800 (serial links)	-	10

Table 4.6 - Cluster peak computational, memory, and communication performances.

The DAP SIMD array or the Transputer T800 MIMD arrays can be replaced by different parts as the respective technology advances. For instance, the replacement of the T800 Transputer with the T9000 would lead to a greater capacity in the MIMD array (approximately a ten times improvement in computational and communication abilities) and the addition of the DAP co-processor would improve its floating point capabilities by a factor of five [AMT90]. However the implementation of the prototype WPM has used available parts to demonstrate its concepts.

The peak performance allows an approximate comparison to be made with other parallel architectures, but in practice is a poor performance indicator. Other factors such as compiler optimisation, overheads in communication or synchronisation and poor utilisation of the array may lead to poor performance on a particular type of algorithm or application. Other comparison metrics are in common usage such as computation efficiency and traffic [Simmons89] which provide indicators on how well a particular algorithm is utilising the features of an architecture. Specific algorithms have also been defined for this purpose taking the form of benchmarks, the results of which form a sounder basis for a relative comparison [Chambers92].

4.6 Summary

The design and implementation of the Warwick Pyramid machine has been detailed in this chapter. The peak performance of this machine is the peak of both of its SIMD and MIMD arrays, which work concurrently. The partitioning of the control and associative response across the SIMD array enables both global operations, when M-SIMD arrays are synchronised, and local operations within Clusters. The programming of the WPM, using the Cluster assembler and the Pyramid C++ languages, has also been described.

The advantages of the local autonomy and partitioned associative response capabilities that the WPM architecture has over that of the conventional SIMD arrays are quantified in the next chapter. Representative image analysis and numerical algorithms are used to illustrate these advantages. Target tracking is used in Chapter 6 to illustrate the different data forms that can arise in an image analysis system and how these can be efficiently mapped onto the different layers within the WPM. Finally, in Chapter 7, the options available to efficiently utilise the operational autonomy within the WPM, by the use of load-balancing techniques, are discussed.

Chapter 5

Mapping and Processing Data on the WPM M-SIMD Array

5.1 Introduction

The way in which data is mapped across a parallel machine affects the speed-up that can be achieved over a serial implementation of an algorithm. The speed-up increases in general with the difference between the sizes of the parallel processor, and the data set to be processed. For example, the intuitive mapping of data from an image sensor (a 2D pixel array), is the mapping of one pixel on to each PE, arranged spatially as in the original image. This however, assumes that the PE array is of the same size as the image and has the same topology, i.e. a two dimensional PE array. For SIMD arrays this situation is feasible and indeed offers, for many applications, real-time processing. However, a programmable array has to cope with a multitude of image sizes from a variety of sensors.

The organisation of the lower level of the WPM, as M-SIMD, affects the most efficient data mapping across the SIMD processors. The partitioned M-SIMD array approach results in a number of computational differences over that of existing conventional SIMD arrays. The increase in operational autonomy can result in an increase in algorithmic performance and additionally, the partitioned associative response mechanism allows one response operation per Cluster in parallel. However, the same partitioned array can also result in increased communication (shifting) cost when all, or a subset of, Clusters operate in their synchronised mode (using the hand-shaking mechanism discussed in Section 4.3.4) over that of a conventional SIMD machine.

This chapter details the performance advantages that M-SIMD offers over conventional SIMD arrays in a number of applications. The contents include :

- a comparison between a conventional DAP SIMD array and the WPM SIMD array. This is detailed in Section 5.2 and is described in terms of their local autonomy, associative responses, associative count, and data communication.
- a description of the ways in which data can be mapped onto an SIMD array and an M-SIMD array. This is given in Section 5.3.
- the use of image processing operations on the M-SIMD array, described in Section 5.4. Included in this is the use of the Clusters associative count.
- a description of the mapping of numeric matrix data for use in standard matrix operations. The performance of these operations is detailed in Section 5.5.

5.2 The Advantages of M-SIMD over SIMD

The cost of the additional hardware for the M-SIMD array of the WPM, over a conventional SIMD array, is in providing separate instruction streams within each SIMD patch. The performance gains (or losses) that result from the additional hardware can be seen by examining the local autonomy gain and the partitioned associative response mechanism. Such considerations provide a bound on the computational gains that can be achieved at an algorithmic independent level. The effect of the local autonomy and the associative networks within the WPM form the basis of the possible performance increases over that of conventional SIMD architectures in any application domain.

The performance differences, between the WPM and a conventional DAP SIMD array, are described below. They are compared in terms of the differences in local autonomy, associative response, count response, and data communications.

5.2.1 Local Autonomy

The local autonomy, available within each Cluster of the WPM, has full effect when independent processing is performed on separate regions within the image data processed. This is sometimes referred to as region of interest processing in image analysis and as object processing in the target tracking domain. The independent region processing is typically only a component of an image analysis processing flow. The performance advantages on an M-SIMD array are only applicable to this component of the total processing.

The size of each SIMD patch, within an M-SIMD array, also effects any such achievable performance improvements. Figure 5.1 shows the effect on the performance of M-SIMD architectures of the region size being processed. Two configurations of M-SIMD architectures are considered; the first is where each M-SIMD patch, from a square processor array, can be configured to be the same size as the image regions being processed (an ideal M-SIMD configuration); the second considers the WPM M-SIMD array, with its fixed 16x16 SIMD arrays. In both arrays, it is assumed that the total number of SIMD PEs is equal to 128x128. In the case of the WPM, this is 8x8 Clusters.

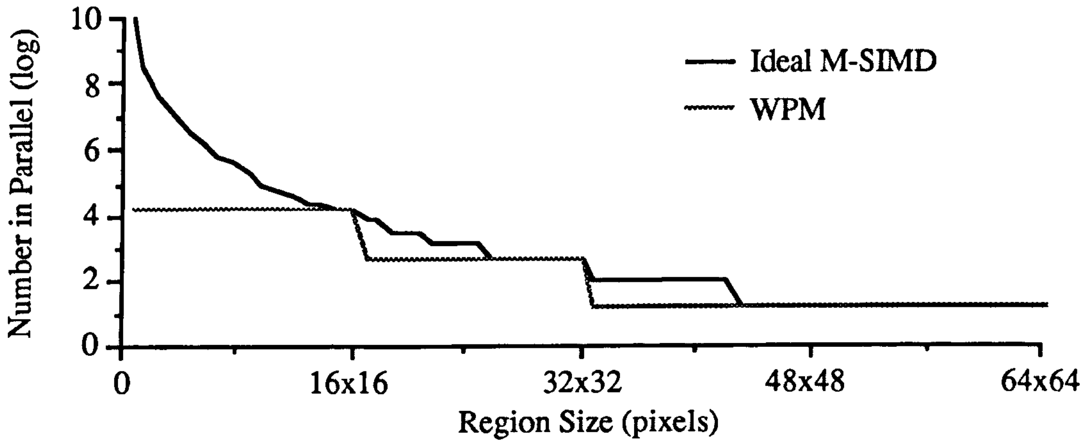


Figure 5.1 - Number of regions that can be processed in parallel on an ideal M-SIMD architecture and on the WPM M-SIMD array.

The graph of Figure 5.1 shows the number of image regions, of a given size, that can be mapped across either type of M-SIMD array. These are mapped such that no two regions lie within the same SIMD patch, i.e. it is the number of regions that can be processed independently at any one time. The number of regions mapped across the M-SIMD array represents the performance increase achievable with local region of interest processing. Note that a natural log scale is used in Figure 5.1.

Note that for the conventional SIMD architecture, with only a single instruction stream, the number of regions that can be processed in parallel is only one - the x-axis in Figure 5.1. The ideal M-SIMD array has the maximum number of regions that can be mapped across it, through the use of its reconfiguration capability. However, on the WPM the same number of regions are mapped across the array for region sizes equal to or less than 16x16 pixels. The same is true for region sizes greater than 16x16 pixels and less than or equal to 32x32. The cost of having the reconfiguration in the ideal M-SIMD architecture is high, one controller per PE in the worst case (an MIMD architecture). This is unrealistic to implement.

The configuration of the WPM represents a compromise between hardware complexity and performance. It achieves an optimum performance increase when the size of the regions being processed are equal to each M-SIMD patch size (16x16 PEs) or a multiple of it. For regions with other sizes there is a loss in possible performance gain. This can be seen more clearly in Figure 5.2. This shows the percentage efficiency of both the WPM and a conventional SIMD array when compared with the ideal M-SIMD architecture. The efficiency is taken as being the ratio of the number of regions mapped across the WPM, or SIMD array, to the number of regions mapped across the ideal M-SIMD array. Note that a conventional SIMD array has poor utilisation when the object is small, but this gradually increases to 100% when the object covers the whole array.

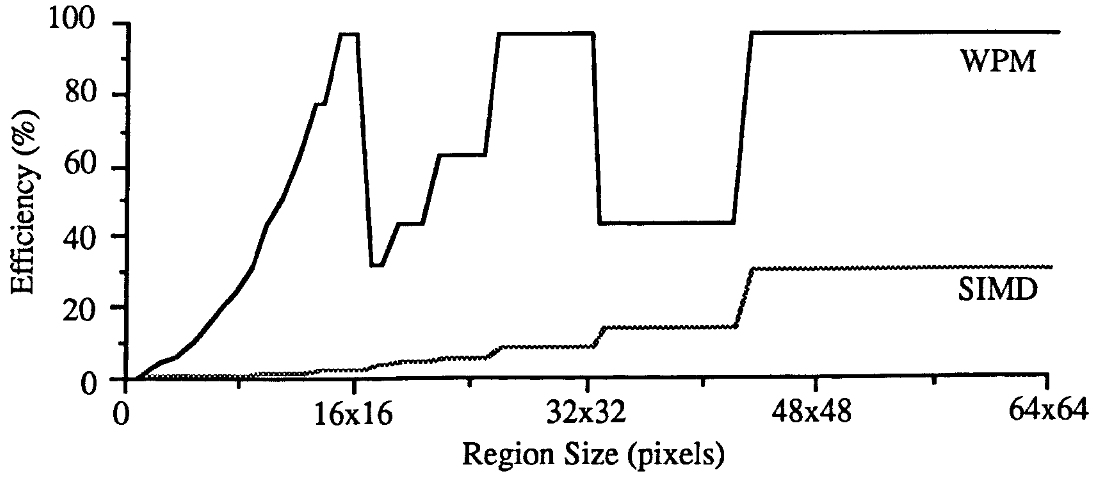


Figure 5.2 - Efficiency of the WPM and SIMD array for varying region sizes.

The efficiency of the WPM is a maximum when the region sizes are multiples of the SIMD patch size (16x16). However this comparison does not include the synchronisation and communication costs in the WPM which exist when a region is mapped across a set of SIMD patches.

5.2.2 Associative Response Operations

In addition to the autonomous control of each SIMD patch within the WPM, independent associative operations can be performed. The associative operations can take the form of the Logical AND of all columns (or rows) across the SIMD patch - to find the position of the first responder for example. Or it can be used to broadcast a value across an SIMD patch.

The multiple associative response mechanisms, within the WPM, can be used in parallel within each SIMD patch. This results in an increase in performance for such operations, equal to the number of Clusters within the M-SIMD machine. The results of the associative response may be used either within the Cluster they originated from, possibly causing data dependent conditional instruction branching, or be broadcast back to the SIMD patch. Alternatively, they can be placed within the Cluster's shared memory for further processing by the MIMD processor.

Similar performance efficiencies occur for the WPM, when compared with the ideal M-SIMD architecture, for associative response operations to that of the operational autonomy, as shown in Figures 5.1 and 5.2 for different sized regions. However, the effects are more marked when the region is larger than the SIMD patch size, due to communication overheads between SIMD patches. In this case the accumulation of partial associative responses between SIMD patches are required.

The accumulation of the results, within the WPM, can be performed on either the M-SIMD array, or on the MIMD array after passing results to the Cluster's shared memory. Accumulation on the M-SIMD array is carried out by broadcasting the associative responses back to the SIMD patch, shifting and reading this on the neighbouring Cluster and then doing an accumulation with its own value by using the associative response mechanism again. On the MIMD processors, the associative results are asynchronously communicated to neighbouring processors for accumulation, in parallel with further operations on the M-SIMD array. The accumulation process in either case can use a distance doubling technique (accumulating neighbouring results followed by every second neighbour and so on), until the accumulation has covered the whole region concerned.

For an object covering N^2 Clusters, $2\lceil \log_2 N \rceil$ accumulation and 2(N-1) shifting operations are required. On the M-SIMD array the total time taken for this operation is $10\lceil \mathrm{Log}_2 N \rceil + 4(N-1)$ cycles, ignoring the synchronisation time between Clusters. On the T800 Transputers with 20Mbit/s link speed, the time taken is dwarfed by the communication time of $\approx 4(N-1)$ µs. The communication time on the Transputer is approximately ten times slower than on the WPM M-SIMD array, but can be performed concurrently with the M-SIMD array.

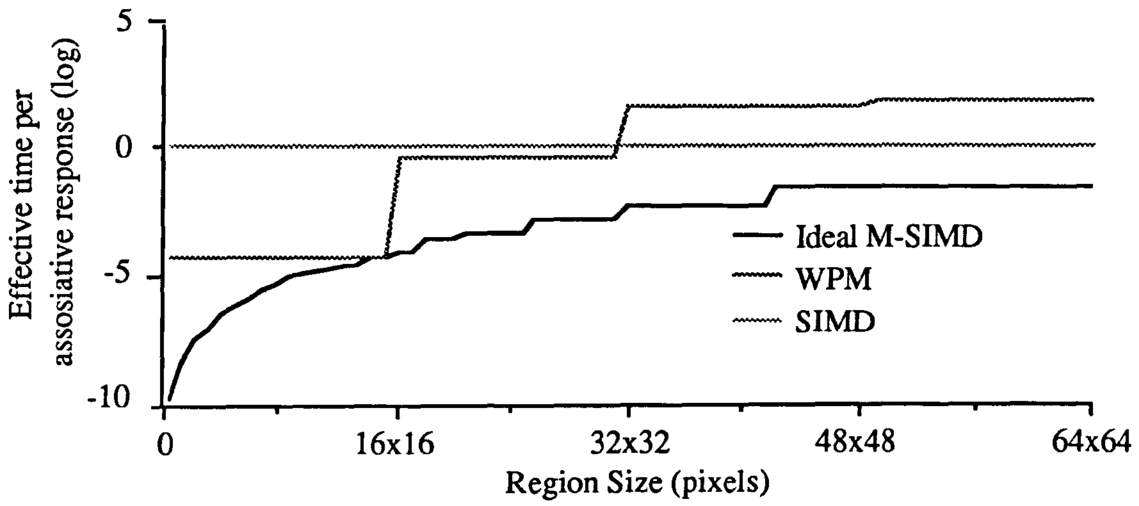


Figure 5.3 - Comparison between the ideal M-SIMD architecture, the WPM M-SIMD array and an SIMD array for associative response operations.

The performance of associative responses on the WPM, an SIMD architecture and the ideal M-SIMD architecture is shown in Figure 5.3. This shows the effective time per associative response for varying region sizes. The effective time is the time taken for the operation divided by the number of associative responses that take place in parallel. The conventional SIMD architecture can only do one such operation at a time. The M-SIMD machines can complete many operations in a single cycle dependent upon the region size. The performance of the ideal M-SIMD architecture reduces to that of the SIMD architecture when only one region can be mapped across it.

Again, the WPM achieves optimum performance on regions of size 16x16, the size of each Cluster SIMD array, but is less effective than the SIMD array when the regions are larger than 32x32. This is due to the overhead involved in accumulating partial results across Clusters. This effect can be seen when the regions exceed the boundaries of one, two and four Clusters.

The relative performance of both the WPM M-SIMD array, and the conventional SIMD array, against that of the ideal M-SIMD architecture is shown in Figure 5.4. The peak for the WPM shows the case when the regions are the same size as the SIMD patches (16x16) giving an optimum performance. The SIMD array also gives the same performance as the ideal M-SIMD array when only one associative operation can be performed at any one time, i.e. when the region size if greater than 64x64.

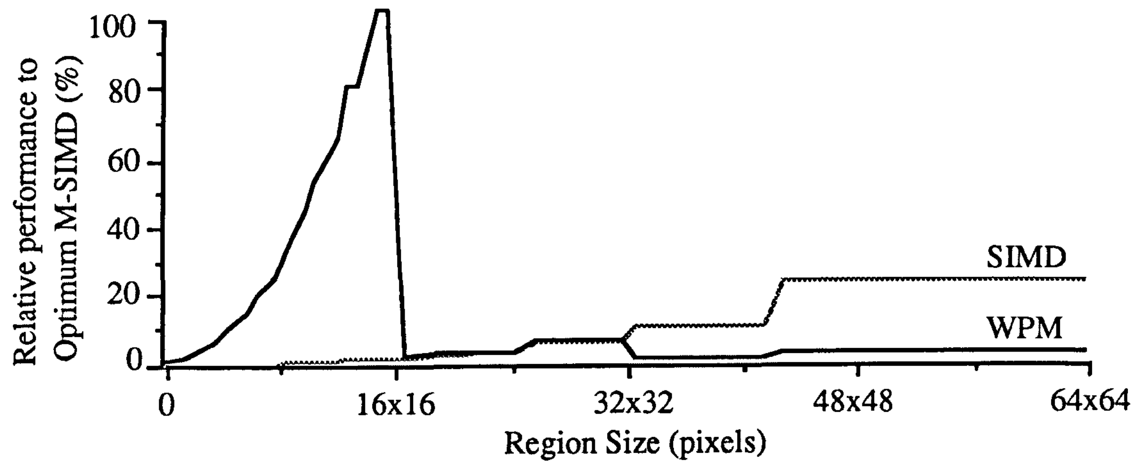


Figure 5.4 - Relative performance of the WPM M-SIMD array, and an SIMD array, with the ideal M-SIMD architecture for associative operations.

The poor performance achievable on the WPM for associative operations, when the regions are not equal to the Cluster SIMD array size, should not be considered in isolation. Other factors that should also be considered when giving the overall performance of the WPM, include the frequency with which such operations are required, the proportion of time allocated to such operations and other benefits that the organisation of the WPM provide.

5.2.3 Count Response

The associative count response mechanism at the M-SIMD level, within each Cluster in the WPM, greatly improves the performance of counting operations. It enables previously unconsidered algorithms to be used for various operations (see Section 5.4.2). On the WPM, a count takes 3 cycles within a Cluster, reducing to a single cycle when using the counts pipelined design. Note that global counts are accumulated in a similar way to associative responses as described above.

A conventional SIMD array performs the count of the number of bits set in a bit-plane by using a distance doubling technique. This is done by firstly adding together neighbouring bits, followed by every second neighbour, every fourth and so on, but noting that the word size of the data increases by one bit after each addition. The time required for this operation is :

$$\sum_{n=1}^{2\log_2 N}(3n+1) + \sum_{n=1}^{\log_2 N} n.2^{(n-1)} + \sum_{n=\log_2 N}^{2\log_2 N} n.2^{(n-1-\log_2 N)}$$

where N^2 is the size of the SIMD array. The first term is the time taken to perform the additions, and the second two represent the communication time. The communication time is split into two parts because the distance doubling is performed in a separable fashion for both dimensions of the SIMD array.

A comparison between the time taken to perform the count on the WPM, and the time taken on a conventional SIMD with no additional hardware, is shown in Figure 5.5. The graph shows the time taken on a total SIMD array size of 128x128 for varying region sizes. The additional hardware in WPM out performs that of the conventional SIMD array by between one and two orders of magnitude.

In some applications it may be necessary to count several bit-planes at the same time (e.g. in histogram calculations). On a conventional SIMD array, a certain amount of parallel computation is possible during the distance doubling, whereas on the WPM the time increases linearly with the number of bit-planes to be counted. However, the count within the WPM will still out perform the possible parallelism on the SIMD array although the performance difference is reduced. For the count of 256 bit-planes the performance of the WPM is approximately three times greater than that of SIMD.

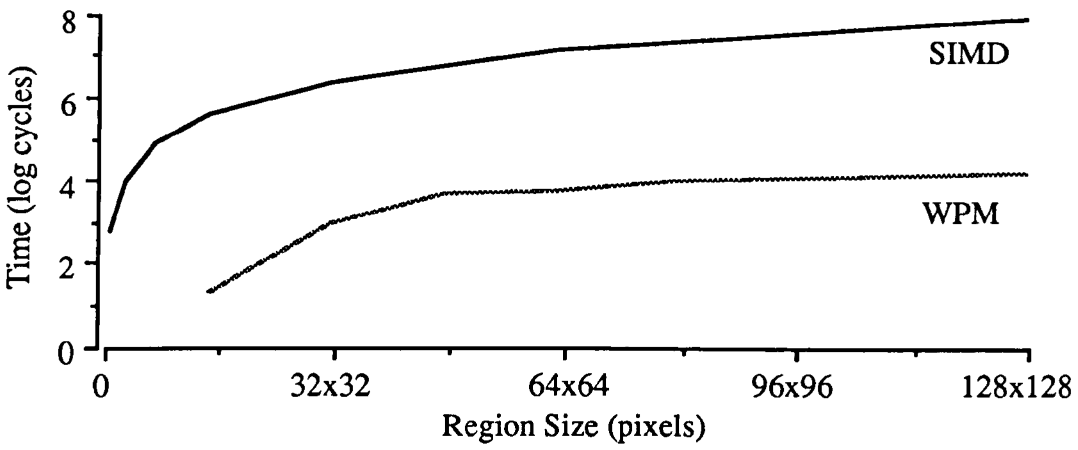


Figure 5.5 - The time taken to count the number of bits set in a single bit plane on both an SIMD array and on the WPM.

5.2.4 Data communication

Communication between Clusters at the M-SIMD level, within WPM, uses a handshaking mechanism adding an extra cycle per shift (ignoring the synchronisation time). This implies that the cycles required to shift the data by a distance of n PEs is (2n+2) cycles on the WPM compared with (n+2) cycles on a conventional SIMD array. The extra two cycles are for a memory read at the start of the shift, and a memory write at the end of the shift. However, the WPM performs better on longer shifts when compared to the conventional SIMD array as follows.

If the shifting distance required is greater than 31, then the internal SIMD torus network within each WPM Cluster can be used to effectively by-pass that Cluster. Consider three Clusters - A, B and C undergoing a shift in the direction of A to C (assumed West to East). The data originates inside A and it is required to shift all the data to C - a distance of 32 PEs. Initially, the first column of this data is shifted east into B, B then does a shift west while in Cluster mode (see Section 4.2), using its internal SIMD torus network, shifting the data to its east-most column. On the next and subsequent shifts east, data is again shifted into B from A but also into C from the last column of B.

The result is two different types of shift, the first (a type one shift) by-passes a Cluster and takes three cycles for each bit of the data. The second (a type two shift) performs the last shifts, ensuring that the data lies in the correct position within the final Cluster. This take 32 cycles per data bit. These times assume that the shifts are performed in units of 16 PEs, however the general case is shown in Figure 5.6 for an arbitrary shift. It can be seen that the WPM out-performs a conventional SIMD array when shifting data a distance of 64 PEs or more.

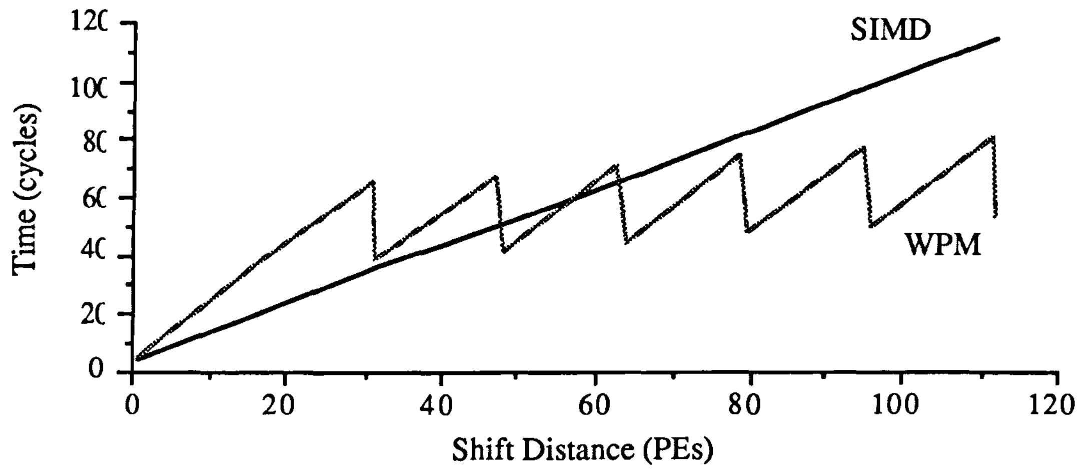


Figure 5.6 - The time taken to shift data across the M-SIMD array within the WPM, and across an SIMD array.

5.3 Mapping Data onto the M-SIMD Array

It is often the case that the data to be processed is larger than the processor array being used. This is especially true with image data. In such situations, consideration must be given to how the data should be mapped across the processor array. There are two main mapping strategies commonly employed: sheet mapping, and crinkled mapping. Both mappings result in one or more data items mapped to each PE. The mapping techniques are general purpose, irrespective of processor array type and are described below, and also by Reddaway [Reddaway88] and Liddell [Liddell87].

The size of the processor array and the image data is generalised for the following discussion, assuming only that they are both square for simplicity. The processor array size is denoted as N^2 with each PE referred to as P_{ij}, and the image size as m^2 with each pixel referred to as I_{ij}. The symbol '$\Rightarrow$' is used to denote the mapping of a specific image pixel onto a specific PE. Using this terminology, for the simplest case of the processor array equal in size to the image, $N = m$, and $I_{ij} \Rightarrow P_{ij}$.

5.3.1 Sheet mapping

In this mapping, the image is divided into array-sized patches or sheets which are then placed onto the processor array, so that each PE has a pixel from each of the sheets. When an individual sheet of the image is addressed on the processor array, each PE operates on a pixel within the same sheet. Sheet mapping can thus be expressed as :-

$$I_{ij} \Rightarrow P_{(i \bmod N)(j \bmod N)} \qquad \text{where } i = 1..m, j = 1..m$$

A total number of $\left\lceil \frac{m}{N} \right\rceil^2$ sheets result from this mapping. N.B. the upper bound, denoted by $\lceil\ \rceil$, is required in this expression to cope with the case when the image size is not an exact multiple of the array size. Sheet mapping retains the spatial arrangement of the image pixels, within each sheet, across the processor array so that shifting can be used to access neighbouring pixel values. However, discontinuities exist at the boundaries of each sheet, requiring additional processing with neighbourhood communications. This adds to the communication costs in most algorithms. However, sheet mapping allows local operations to be performed on a subset of the image, corresponding to a single sheet (or sheets) of the mapping. Thus, only the relevant sheets need to be processed, reducing computation time in a manner suggested in Section 5.2.1 for an M-SIMD architecture.

5.3.2 Crinkled mapping

With crinkled mapping, the image is divided into as many patches as there are PEs. Each patch is mapped to a separate PE, thus preserving locality between all image patches on the processor array. When the image is addressed on the processor array, each PE operates on a pixel within its image patch, e.g. the top left of each image patch. This mapping is expressed as

$$I_{ij} \Rightarrow P_{(i \text{ div } \lceil m/N \rceil)(j \text{ div } \lceil m/N \rceil)} \qquad \text{where } i = 1..m,\ j = 1..m$$

In this mapping, neighbouring pixels within an image patch are mapped to the same PE. This means fewer neighbourhood communications are required on SIMD shift operations. However, only global operations can be performed and each PE will operate on a pixel from each image patch across the whole image. There is no performance gain for region processing in crinkled mapping - it is the same as if the whole image is being processed. In addition, temporary storage for intermediate results may be needed for the entire image during processing, unlike sheet mapping where memory is only required for the current sheet being processed.

5.3.3 Comparison of Data mappings

A comparison of both sheet and crinkled mapping is shown in Figure 5.7. In both mappings the total number of pixels held within each PE is $\left\lceil \frac{m}{N} \right\rceil^2$. The properties of each mapping, along with the overhead involved in shifting data across the processor array, is shown in Table 5.1. The time for a single shift, a copy and the amount of temporary memory required per pixel, are denoted as T_{shift}, T_{copy} and M_{temp} respectively.

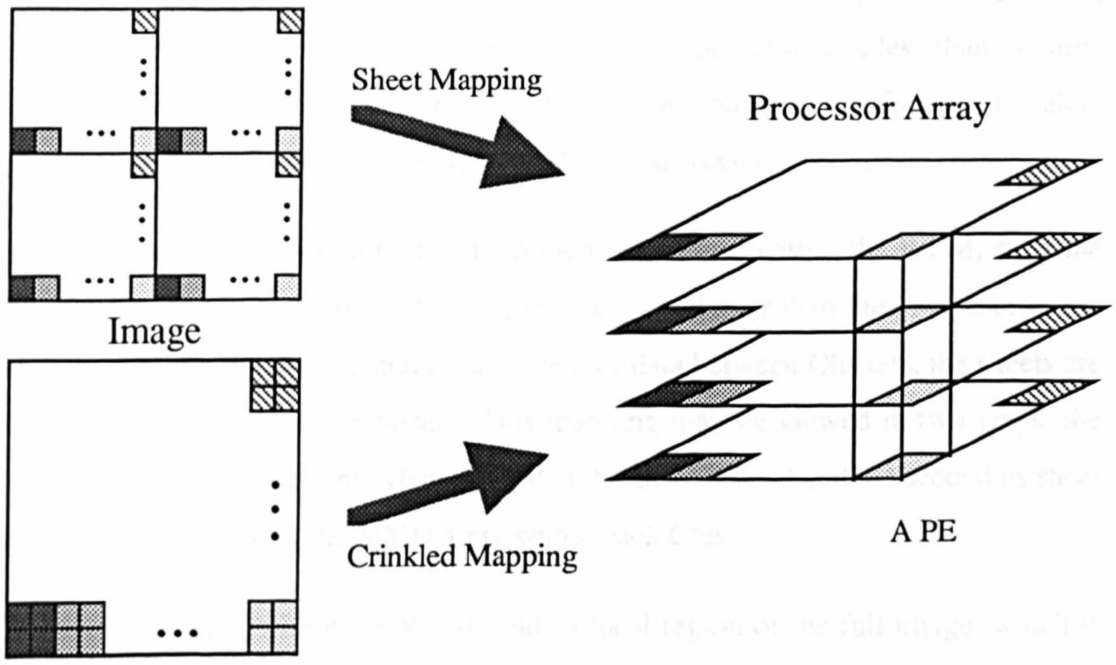


Figure 5.7 - Sheet and Crinkled data mappings on an array processor.

For data shifting, each of the $\lceil \frac{m}{N} \rceil^2$ sheets in sheet mapping need to shifted separately. In crinkled mapping, most of the neighbouring pixels are contained within the correct PE, thus requiring local memory addressing (which has no extra time cost) and the shifting time is proportional to $\lceil \frac{m}{N} \rceil$. Additionally, in sheet mapping each of the sheet boundaries require further manipulation, by using the wrap-around facility of the processor array. Data is shifted to the correct destination PE, but data at the edge of the array have to be copied into the correct sheet.

	Sheet mapping	Crinkled mapping
Local Operations	Good	Poor
Amount of Temporary Storage	$1 * M_{temp}$	$\lceil \frac{m}{N} \rceil^2 * M_{temp}$
Neighbourhood shifting (cycles)	$\lceil \frac{m}{N} \rceil^2 * T_{shift}$	$\lceil \frac{m}{N} \rceil * T_{shift}$
Sheet boundary processing (cycles)	$\lceil \frac{m}{N} \rceil^2 * T_{copy}$	none

Table 5.1 - Characteristics of Sheet and Crinkled mappings.

There is a trade-off between the two mappings in terms of memory requirements, communication costs, and local region processing. Sheet mapping should be used if the time for communications, plus the time for local operations, is less than the time for the same communications in the crinkled mapping, plus the time for the equivalent global operations. Crinkled mapping should be used when the converse is true.

The local autonomy available, for the region processing within the WPM, suits the sheet mapping characteristics of an image when it is larger than the processor array. However, to preserve the spatial arrangement of data between Clusters, the sheets are crinkled mapped between Clusters. This mapping may be viewed in two ways, the first being crinkled mapping when viewed at the Cluster level and the second as sheet mapping when viewing the SIMD array within each Cluster.

Thus, each Cluster within the WPM holds a local region of the full image, which if necessary is divided into sheets. These can be processed independently in local operations. In the following examples in Sections 5.4 and 5.5, it will be assumed that the processor array size is the same as the image size for simplicity. When the images are larger than the array size the performance of such operations can be scaled, and the communication costs of the mapping techniques added, as discussed above.

5.4 Image Operations on the M-SIMD Array

The performance of the M-SIMD array on several image processing operations is considered below. The first is a global filtering operation which is performed in a similar way on either an SIMD array or an M-SIMD array. An example filter is shown for the Sobel edge detection filter which can be decomposed into filters of smaller size. The use of the WPM associative count is also described for a number of image operations.

5.4.1 SIMD filtering operations

A common operation in early image analysis applications is that of a filtering operation in the form of a two-dimensional convolution. Examples can be found in [Schalkoff89, Pratt78] for template matching, enhancement (e.g. smoothing) and edge gradient operators (Sobel, Laplacian). All the filtering operations take the same mathematical form. For a two-dimensional convolution this is :

$$F(i,j) = W * I(i,j)$$

where I is the input image, F the output image, '*' is the convolution operator, W is the convolution weighting function commonly referred to as the convolution mask and i,j indicate the image pixel. Each output pixel is a function of the input image pixels, around the same location, weighted by the convolution mask. The filtering operation can be re-written as a summation in each of the convolution mask dimensions :-

$$F(i,j) = \sum_{p=-R}^{R} \sum_{q=-S}^{S} W(p,q) \,.\, I(i-p, j-q)$$

where R and S specify the size of the convolution mask.

On an SIMD array processor the convolution is very easily performed, since the same operation can be performed for each image pixel in parallel. The processing consists of a multiply-accumulate operation for each of the mask coefficients, thus shifting the resulting accumulation to the position of the next mask coefficient location. Then performing the next multiply-accumulate, and so on until all of the mask coefficients have been used. Note that the word-size of the data increases both with the multiplication (8bit data increases to 16bit) and with the accumulations. It is assumed below that the data remains at 16bits after the multiplications.

The whole operation thus consists of a series of multiply-accumulates and shift operations. The efficiency of the filter operation depends upon the convolution path taken while shifting and it must be ensured that the final result ends at the centre of the mask. It was shown by Lee [Lee87] that for rectangular masks there is an optimum shortest convolution path. For square masks the path starts at a corner point of the mask and spirals in to its centre. The shortest path for a general 5x5 convolution mask is shown in Figure 5.8a.

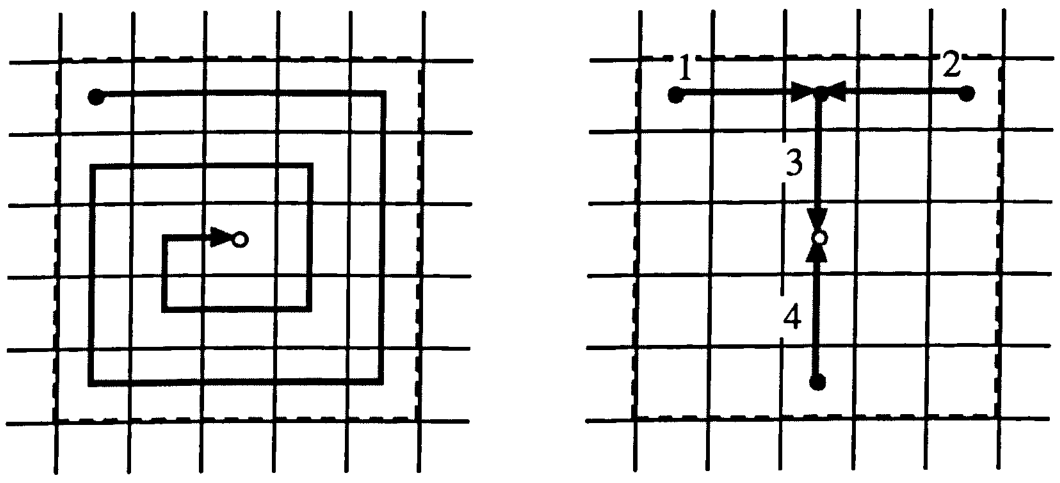


Figure 5.8 - Convolution paths for a 5x5 mask on an array processor.

Fewer multiply-accumulate and shifting operations are required if the mask is separable, i.e. when the two-dimensional convolution can be formed from two one-dimensional convolutions, such as the Laplacian [Nudd89]. This reduces the complexity of the convolution path from $O(n^2)$ to $O(n)$ and can be seen in Figure 5.8b. Note that the path taken is split into four parts requiring a total convolution path equal to two one-dimensional convolutions of size 1x5.

The time taken to perform non-separable and separable convolutions on the WPM is given in Table 5.2. This uses the multiply-accumulate time, T_{macc}, for 8-bit data, with known multiplicand values (convolution coefficients are usually known at compilation time), and the shifting time for 16-bit data, $T_{shift} = 67$ cycles. The time taken increases in proportion to the mask size. The worst case given is when all the convolution coefficients consist of a maximum number of bits set, requiring a maximum number of additions in the multiplication ($T_{macc} = 276$ cycles). The average case is when the convolution coefficients consist of an average of four set bits ($T_{macc} = 164$ cycles).

	Non-separable (cycles)	Separable (cycles)
nxn General mask	$n^2T_{macc} + (n^2-1)T_{shift}$	$2nT_{macc} + 2(n-1)T_{shift}$
3x3 General mask (average)	2012	1252
(worst)	3020	1924
7x7 General mask (average)	11252	3100
(worst)	16740	4668

Table 5.2 - Time taken for convolution operations on the WPM.

It can be seen from Table 5.2 that the time taken to perform a single 7x7 mask is already in the order of milli-seconds. Thus, it is advantageous to try and optimise the convolution masks to minimise the number of multiplications used within the convolution operation. Simple convolutions are shown in Figure 5.9 for a 3x3 smoothing filter and a 3x3 high pass filter. These masks require only additions and subtractions for each mask coefficient, reducing the time taken for the convolution, and in the case of the smoothing mask, is also separable. Note that the coefficient value of eight, in the high pass filter, can be achieved by adding the middle value, shifted by 3 bits relative to the accumulation value, so as to appear 8 times larger. Such shifts require a change of addressing only and have zero time cost.

$$\begin{array}{|c|c|c|}\hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline \end{array} \qquad \begin{array}{|c|c|c|}\hline -1 & -1 & -1 \\ \hline -1 & 8 & -1 \\ \hline -1 & -1 & -1 \\ \hline \end{array}$$

Figure 5.9 - 3x3 convolution masks for smoothing and high-pass filtering.

One of the simplest gradient filters is the Sobel operator. This consists of two 3x3 masks, one for the horizontal gradient, G_x, and one for the vertical gradient, G_y. A brief description of the foundations of the Sobel operator is given by Danielsson in his appendix [Danielsson90]. The gradient at a particular point is given by the square root of the sum of these two gradients squared, but is sometimes approximated by taking the sum of the modulus of the two gradients - known as the city-block form. The mask for G_x is given in Figure 5.10 which also shows its conventional decomposition into two one-dimensional convolutions. The vertical gradient mask, G_y, is a 90° rotated version of the horizontal mask, G_x.

$$G_x = \begin{array}{|c|c|c|}\hline 1 & 0 & -1 \\ \hline 2 & 0 & -2 \\ \hline 1 & 0 & -1 \\ \hline \end{array} = \begin{array}{|c|}\hline 1 \\ \hline 2 \\ \hline 1 \\ \hline \end{array} * \begin{array}{|c|c|c|}\hline 1 & 0 & -1 \\ \hline \end{array}$$

Figure 5.10 - The horizontal Sobel gradient operator.

The Sobel mask can be decomposed further into a set of very simple convolution masks, consisting only of neighbourhood additions, as described by Danielsson [Danielsson90]. This decomposition, in addition to reducing the computational requirements even further than that shown in Figure 5.10, suggests a set of generalised gradient operators based around the original Sobel operator but with increased mask sizes.

The decomposition of the Sobel, into a set of neighbourhood additions, is shown in Figure 5.11. It consists of smoothing convolutions followed by gradient convolutions for each of the gradients G_x and G_y. Larger Sobel like masks are obtained by adding further sets of smoothing convolutions to the front of the sequence in Figure 5.11.

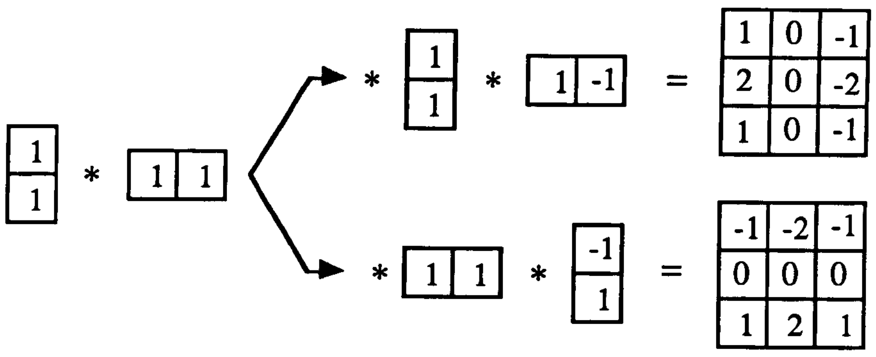


Figure 5.11 - The Sobel operator decomposed into neighbour additions.

The total number of additions and subtractions required using this decomposition is 2n, where nxn is the mask size. Similarly a total of 2n communications are required. The times taken for the Sobel filter, in its conventional 3x3 form and decomposed 3x3 and 5x5 forms, are given in Table 5.3. These times also take into account the increase in the data word-length after the addition operations (from 8-bit to 9-bit after the first addition, 9-bit to 10-bit after the fourth, and so on). The times assume that the city-block method is used to calculate the final gradient to the accuracy of the final addition. The accuracy is 11 bits for the 3x3 Sobel.

Sobel (both G_x and G_y)	Time (cycles)
Conventional Sobel (3x3)	481
Decomposed Sobel (3x3)	404
Decomposed Sobel (5x5)	762

Table 5.3 - Comparison of the time taken for the computation of the Sobel filter.

5.4.2 The use of the WPM associative count

Described here are several algorithms that benefit from the use of an associative count within each Cluster of the WPM. The operations examined are those of histogram generation, rank order filtering, mean and variance, and image moment calculations. These operations occur often within image analysis applications, performed not only on grey-scale images, but also on output filtered images which remain as two-dimensional data.

The count operates only within a Cluster boundary. Consider an operation to count the area of an object which is in the form of a binary image. If the object is contained within a single Cluster then a single count operation can be performed within that Cluster and the count output stored in its the shared memory. If the object lies across Cluster boundaries then a count on each of the Clusters containing part of the object can be performed in parallel and the results accumulated as described in Section 5.2.2.

For the following operations it is assumed that the count is required only within a single Cluster. However, some operations provide a number of output values which would need to be accumulated across Clusters when the locality constraint is not imposed. In this case, the times for the algorithms would be increased by the factors described in Section 5.2.2.

5.4.2.1 Histogram generation

Histogram techniques have been widely used within image analysis, not only for grey-level segmentation [Schalkoff89], but also for further analysis such as in finding the modal image plane velocity for motion segmentation [Burt91]. Histograms are generated by counting the occurrence of each of the possible values taken by the data set.

An implementation for the generation of histograms was given by Howarth [Howarth88]. This performed a comparison, on the SIMD array, between the image data and the value being looked for, with the number of correct comparisons counted. This is repeated for all the values required. For 8-bit data, a total of 5000 cycles were required - the time being dominated by the 256 comparisons, one for each histogram value.

A more efficient method is described by Reeves [Reeves80]. This uses a theoretical model of an array count device which evaluates a Boolean expression for each of the grey levels required. For example, the grey level of 64 (using 8-bit data) is

$$I_{64} = \bar{x}_7 \cdot x_6 \cdot \bar{x}_5 \cdot \bar{x}_4 \cdot \bar{x}_3 \cdot \bar{x}_2 \cdot \bar{x}_1 \cdot \bar{x}_0$$

The intermediate results are saved, using the fact that some adjacent grey levels differ by a single bit, e.g. the value of I_{64} differs from I_{65} by just the LSB. This method can be extended, saving all intermediate results, and requires (n-2) storage bit-planes. It takes (5.2^n - 8) operations on the WPM, where n is the number of bits in the data. For 8-bit data, the total number of cycles required is 1272. Note that the time taken on the WPM differs from Reeves analysis where only (4.2^n -8) operations are required. The extra cycle per grey level required on the WPM is due to the count being attached to the DAP PE memory lines and is not an integral component of the PEs. Consequently, an extra cycle is required to output the result of the Boolean calculation on to the DAP memory lines for counting.

The calculation of the histogram can be improved further, as described by Francis [Francis91], by performing the Boolean calculations to decode each half of the data word (for 8-bit data, x_7->x_4 and x_3->x_0) and storing the results for each half. Each of the decoded results require $2^{n/2}$ storage bit-planes. The two decoded halves are combined sequentially, using an ANDing operation the result of which represents a single grey level, and can be counted. The number of operations required, for 8-bit data, is 160 cycles (for the decoding operation) and 768 cycles for the combination and counting operation.

The histogram calculation using the decoding method is generalised for arbitrary word-length data in Table 5.4 and compared with the method of Reeves. The different operation cycle counts for n odd and n even, with the method of Francis, results from having uneven lengths for half the data word when n is odd. The amount of storage required, for the method of Francis, increases as 2^n which may become a limiting factor for larger word-length data.

Histogram Generation	Reeves Algorithm		Francis Algorithm	
word-size	Cycle count	Storage	Cycle count	Storage
nbit (n even)	5.2^n - 8	n-2	$2^{n/2}(n+2)+3.2^n$	$2^{(n/2+1)}$
nbit (n odd)	"	"	$2^{(n-3)/2}(3n+7)+3.2^n$	$3.2^{(n-1)/2}$
5 bit	152	3	140	12
8 bit	1272	6	928	32

Table 5.4 - Comparison of histogram generation algorithms on the WPM.

5.4.2.2 Rank order filters

Producing a particular rank value from a local image region is a common operation in image processing. It includes finding the minimum or maximum value, within a region, or using the median value for noise reduction with edge preservation. The operation is typically performed over a region of the image in a similar way to a convolution, such that the rank is calculated in parallel at all pixel locations.

The most common way in which the calculation of a rank, such as the median, is to sort the values from each of the local regions in parallel across the processor array. The sorting of n^2 values (where nxn is the size of the local region around each pixel) takes $O(n^4)$ comparison operations plus $O(n^2)$ shifting to get all the required values within each PE. In fact, the sort can be implemented by calculating the maximum value, in $O(n^2)$ operations, and repeating until the required rank is obtained, or, if the rank is nearer the 0th rank, starting with the minimum. A total of $\mathrm{Min}[(n^2-R),R]$ iterations are required where R is the rank required. The worst time for this operation occurs for the median, which takes

$$3n^4(T_{comp}+T_{ccopy})/8 + (n^2+2n-3)T_{shift} \text{ cycles}$$

where T_{comp} is the comparison time, T_{ccopy} is the time for a conditional copy, and T_{shift} is the time for a shift - all for 8-bit data. (T_{comp} = 19 cycles, T_{ccopy} = 27 cycles, and T_{shift} = 35 cycles). This algorithm is termed the conventional method below.

An approximation to the median calculation can be made using a method known as the median of medians and is discussed by Reddaway [Reddaway85]. This involves decomposing the two dimensional region into two one-dimensional regions of size n. This requires $O(n^2)$ operations thus reducing the computation required.

The operation to find a particular rank is slow due to the effective calculation of all ranks resulting from the sort. However, a novel algorithm which calculates a particular rank, without the need to sort, was introduced by Danielsson [Danielsson81]. This algorithm works in a bit-serial fashion, starting at the MSB, and counts the number of values with a zero MSB. If this is less than the rank required then the value required must contain a 1 in the MSB. This is repeated for all bits down to the LSB of the data such that the set of remaining values contains the rank required as its maximum.

This algorithm, which is termed the iconic method below, has been implemented on the WPM [Francis91] by shifting all values, within the local mask, in to each PE. The operation was found to take

$$8*5*n^2+8\log_2 n+3 + (n^2+2n-3)T_{shift} \text{ cycles}$$

for an nxn mask size using 8-bit data, where T_{shift} is the shift time. However the counting operation can also be done on the associative count within the Cluster. The count can only be used to calculate one rank, at one pixel location, at a time and it takes 17*8 cycles per calculation. To calculate a rank for each of the PEs within the Cluster takes

$$17*8*(PE_{num}) + 255*T_{shift} \text{ cycles}$$

for a mask size of up to 16x16 using 8-bit data, where T_{shift} is the time for a shift, and PE_{num} is the number of PEs requiring a count. This operation is O(p) cycles, where p is the number of PEs within the Cluster, whereas that implemented without the count takes $O(n^2)$ - i.e. dependent upon the mask size.

If only a small number of ranks are required, from the image area within the Cluster, the use of the hardware count is more efficient. A comparison is shown in Table 5.5. However, if the rank is to be calculated for each pixel location, the iconic based method is more efficient. A further consideration is the amount of temporary storage required during the operation especially for a large mask size, e.g. for a 15x15 mask. In this case, the iconic method would require 1800 bits of storage per PE (for 8-bit data) in comparison to the count method of zero.

Method (using 8-bit data)	Cycle Count	Storage (8-bit)
Conventional (1 per PE)	$3n^4(T_{comp}+T_{ccopy})/8+(n^2+2n-3)T_{shift}$	$8n^2$
Iconic (1 per PE)	$40*n^2+8\log_2 n+3+(n^2+2n-3)T_{shift}$	$8(n^2 + \log_2 n^2)$
Count (per rank)	136	None

Table 5.5 - Comparison of Rank order filter calculations on the WPM.

5.4.2.3 Mean and Variance calculation

The mean grey-level and variance of an image region are fundamental quantities used within image analysis and are sometimes used in the classification and correspondence of objects through image sequences. Both calculations involve a summation across the image followed by a single division by the number of values within the summed region. The mean, $\bar{I}$, and the variance σ_I^2 are given by:

$$\bar{I} = \frac{1}{N} \sum_{x,y \in R}\sum I(x,y) \qquad \text{and} \qquad \sigma_I^2 = \frac{1}{N} \sum_{x,y \in R}\sum (I(x,y) - \bar{I})^2$$

where N is the size of the summed region R. Both of these quantities can be calculated in a bit serial fashion, by counting powers of two as described by Bowen [Bowen82], using the associative count within each Cluster.

The PEs within the SIMD array are initially labelled with their respective x and y position within the whole processor array. The mean is calculated by initially starting with the MSB of the data, counting the number of 1's set within the region, and the resultant count shifted one place to the left on the controller (i.e. doubling the resultant count). This is repeated for subsequent bits, although no shift is performed on the last bit of the data. The shift on the controller ensures that progressive bit-planes are treated as having half the significance of the previous bit-plane.

The variance is found in a similar way, but is preceded by the calculation of the mean, a subtraction and a square operation within each PE. The count and controller shift operations take two cycles per data bit-plane (using the pipeline design of the count). The total time taken is

$$(2*\text{nbits}+3) \text{ cycles}$$

for the mean and

$$(6*\text{nbits}+6+T_{sub}+T_{mult}) \text{ cycles}$$

for the variance, where nbits is the size of the data, T_{sub} is the subtraction time and T_{mult} is the multiplication time . For 8-bit data ($T_{sub} = 28$, $T_{mult} = 236$), the time for the mean and variance are 19 and 318 cycles respectively. The variance time is dominated by the multiply operation, T_{mult}. The normalisation of the resulting two values, by the division, can be performed on the Cluster controller taking an additional 85 cycles.

5.4.2.4 Image moment calculations

A set of moment invariants, used as shape descriptors, can be used for object classification purposes [Nevatia82]. These are invariant to rotations, translations and scaling in size. A set, M_1 to M_7, of these moment invariants is listed below :-

$$\begin{aligned}
M_1 &= \eta_{20} + \eta_{02} \\
M_2 &= (\eta_{20} - \eta_{02})^2 + 4\eta_{11}^2 \\
M_3 &= (\eta_{30} - 3\eta_{12})^2 + (3\eta_{21} - \eta_{03})^2 \\
M_4 &= (\eta_{30} + \eta_{12})^2 + (\eta_{21} + \eta_{03})^2 \\
M_5 &= (\eta_{30} - 3\eta_{12})(\eta_{30} + \eta_{12})[(\eta_{30} + \eta_{12})^2 - 3(\eta_{21} + \eta_{03})^2] + \\
&\quad (3\eta_{21} - \eta_{30})(\eta_{21} + \eta_{03})[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2] \\
M_6 &= (\eta_{20} - \eta_{02})[(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2] + \\
&\quad 4\eta_{11}(\eta_{30} + \eta_{12})(\eta_{21} + \eta_{03}) \\
M_7 &= (3\eta_{21} - \eta_{03})(\eta_{30} + \eta_{12})[(\eta_{30} + \eta_{12})^2 - 3(\eta_{21} + \eta_{03})^2] - \\
&\quad (\eta_{30} - 3\eta_{12})(\eta_{21} + \eta_{03})[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2]
\end{aligned}$$

where $$\eta_{pq} = \frac{1}{\mathrm{area}^{[(p+q)/2+1]}} \sum_{x,y \in R} \sum (x - \bar{x})^p (y - \bar{y})^q$$

for an image region R, of known size, with a centroid position of $(\bar{x}, \bar{y})$. The calculation of the moment invariants at first glance seems complex but they can be decomposed into a set of products in the form $(\eta_{pq} + \eta_{rs})$ which, once calculated, can be used in the calculation of subsequent moments. The processing required is as follows, assuming that the PEs are initially labelled with their (x,y) position within the processor array and that all resulting values are in 16-bit fixed point integer form.

1) Calculate image area and centroid $(\bar{x}, \bar{y})$, broadcast results back to the SIMD array

2) perform the subtraction between the centroid and the PE location, followed by the squaring and cubing of each value, using the required combinations of p and q (nine multiplications)

3) perform the division by the area for each one of η_{11}, η_{02}, η_{20}, η_{03}, η_{30}, η_{21}, η_{12} and a further $\sqrt{}$area division (approximated to be a division by the upper half bits of the area) for the latter 4 values

4) use the count to perform the summation for each of the seven values of η

5) add together the various values of η on the controller forming partial product terms, which can then placed on the SIMD array, in vector mode, and perform the necessary multiplications to calculate M_1 to M_7. In total two iterations of this process are required due to data dependencies.

The total time taken for this operation on the WPM is 3838 cycles. The values of M_1 to M_7 are calculated in 16-bit fixed point format.

5.5 Matrix Operations on the WPM

The matrix operations of addition, subtraction, multiplication, inversion and transposition are frequently used within many computational applications such as the analysis of data where relationships between many variables exist. An example is the formulation of a Kalman filter [Kalman60], containing information about a system being modelled. This is related to a measurement model and is updated at every measurement time point. The system and measurement models can contain an arbitrary number of values. The computational requirements of the matrix operations are examined here.

Many Kalman filters may be operating in parallel and require updating at the same time. The use of Kalman filters for tracking applications is discussed later in Chapter 6, along with their computational requirements. The mapping of the matrices across a processor array differs from the previously described methods, in Section 5.3, in that the matrices are typically smaller than the processor array. Any data mapping technique used must maximise the number of such matrices which can be operated on in parallel, whilst also minimising the time taken.

5.5.1 Mapping considerations

A major impact, on the performance of matrix operations, is the way in which the matrices are mapped across the processor array and the number of matrices undergoing the same operation at any one time. Two extreme cases occur when there is only a single matrix mapped across the processor array and the other when there are sufficient matrices such that one can be mapped to each PE and processed in parallel. The computation required for the matrix operation can be separated into two parts - the actual computation between the matrix elements (addition/subtractions or multiplications/divisions) and the time taken for necessary routing of data between PEs (required in matrix multiplication, inversion and transposition). The mapping used should be chosen so as to minimise the latter routing time, while mapping as many matrices across the array as possible for greatest utilisation.

In the following analysis both a WPM M-SIMD array and a conventional DAP SIMD array are compared. It is assumed that the size of the matrices being processed are of size nxn with each data element in 32bit floating-point format, the processor array is of size NxN and that an integer number of Clusters are within the WPM. The data mappings used are centred around those of sheet and crinkled mappings, discussed in Section 5.3. However, for the mapping of matrices it is more important to analyse the number of matrices that can fit across the array at any one time. Four data mappings are considered and listed below. This should not be treated as a definitive list but is representative of the options available for data mapping across an array processor.

1) One per PE - all elements of a matrix are mapped to a single PE

2) Sheet mapping - each matrix element is mapped to a PE, spatially arranged.

3) Crinkled mapping - a set of matrix elements are mapped to a single PE, such that for a crinkling factor of c, c^2 elements of each matrix are contained within each PE.

4) Linear Crinkled mapping - all the matrix elements within the same row are mapped to a single PE and spatially arranged in a column. This is a combination of sheet mapping, down the columns, and crinkled mapping, across the rows.

The mappings of 2, 3 and 4 above can be used to map, in a similar fashion to that described in Section 5.3, a matrix onto a processor array which is smaller than itself. However, in the WPM the processor array can be treated either as the whole SIMD array or as the set M-SIMD arrays of the Clusters. For instance, a 32x32 matrix may be sheet mapped, or crinkled mapped, into a 16x16 SIMD array within a Cluster, or sheet mapped across four Clusters.

A comparison of the mappings used, to place the matrices onto a conventional DAP SIMD array, within a WPM Cluster, and across the WPM array, is given in Table 5.6. The number of PEs required for each mapping, along with the number of such matrices that can be mapped across the whole SIMD array, and within a single WPM Cluster (if the matrix is small enough), are shown. The lower bounds on the quantities within Table 5.6 ensures that only whole matrices mapped across the array are considered. The number of matrices mapped within the WPM array is the total number of Clusters within the array divided by the number of Clusters used by each matrix. It is assumed for simplicity that each matrix larger than 16x16 has sole use of the Clusters it is mapped onto, e.g. a matrix of size 17x17 would have the sole use of four Clusters.

Mapping	PEs used	# of matrix elements/PE	# of matrices in SIMD array	# of matrices in the WPM	
				1) Cluster ($n<=16$)	2) array ($n>16$)
One per PE	1	n^2	N^2	16^2	N^2
Sheet	n^2	1	$\left\lfloor \frac{N}{n} \right\rfloor^2$	$\left\lfloor \frac{16}{n} \right\rfloor^2$	$\left\lfloor \frac{N/16}{\lceil n/16 \rceil} \right\rfloor^2$
Crinkled	$\frac{n^2}{c^2}$	c^2	$\left\lfloor \frac{N}{(n/c)} \right\rfloor^2$	$\left\lfloor \frac{16}{(n/c)} \right\rfloor^2$	$\left\lfloor \frac{N/16}{\lceil n/(16c) \rceil} \right\rfloor^2$
Linear crinkled	n	n	$N*\left\lfloor \frac{N}{n} \right\rfloor$	$16*\left\lfloor \frac{16}{n} \right\rfloor$	$16*\left\lfloor \frac{N/16}{\lceil n/16 \rceil} \right\rfloor$

Table 5.6 - Comparison of the mapping of matrices across a processor array.

The number of matrix elements mapped on to each PE in each mapping represents the increase in computation required for that mapping, and also the data storage (in words) required per PE. The product between the number of matrix elements per PE, and the number of PEs used remains constant at n^2, irrespective of the mapping used (i.e.- the total number of matrix elements within the matrix remains constant).

5.5.2 Matrix Algorithms

The algorithms for matrix addition, subtraction, division, inversion and transposition are described below. It is assumed, for operations between two matrices, that they are both mapped in the same way across the same PEs within the processor array. The way in which these matrix elements are initially placed across the array has not been considered. The time for the 32-bit floating point operations of addition, subtraction, multiplication and division are denoted below as T_{fadd}, T_{fsub}, T_{fmult}, T_{fdiv} respectively.

5.5.2.1 Matrix addition and subtraction

Both the operations of matrix addition and matrix subtraction can be very easily performed on a processor array no matter which data mapping is used. There is no overhead in data communications - all the additions/subtractions to be performed are with matrix elements mapped to the same PEs. Thus, the total time taken is simply the time taken to perform the floating point addition, T_{fadd}, or subtraction, T_{fsub}, times the number of matrix elements mapped to each PE.

5.5.2.2 Matrix multiplication

Matrix multiplication requires the dot product between each row of the first matrix and each column of the second matrix. A total of n^3 multiplications and $(n-1)n^2$ additions are required. The multiplication of two matrices **A, B** is shown below. Matrix elements are referred to as a_{ij} and b_{ij} respectively.

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & & a_{2n} \\ \vdots & & & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} * \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{12} & & b_{2n} \\ \vdots & & & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{bmatrix} =$$

$$\begin{bmatrix} a_{11}b_{11}+a_{12}b_{21}\cdots a_{1n}b_{n1} & a_{11}b_{12}+a_{12}b_{22}\cdots a_{1n}b_{n2} & \cdots & a_{11}b_{1n}+a_{12}b_{2n}\cdots a_{1n}b_{nn} \\ a_{21}b_{11}+a_{22}b_{21}\cdots a_{2n}b_{n1} & a_{21}b_{12}+a_{22}b_{22}\cdots a_{2n}b_{n2} & & a_{21}b_{1n}+a_{22}b_{2n}\cdots a_{2n}b_{nn} \\ \vdots & & & \vdots \\ a_{n1}b_{11}+a_{n2}b_{21}\cdots a_{nn}b_{n1} & a_{n1}b_{12}+a_{n2}b_{22}\cdots a_{nn}b_{n2} & \cdots & a_{n1}b_{1n}+a_{n2}b_{2n}\cdots a_{nn}b_{nn} \end{bmatrix}$$

Although this looks complex, it can be split up into dot products between each row and each column of the matrices. This results in a total of n matrices, one from each dot product, which are added together forming the matrix multiplication result above. The form of the dot products are shown below, where the terminology of $A_{n\bullet}$ specifies the nth row of A and $A_{\bullet n}$ specifies the nth column.

$$\begin{bmatrix} A_{\bullet 1} & A_{\bullet 1} & \cdots & A_{\bullet 1} \end{bmatrix} * \begin{bmatrix} B_{1\bullet} \\ B_{1\bullet} \\ \vdots \\ B_{1\bullet} \end{bmatrix} +$$

$$\begin{bmatrix} A_{\bullet 2} & A_{\bullet 2} & \cdots & A_{\bullet 2} \end{bmatrix} * \begin{bmatrix} B_{2\bullet} \\ B_{2\bullet} \\ \vdots \\ B_{2\bullet} \end{bmatrix} + \dots +$$

$$\begin{bmatrix} A_{\bullet n} & A_{\bullet n} & \cdots & A_{\bullet n} \end{bmatrix} * \begin{bmatrix} B_{n\bullet} \\ B_{n\bullet} \\ \vdots \\ B_{n\bullet} \end{bmatrix}$$

The decomposition of matrix multiplications can be very easily implemented on an SIMD processor array. It requires a column of the first matrix, A, to be broadcast across the array and a row of the second matrix, **B**, to be broadcast down it. The broadcast can be done either by using the associative response mechanism of the array, and broadcasting a row (and column) across the array, or by shifting the values of the row (and column) across the array (using masking and conditional storage operations). Either method may be employed, the quickest depends upon the size of the matrices and the number of matrices mapped across the array.

Once the broadcast has taken place the first multiplication between each broadcasted value can be performed, taking T_{fmult} cycles, times the number of matrix elements mapped to each PE. The broadcast is then repeated for the second and subsequent row/column and the associated multiplication performed. A post addition stage accumulates all of the partial products taking $(n-1)*T_{fadd}$ cycles.

5.5.2.3 Matrix inversion

Matrix inversion on an array processor can be performed through the use of an algorithm developed by Faddeev [Faddeev59] which allows in-place matrix inversion operations [Grinberg84]. The algorithm works on the four matrices, **A**, **B**, **C** and **D** as shown in Figure 5.12. Gaussian elimination is applied to the rows and columns of the whole matrix forming the result of **C** $\mathbf{A}^{-1}$ **B** + **D**. Thus, if **A** is set to be the data matrix, **C** and **B** to be the identity matrix and **D** to be a zero matrix, the inverse, $\mathbf{A}^{-1}$, can be obtained.

$$\begin{array}{c|c} \mathbf{A} & \mathbf{B} \\ \hline -\mathbf{C} & \mathbf{D} \end{array} \longrightarrow \mathbf{CA}^{-1}\mathbf{B} + \mathbf{D}$$

Figure 5.12 - The Faddeev formulation of matrix inversion.

A Gaussian elimination of a single row and column is performed by forming a new matrix derived from **A**, in which each element is equal to the product of the leading row and column elements from **A**, followed by a division by the top left element of **A**. The new matrix is then subtracted from the original matrix **A**, and carried forward to the next iteration of the algorithm. A total of n iterations are required. The products between the leading row and column elements can be formed by the broadcast of the first row and first column, of the matrix, across the processor array, followed by an in-place multiplication. The division can be performed after the global broadcast of the top left element of the matrix **A** across the processor array.

The double sized matrix implied by Figure 5.12 is not required, if the original matrix is shifted up and left one place before the subtraction of the new matrix. This keeps the active part of the double sized matrix on the same PEs within the array on all iterations. This is shown more clearly in the example of Figure 5.13 which depicts one iteration of the inversion algorithm on a 3x3 matrix (taken from the example by Grinberg [Grinberg84]). The shaded region indicates the active region of the double sized matrix which is mapped onto the same PEs within the processor array, throughout the algorithm.

$$\left[\begin{array}{ccc|ccc} 8 & 2 & 6 & 1 & 0 & 0 \\ 7 & 9 & 4 & 0 & 1 & 0 \\ 3 & 1 & 5 & 0 & 0 & 1 \\ \hline -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 \end{array}\right]$$

a) Original

$$\begin{array}{cccccc} 8 & 2 & 6 & 1 & 0 & 0 \\ 7 & 1.75 & 5.25 & 0.875 & 0 & 0 \\ 3 & 0.75 & 2.25 & 0.375 & 0 & 0 \\ -1 & -0.25 & -0.75 & -0.125 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

b) row*col / (first element)

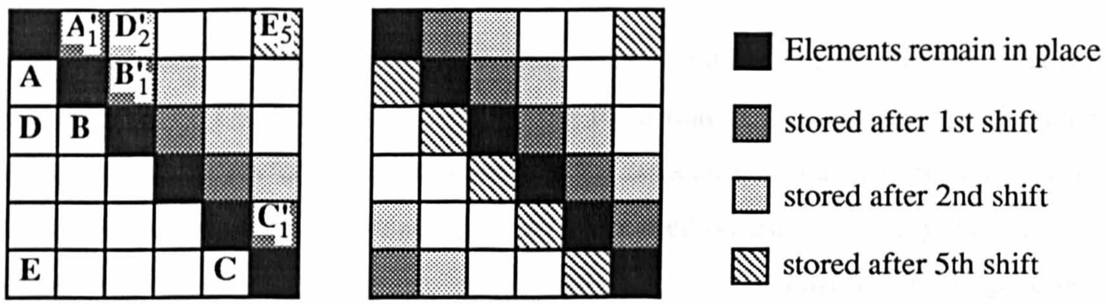
$$\begin{array}{cccccc} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 7.25 & -1.25 & -8.75 & 1 & 0 \\ 0 & 0.25 & 2.75 & -0.375 & 0 & 1 \\ 0 & 0.25 & 0.75 & 0.125 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 \end{array}$$

c) Subtraction from original

Figure 5.13 - Example iteration of the Faddeev inversion algorithm on a 3x3 matrix.

5.5.2.4 Matrix transpose

The transposing of a matrix requires only data routing to move the elements of the matrix across the processor array. If all the matrix were mapped onto a single PE - no routing would be required, only a change in addressing (which can be performed with no time cost). When the matrix is mapped across a processor array, shifting operations are required to move the elements to their destination such as is shown for a 6x6 matrix in Figure 5.14.



No wrap-around With wrap-around

Figure 5.14 - Transposing a matrix on an array processor.

All elements of the matrix need to be moved except the leading diagonal elements as illustrated in Figure 5.14. The operation can be understood by considering the lower left half of the matrix (not including the leading diagonal). This region is shifted up and right one position with the elements on the top diagonal of this being stored on the array - e.g. **A, B, C** are moved to **A'**$_1$, **B'**$_1$, **C'**$_1$ respectively. This is repeated for n-1 iterations (n being the dimension of the matrix, in this case 6). The index on the values in Figure 5.14 indicate the iteration in which the values are stored.

The matrix transpose algorithm is performed twice, once for the lower left region of the matrix and another for the upper right (shifting it down and left). However, if the matrix is the same size as the array, the algorithm need only be performed once by using the arrays wrap-around facility as illustrated on the right hand matrix in Figure 5.14.

The time taken to perform a matrix transpose is

$$(128T_{shift} + 194)(n-1) \text{ cycles}$$

when not using the wrap-around of the array, where T_{shift} is the time for a single bit shift. If the matrix is the same size as the processor array (16x16 on a single Cluster of the WPM or 128x128 for the whole WPM array) this time is halved requiring

$$(64T_{shift} + 97)(n-1) \text{ cycles.}$$

Note that, if the matrix is larger than the Cluster size within the WPM then the shift time, T_{shift}, is two cycles, due to handshaking between Clusters, otherwise it is one.

5.5.3 Performance of matrix operations

The time spent on the computation and the data routing for all four matrix operations can be seen in Table 5.7. The data routing for a matrix multiplication and matrix inversion can be performed by using either the associative broadcast mechanisms of the SIMD array, or by a shifting operation. The method used is simply the one that takes the minimum number of cycles. The distance, d, in the shifting term is governed by the number of PEs the matrix is mapped over, i.e. n, n/c and n, in sheet, crinkled, and linear crinkled mappings respectively.

Matrix	Computation (cycles)	Routing (Broadcast)	Routing (Shifting)
Addition	T_{fadd}	zero	zero
Multiplication	$(T_{fmult}+T_{fadd})*n - T_{fadd}$	$64nT_{broad}$	$64n(1+(d-1)(T_{shift}+2))$
Inversion	$(T_{fmult}+T_{fdiv}+T_{fsub})*n$	$96nT_{broad}$	$128n(1+(d-1)(T_{shift}+2))$
Transposition	zero	-	$(128*T_{shift}+194)*(d-1)$

Table 5.7 - Time for the computation and communication in matrix operations.

The time taken to perform a single 1-bit shift, T_{shift}, is 2 cycles on the WPM (ignoring any necessary synchronisation) when the matrix is mapped across more than one Cluster and 1 cycle otherwise (the same as on the conventional DAP). The broadcast time, T_{broad}, is 4 cycles when only one matrix is mapped across the array (when n=16 on a Cluster or n=N on a conventional SIMD array). T_{broad} is 6 cycles when several matrices are mapped within one SIMD array or across several Clusters in the WPM. In the case of the WPM, the broadcast time is inflated by an additional 6 cycles per Cluster that the matrices are mapped over, to account for its partitioned associative mechanism.

When several matrices are mapped across the processor array, the associative mechanism, although allowing quick broadcasting of values across the array, must be used serially for each row/column of the matrices in the array. This increases the broadcast time by a factor of $\left\lceil \frac{N}{m} \right\rceil$ (representing the integer number of matrices within the single dimension of the processor array of size N). An example of four 4x4 matrices mapped onto a 16x16 PE array is shown in Figure 5.15. Each of the first columns of the matrices are broadcast across the PEs containing their matrix elements. This is done serially for column A, B, C and D. The WPM can perform a separate broadcast within each Cluster, decreasing the overall time taken for the broadcast operations.

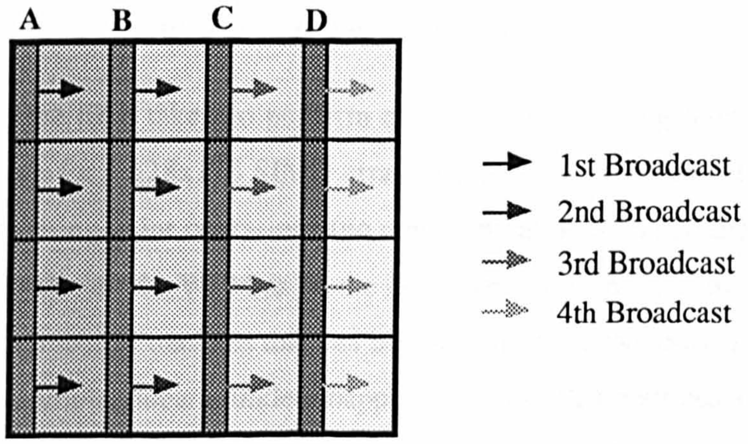


Figure 5.15 - An example of sixteen matrices mapped onto a processor array.

The times in Table 5.7 assume that each matrix element is mapped onto a single PE (i.e. sheet mapping). However, other data mappings can be employed as detailed in Table 5.6. The effect on the computation, broadcasting and shifting for the matrix operations is shown in Table 5.8. These are multiplicative factors which should be applied to the matrix cycle times given in Table 5.7.

The performance of the matrix operations, on various size and numbers of matrices, has been simulated using the computational and routing requirements listed in Tables 5.7 and 5.8. The times for the floating point operations were taken from those given by Beal [Beal90] and also listed in Table 4.6.

Mapping	Multiplicative increase in :-		
	Computation	broadcast	shifting
One per PE	n^2	0	0
Sheet	1	$\lfloor \frac{N}{n} \rfloor$	1
Crinkled	c^2	$\lfloor \frac{N}{(n/c)} \rfloor$	c^2
Linear crinkled	n	$\frac{1}{2}\lfloor \frac{N}{n} \rfloor$	$\frac{n}{2}$

Table 5.8 - Multiplicative factors applied to the computation, broadcast and shifting times of the various data mappings.

5.5.3.1 Addition and subtraction performance

In Figure 5.16, the time taken to perform addition on a varying number of 16x16 matrices is shown for a 128x128 SIMD array (either DAP or WPM). The operation involves only floating point operations (no communications) and so the time for the operation is the same on the WPM as on the DAP. The steps that are apparent are due to the differing number of matrices that can be mapped across the array in the different mappings. For example, linear crinkled mapping enables 1024 matrices to be operated upon in parallel. When this number is exceeded there is a sudden jump to having to processes up to an extra 1024 matrices, i.e. a doubling of the time taken. Note that the change in the steps within Figure 5.16 (and subsequent figures) should be vertical but are shown sloped due to the plotting resolution used.

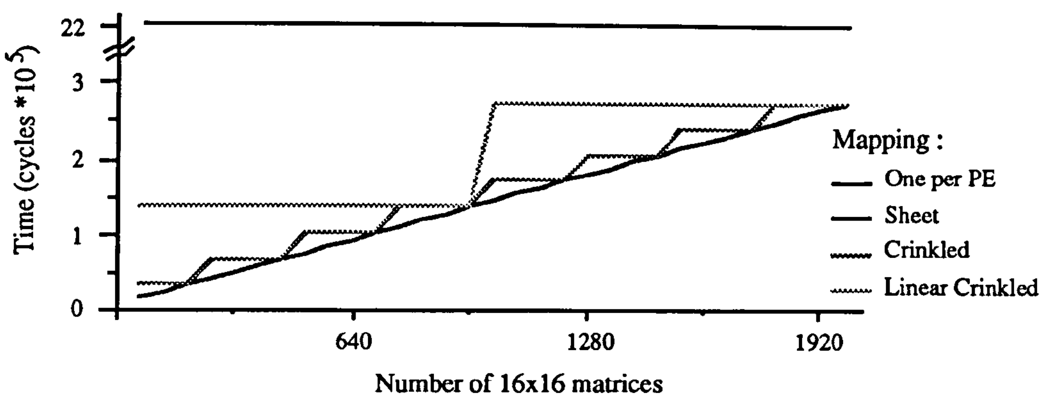


Figure 5.16 - Time taken to perform a 16x16 matrix addition for a varying number of matrices.

5.5.3.2 Multiplication performance

The time taken for the multiplication of 16x16 matrices is shown in Figure 5.17 for a 128x128 DAP SIMD array. In Figure 5.18 this is shown for an 8x8 Cluster WPM. It can be seen in both graphs that the times taken by the different mappings are affected by the broadcast/communications required within the matrix multiplication. The Linear Crinkled mapping and crinkled mapping have the minimum communication times on the DAP and can be seen to be most effective when the number of matrices lies between 600 and 1024 (when the utilisation of both of these mappings is high). The one per PE mapping is not shown - it has a constant time requirement of 8.5×10^6 cycles over the range of number of matrices considered. When the number of matrices approaches that of the number of PEs, the one per PE mapping will be the most efficient - it would have high utilisation and no requirement for data routing.

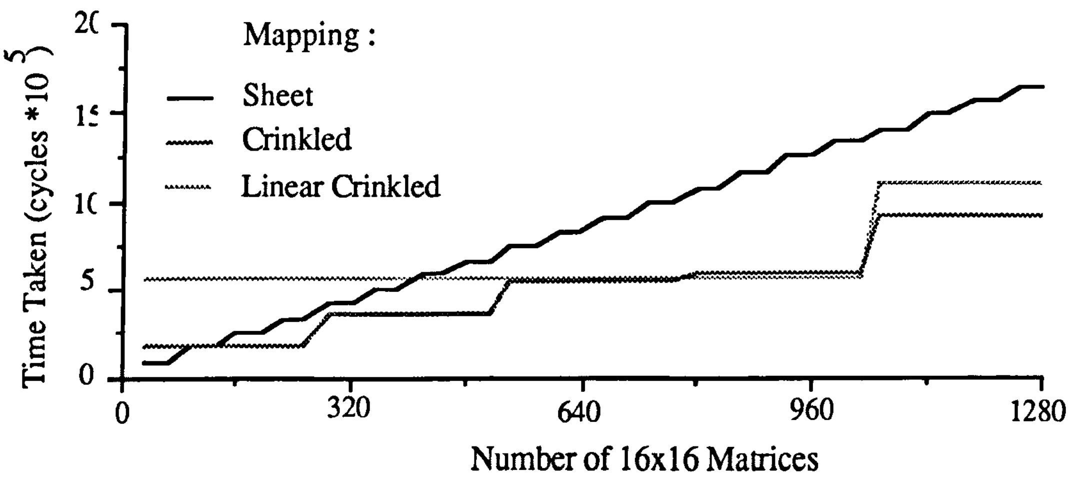


Figure 5.17 - 16x16 matrix multiplication on a 128x128 DAP SIMD array.

On the WPM, the communications required can be performed in less time due to the partitioned associative mechanism within each Cluster. This can be seen in Figure 5.18, the curve of each mapping being similar to that for the addition in Figure 5.16, where no communications were needed.

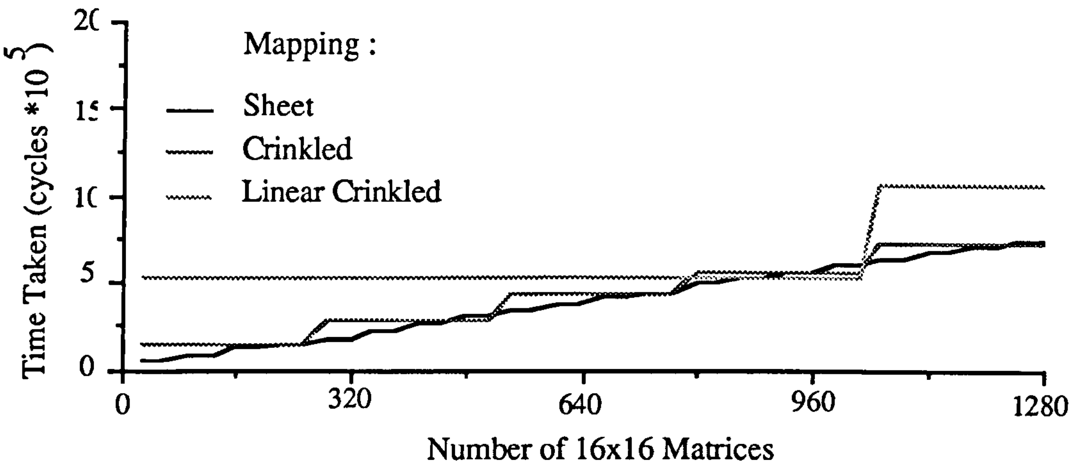


Figure 5.18 - 16x16 matrix multiplication on an 8x8 Cluster WPM.

The mapping which has the minimum time requirement, over the range of matrices considered, is plotted in Figure 5.19 for both the DAP and WPM arrays. It can be seen that the WPM out-performs the DAP array, in places by 50%. This is purely due to the effect of the decrease in communication times using the local associative mechanisms. The required floating point operations are the same for both arrays.

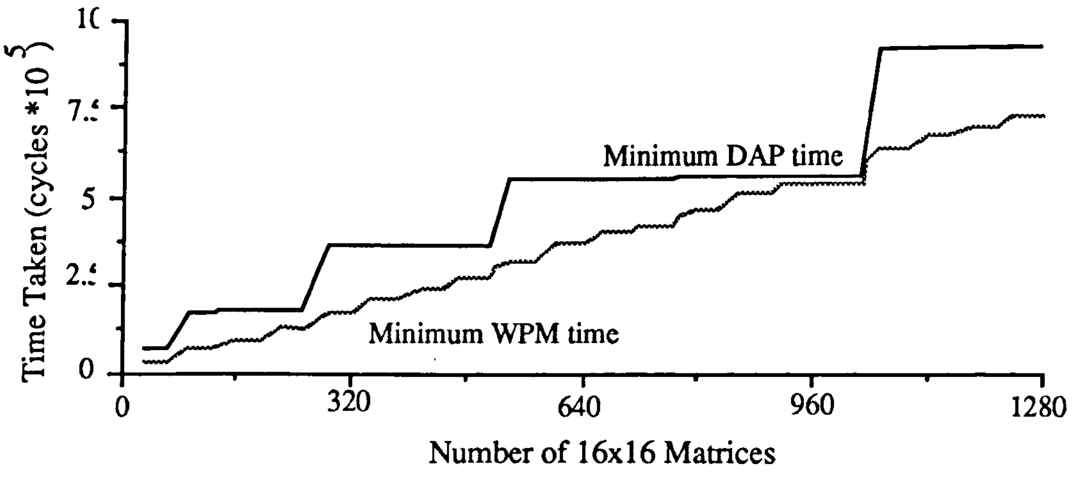


Figure 5.19 - Comparison of the minimum processing time for matrix multiplication between the DAP and WPM arrays.

The size of the matrices was chosen in the above example to match the size of the WPM Cluster, giving the WPM an ideal processing capability. Recall however, that if the size of the matrices is 17x17, 4 Clusters are required, reducing array utilisation. Thus, one should examine a range of matrix sizes in order to gain a fuller comparison between the WPM and the DAP. Figure 5.20 gives a comparison for the mappings considered, for a range of matrix sizes. It is assumed that the processor array was as fully utilised as possible within each mapping, i.e. the number of matrices was equal to the maximum that could be placed across the array (see Table 5.6).

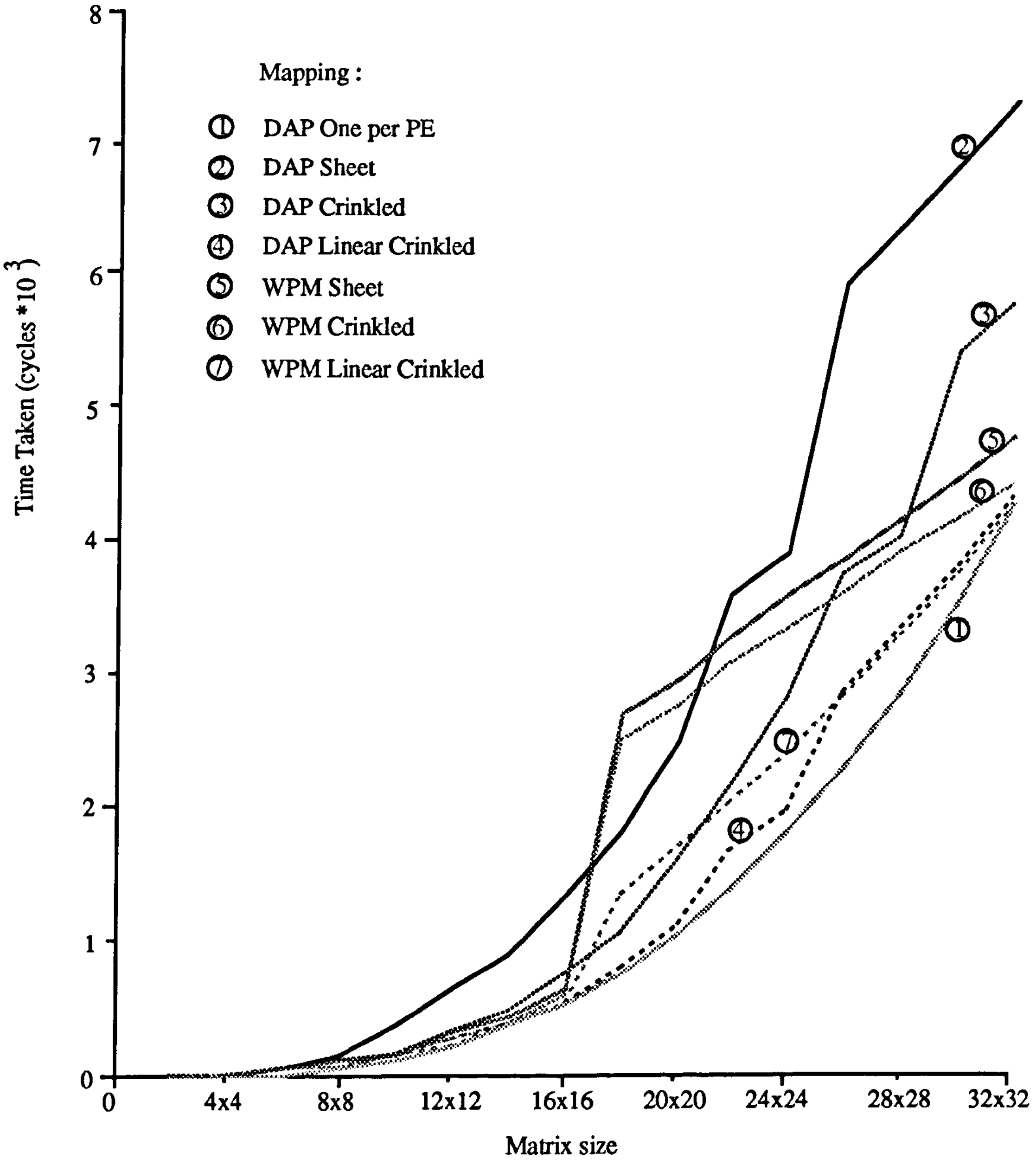


Figure 5.20 - Comparison between the WPM and the DAP SIMD array for matrix multiplication over a range of matrix sizes.

The times shown in Figure 5.20 are given as the effective number of cycles per matrix multiply, i.e. the overall time taken divided by the number of matrices worked on in parallel. The effect of the partitioned associative mechanism within the WPM can be seen for a matrix size of 17x17. The sheet mapping, crinkled and linear crinkled mappings all exhibit a sudden jump due to the increase in broadcast/communication required when performing the inter-Cluster operations. The jump is also a feature of the poor utilisation that a 17x17 matrix has using 2x2 Clusters (32x32 PEs).This poor utilisation does not occur on the DAP SIMD array.

5.5.3.3 Inversion performance

A similar comparison to that of matrix multiplication was carried out for matrix inversion and yielded similar results. The comparison of the minimum time taken over the range of the number of matrices considered is shown in Figure 5.21. This is similar to that of Figure 5.19 for the matrix multiplication. The effect of the size of the matrices on performance of matrix inversion is similar to that of the multiplication, as shown in Figure 5.20, and has not been included here.

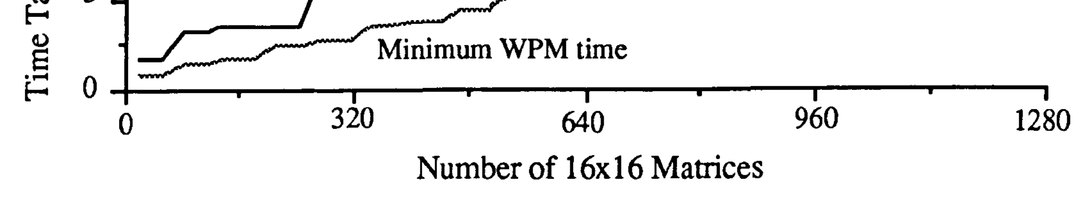


Figure 5.21 - Comparison of the minimum processing time for matrix inversion between the DAP and WPM arrays.

5.5.3.4 Transpose performance

The time taken to perform a transpose is dependent only on the shifting communication time (see Table 5.7). The times for the One per PE data mapping is zero - no communications are required, only a change in data addressing. A comparison between the time taken for matrix transposition on the WPM and the DAP SIMD array is shown in Figure 5.22 using sheet mapping for a range of matrix sizes. The performance on either array is the same for matrices less than 16x16. The torus network within the WPM Cluster can be used when the matrix is of size 16x16, halving the time required. For matrices larger than 16x16, the shifting time on the WPM doubles, thus increasing the overall time taken for the transposition in comparison to the DAP SIMD array.

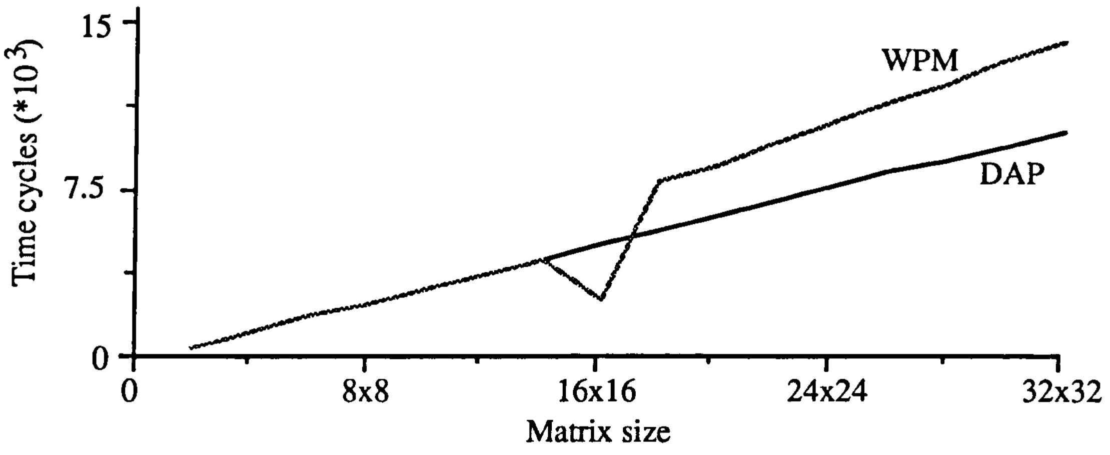


Figure 5.22 - Comparison of matrix transpose on the DAP and WPM arrays.

The operations of matrix addition and subtraction can be performed on either the SIMD array within the DAP or on the M-SIMD array within the WPM, with no difference in the time for execution. These operations do not require any communications only the use of the computational components of the PEs. This is not true for matrix multiplication and inversion - both of which need communications across PEs when the matrix is mapped across more than one PE. The communication time is altered on the WPM by the use of its partitioned associative response networks. For matrices of size less than or equal to 16x16 the time taken is less on the M-SIMD array than on the conventional SIMD array. However, it is worse for matrices of size between 17x17 and 26x26 (see Figure 5.19).

Matrix transposition was shown to have similar performance on both the WPM and the conventional SIMD array for matrices of size up to 16x16. However when inter-

Cluster communications are required, the shifting time is doubled on the WPM, thus increasing the time taken in comparison to the SIMD array.

The SIMD element used for the comparison was the simple 1bit DAP PE (as used in the DAP product and the WPM). However the computational power of SIMD PEs is constantly increasing - the DAP is now produced with an 8-bit co-processor [AMT90] enhancing floating-point operations by a factor of 5, and other 4-bit PEs have appeared such as in the MAS-PAR [Nickolls90]. The effect of these processors on matrix operations will be to decrease the computation time required, making the communication time more prominent in the overall processing time. The increase in performance the WPM achieves, over that of a conventional SIMD array, for matrices of size of 16x16 (and others) will be further improved upon in such situations.

Only square matrices have been considered above. The analysis could be extended to rectangular matrices, adding another matrix size parameter, but this has not been done. Another factor would be to allow matrices of different sizes to be processed across the array processor. It is expected that the local-autonomy within the WPM would aid the processing of different sized matrices, one size of matrices could be mapped onto one set of Clusters, and another size to a different set.

5.6 Summary

A comparison of the WPM with its M-SIMD array over that of the conventional SIMD array has been given in Section 5.2. The local autonomy, associative response mechanisms and the inter-Cluster communications have all shown that a performance increase can be achieved over a conventional SIMD array in situations where the size of the data being processed is the same size as the M-SIMD array within a Cluster.

Image processing operations were examined in Section 5.4. Such operations are iconic and global, requiring data parallel usage of the processor array. The performance of the WPM in such situations is affected by the increase in time required for inter-Cluster synchronisation and the increased time required for the shifting of data between PEs.

Local area operations using the count were examined in Section 5.4.2 including histogram generation, rank order filters and various moments. Such operations require one output over the local region in which they are performed. The local autonomy coupled with the enhanced associative mechanisms within the WPM improve the performance of these operations.

Finally in Section 5.5 the matrix operations of addition, subtraction, multiplication, inversion and transposition were examined. These operations are effectively local region operations, in the image processing sense, when considering a matrix as a local region. It was shown that matrix multiplication and inversion can be performed in reduced time on the WPM compared to a conventional SIMD array depending upon the matrix size. Matrix subtraction and addition require computation which is only affected by the performance of the PEs, and not by the way in which the PEs are configured. A matrix transpose can be performed on either processor array with similar time requirements, although the time taken on the WPM is increased by the synchronisation of adjacent Clusters and by the increased communication time.

In the next chapter, the application domain of target tracking is considered. The ways in which the differing types of data, that occur, can be mapped across the WPM array is examined. The use of both the M-SIMD and the MIMD arrays within the WPM is required.

Chapter 6

The Performance of Tracking Operations on the WPM

6.1 Introduction

The tracking of objects, both in the form of iconic regions which have an observable size and shape, and those that appear as only a few pixels, and its implementation on the WPM is considered within this chapter. Typically, the computational requirement is complex in that both image data and derived numerical quantities have to be processed over a number of frames, extracting temporal information. In this chapter, the structure of the data involved within the computation is described, along with the use of the different levels of the WPM.

Target tracking has received much attention over the last two decades for both military and civilian applications [Bar-Shalom88]. More recently, there has been increasing interest in applying similar techniques to computer vision for the tracking of image and object features. Algorithms have been developed for the estimation of the motion parameters of moving objects, the estimation of camera ego-motion, and a combination of the two [Vega89].

The techniques used for tracking and computer vision are similar, and the underlying models are sometimes identical. For example, both require feature points (or measurements) as inputs from each image. These are incorporated into models of the kinematic motion, using optimal estimators to produce estimates of the motion, and predications of where the motion will take the feature points in subsequent frames.

The processing involved for tracking applications requires the initial processing of the sensor data, in an application specific manner, in order to produce measurements. These are then incorporated into estimators for kinematic information and predications. The computational requirements range from global iconic processing on the sensor data, through to numeric processing for the estimation process. Both types of processing requirements are illustrated in this chapter, through two example tracking situations.

This chapter is divided in the following way. The remaining part of this introduction gives an overview of estimation using observed measurements. Section 6.2 details the different types of estimators that can be used, along with tracking models commonly used for the estimation of kinematic information. The first example application describes a low density situation, detailed in Section 6.3, using illustrative tracking models and image processing to track objects across an image. The second example application describes a higher density situation, detailed in Section 6.4, using a generic form of tracking which can arise in both computer vision and target tracking domains. The mapping and performance considerations of both applications, on the WPM, are discussed in Section 6.5.

6.1.1 Estimation of unknown quantities

If the motion of an object under track is unknown by an observer, its effect can be observed through measurements from the object, such as its position (x, y) on the image plane, or range, bearing, and angle from RADAR. Each measurement has an associated error variance resulting from measurement noise. The error variance gives a measure of the amount of information contained within the measurement and thus a measure of how much it can be relied upon.

The motion must be estimated from a sequence of such measurements using both a model of the motion being performed, and an estimation strategy. Inadequacies in either the model or the estimation strategy leads to poor tracking performance. The optimal estimation of a system from a sequence of measurements is discussed below. It has been rigorously covered in mathematical terms in several texts, see for example Maybeck [Maybeck79] and Sorenson [Sorenson75].

The problem of estimating the states of a given system, **x**, at a time n, broadly stated, is to find a function, f, that estimates the value of **x**, using available measurement information :-

$$\mathbf{x}(n) = f\left[\, n, Z^n \,\right] \qquad \text{where} \qquad Z^n = \{\, \mathbf{z}(j),\ j = 1,..,n \,\}$$

Z^n is the time history of the observations (measurements) and z(j) is the measurement of the system at time j. The measurement, at time n, depends upon the state, **x**(n), time n, and some random perturbation (measurement noise) **w**(n) :-

$$\mathbf{z}(n) = h[\, n, \mathbf{x}(n), \mathbf{w}(n) \,]$$

An example showing the combination of two measurements, each with an associated error variance is shown in Figure 6.1. It is assumed in this example that the variance within both measurements has Gaussian distributions. The measurements are represented by the peak of each Gaussian and the error represented by the width. Hence, the wider it is the greater the error variance. For example, the measurement Z_1 in Figure 6.1 has a smaller error variance than that of Z_2. The combination of the two forms an estimate with a narrower error distribution, which is nearer to the more accurate measurement, Z_1, than to the less accurate measurement, Z_2.

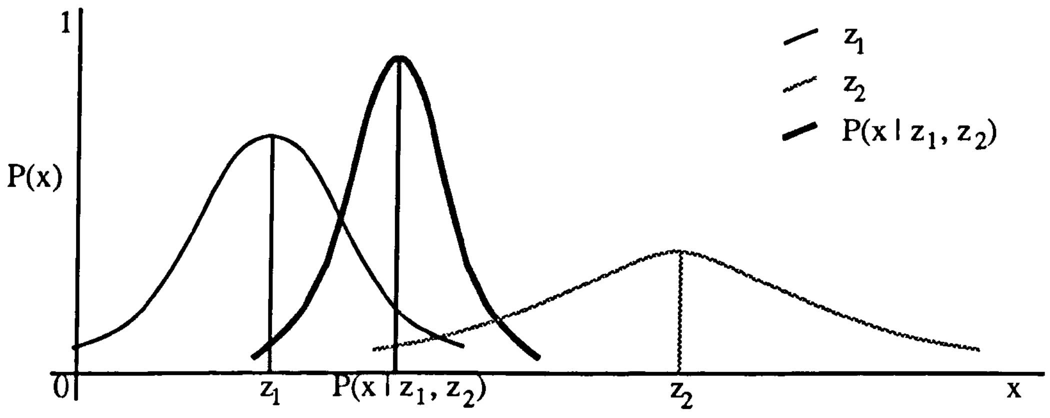


Figure 6.1 - The combination of two Gaussian measurements

There are two widely used approaches for estimation: Bayesian and non-Bayesian. The non-Bayesian approach is used when the states being estimated are not random variables. The Bayesian approach is used to estimate states which are random variables. The Kalman filter [Kalman60] is a Bayesian approach in common usage which uses a state-space representation, or model, of a linear system. The Kalman filter can be extended to model (sub-optimally) non-linear systems by the linearisation of the system at each time point. This is known as the Extended Kalman filter (EKF).

Two models are used by the Kalman filter, that of the system and that relating the measurements to the system. A generalised system and measurement model for a linear Kalman filter is :-

$$\mathbf{x}(n) = \mathbf{F}(n)\,\mathbf{x}(n-1) + \mathbf{v}(n)$$
$$\mathbf{z}(n) = \mathbf{H}(n)\,\mathbf{x}(n) + \mathbf{w}(n)$$

where

$\mathbf{x}(n)$	= the system states at time n
$\mathbf{F}(n)$	= the state transitional matrix, relating the state, $\mathbf{x}(n-1)$, to $\mathbf{x}(n)$
$\mathbf{z}(n)$	= the measurement vector at time n
$\mathbf{H}(n)$	= the observation matrix, relating the measurements to the system states
$\mathbf{v}(n)$	= additive system white noise
$\mathbf{w}(n)$	= additive measurement white noise

Associated with the system states is a covariance matrix, $\mathbf{P}(n)$. This is a measure of the noise within the system states using the system and measurement noise processes.

A discussion of estimation theory, using both approaches, is given by Maybeck [Maybeck79] and others. An introduction is given in Appendix C, along with the formulation of both the Kalman filter and the EKF. Both the Kalman filter and the EKF are used in the example tracking applications in Sections 6.3 and 6.4.

6.2 Tracking Algorithms

There are two main approaches to the analysis of motion within image sequences. The first is concerned with a dense motion calculation. Here, the motion is estimated at each pixel and examples include methods based on the calculation of optical flow. The second uses feature points, extracted from the input images, which are combined temporally within a model to give an estimation of the motion. Both the dense and sparse methods are described below.

Dense motion estimation

Optical flow is the apparent velocity of the brightness patterns within an image, resulting from relative motion between objects on the image plane and the camera. It is commonly calculated through a brightness and smoothness constraint as first described by Horn and Schunck [Horn81]. Optical flows methods, until recently, have been concerned with the calculation of the flow over two successive image frames. More recently, methods dealing with more frames have been devised, an example being the incremental calculation by Singh [Singh91]. The optical flow calculation assumes a constant image intensity between frames. This can limit its practical usage. Additionally, the smoothness constraints can result in large errors occurring along the boundaries of objects moving at different velocities.

Another example of a dense map estimation is given by Matthies et. al. [Matthies89]. He calculates a dense depth map using known camera lateral and transversal motion. The movement of each of the pixels in the image is a function of the camera motion and the distance between the scene and the camera. This relationship is incorporated into a model suitable for recursive (Kalman) depth estimation. A similar method has been used by Heel [Heel88] which also considers the possible motion of the objects contained within the image.

Sparse motion estimation

Motion estimation can be performed on sparse data sets in a similar manner to that of the dense motion estimation. Analysis, such as that given by Matthies [Matthies89] for depth estimation, gives both dense and sparse estimation techniques. The sparse data is usually assumed to be the result of some pre-processing, ideally locating temporally invariant features which appear across the image sequence. Harris [Harris87] has used corners for this purpose, to determine camera ego-motion. Line parameters have also been considered as suitable features by Deriche and Faugeras [Deriche90], such as a line's orientation, mid-point, and length.

Other examples of motion estimation using sparse data sets include that of Broida and Chellappa [Broida86]. They use a recursive solution to estimate object motion on a set of features extracted from an object. The features used in their case where assumed to be extracted by some other means. Du et. al. [Du91] use the object position in three-dimensions for the gaze control of a robots head. The head incorporates two cameras to give both the x and y position on the image plane, and a depth estimate is obtained using the stereo effect. Marslin et. al. [Marslin91] use the velocity (extracted over two frames) and the orientation of a road-vehicle (after a model matching process). This is incorporated into a four state tracking model for recursive filtering.

The detailed differences in the underlying mathematical models, of each of the applications mentioned above, vary considerably. However, it is the computational requirements that are of particular interest here. Example models used in the tracking of objects are considered in Section 6.2.1. Methods which relate the estimates produced from the tracking models to the measurements on the image plane are described in Section 6.2.2. This latter process is commonly referred to as data association.

6.2.1 Tracking models

One of the simplest linear system models used for target tracking over the last few decades is the α-β and the α-β-γ trackers. Both trackers can be implemented easily and are not computationally demanding. The α-β tracker uses a 1st-order model considering position and velocity only. The α-β-γ tracker uses a 2nd-order model which includes acceleration. Both trackers can be used when only position information can be observed.

The α-β-γ considers each tracking dimension separately. Further dimensions may be added as appropriate if they can be assumed to be de-coupled from each other. For example, tracking in the x and y image dimensions is often modelled by two separate tracking filters. The α-β-γ has the following system and measurement models :

$$\hat{x}(n|n) = \begin{bmatrix} 1 & T & \frac{1}{2}T^2 \\ 0 & 1 & T \\ 0 & 0 & 1 \end{bmatrix} \hat{x}(n-1|n-1) + \begin{bmatrix} \alpha \\ \frac{\beta}{T} \\ \frac{\gamma}{T^2} \end{bmatrix} [\, z(n) - \hat{z}(n|n-1) \,]$$

$$z(n) = [1 \quad 0 \quad 0]\, x(n) + w(n)$$

where $\hat{x}(n) = [\, x \ \dot{x} \ \ddot{x} \,]'$ (position, velocity and acceleration), z(n) is the position measurement, $\hat{z}(n|n-1)$ is the position prediction, w(n) is the observation noise and T is the sample time period.

The values of α, β, and γ are constant filter coefficients affecting the position, velocity and acceleration respectively. They determine how quickly the tracker can respond to movements of the target. Large gain values lead to more responsive filters but are also more susceptible to noise. Optimal values of α, β, and γ have been calculated by Kalata [Kalata84].

Similar models to those of the α-β-γ trackers can be used in a Kalman filter. The main advantage of a Kalman filter approach, over the fixed coefficient α-β-γ models, is that it can adapt more readily to a changing environment. As examples, two models are considered below. The first models position with constant velocity, and the second position and velocity with constant acceleration. Again, the image dimensions of x and y can be modelled separately if they can be assumed to be de-coupled from each other.

Position and velocity tracking model

The system model and measurement model for a constant velocity system, in a single dimension, is given by :

$$x(n) = \begin{bmatrix} 1 & T \\ 0 & 1 \end{bmatrix} x(n-1) + v(n)$$

$$z(n) = [\ 1 \quad 0\ ]\, x(n) + w(n)$$

where $x(n) = [x \ \dot{x}]'$ (position, velocity), z(n) is the measurement, T is the sampling time period, and w(n) is the measurement noise.

The system noise process v(n) represents the change that can occur in the kinematics of the system due to un-modelled higher order motions. If a constant acceleration, $\ddot{x}(n)$ with variance σ_v^2, is assumed throughout the time period, T, then the change in the velocity of the system will be $T\ddot{x}(n)$, with a variance of $T^2\sigma_v^2$. The change in position will be $\frac{1}{2}T^2\ddot{x}(n)$ with a variance of $\frac{1}{4}T^4\sigma_v^2$. This gives the covariance matrix of the system noise v(n) :-

$$Q(n) = \begin{bmatrix} \frac{1}{4}T^4 & \frac{1}{2}T^3 \\ \frac{1}{2}T^3 & T^2 \end{bmatrix} \sigma_v^2$$

Position, velocity and acceleration tracking model

The system model and the measurement model for a constant-acceleration system, in a single dimension, is given by :

$$x(n) = \begin{bmatrix} 1 & T & \frac{1}{2}T^2 \\ 0 & 1 & T \\ 0 & 0 & 1 \end{bmatrix} x(n-1) + v(n)$$

$$z(n) = [\ 1 \quad 0 \quad 0\ ]\, x(n) + w(n)$$

where this time $x(n) = [\ x\ \dot{x}\ \ddot{x}\ ]'$ (position, velocity and acceleration), z(n) is the measurement, T is the sampling time period, and w(n) is the measurement noise.

The system noise process, v(n), now represents the change that can occur in the kinematics of the system due to third and higher order motions. If the third order motion is assumed constant, then the covariance matrix of the noise v(n) is :

$$Q(n) = \begin{bmatrix} \frac{1}{36}T^6 & \frac{1}{12}T^5 & \frac{1}{6}T^4 \\ \frac{1}{12}T^5 & \frac{1}{4}T^4 & \frac{1}{2}T^3 \\ \frac{1}{6}T^4 & \frac{1}{2}T^3 & T^2 \end{bmatrix} \sigma_v^2$$

Other noise processes exist for these tracking models when the higher order motions are not constant. These are discussed by Bar-Shalom [Bar-Shalom88].

It is important to keep the tracking model of lowest order possible to maintain accuracy. Information theory states that a piece of data contains a finite amount of information and distributing that amongst a large set of estimates results in a smaller proportion of the information being assigned to each. This increases estimation errors [Maybeck79]. However, a low order tracking model will not be able to track a target with higher order dynamics. Conversely, using a higher order tracking model for a target with low order dynamics may result in poor tracking performance due to the higher order terms.

The change in tracking model can be performed dynamically and is required when the target being tracked undergoes some form of manoeuvre. In such situations, a low order model can be used until a manoeuvre is detected and then a change to a higher order one made. This is known as variable dimension filtering [Bar-Shalom88]. When the higher order term in the model is deemed insignificant it may be removed from the model, thus reducing the order of the model.

The manoeuvre is seen through a 'large' innovation component of the Kalman filter. The innovation is the difference between predictions and measurements, and can be monitored for significance. A fading memory average of the normalised innovations squared is commonly used for this :-

$$\rho(n) = \alpha\rho(n-1) + \nu'(n)\ \mathbf{S}^{-1}(n)\ \nu(n)$$

where $\nu(n)$ is the innovation from the Kalman filter, S(n) is the error covariance of the innovation and α is a 'forgetting' factor giving an effective memory of $\frac{1}{1-\alpha}$. If the value of $\rho(n)$ exceeds a certain threshold, then a manoeuvre is deemed to have taken place and the order of the model can be increased.

Similarly a test for significance of the highest order term, $\mathbf{x_a}(n)$ in the tracking model, can be performed. If the value of $\rho(n)$ falls below a certain threshold then the model can be reduced to a lower order one. Such that :-

$$\rho(n) = \alpha\rho(n-1) + \mathbf{x_a}'(n)\ \mathbf{P_a}^{-1}(n)\ \mathbf{x_a}(n)$$

where $\mathbf{x_a}(n)$ is the highest order term (for both the x and y dimensions) and $\mathbf{P_a}(n)$ is its respective covariance matrix. Each of the thresholds on $\rho(n)$ are chosen from chi-squared distribution tables. The values depend upon the number of degrees of freedom and the probability required for a correct hypothesis. Chi-squared testing is discussed by Bar-Shalom [Bar-Shalom88]. Manoeuvres will not be discussed further.

6.2.2 Data Association

Both the dense and sparse methods of feature tracking are termed as correspondence methods in the review by Vega [Vega89]. This is derived from a matching operation, between frames, for correspondence and can lead to one of the most computationally demanding elements of a tracking system. This processing is known as 'data association' in the target tracking domain and is the problem of associating measurements with previously tracked data.

Correlation-based matching can be used in dense methods for pixel correspondence. An example is the Sum of Squared Differences (SSD) method of Anandan [Anandan87]. It integrates the squared intensity difference between two shifted images over a small area to obtain an error measure :-

$$e(\Delta x, \Delta y; x, y) = \sum_{i=-m}^{m} \sum_{j=-n}^{n} w(i,j)[I_i(x-\Delta x+i, y-\Delta y+j) - I_{i-1}(x+i, y+j)]^2$$

where I_i and I_{i-1} are two consecutive image frames, and w(i,j) is an optional weighting function. The summations are performed over a finite window indicated by the bounds on the summations. The error measure, e( ), is calculated for a number of translations of the second image, compared to the first, indicated by Δx and Δy. An error surface results for each pixel location, the minimum of which represents the best correspondence, and whose shape can be used to determine its covariance. A poor match produces a flat surface, a good match produces a ‘peak’ minimum [Anandan87]. However, such techniques are rotation and scale variant and can produce large errors in the matching process.

For feature based methods, a situation can exist where the features extracted in the current frame can correspond with any of the features tracked from previous frames. A large combinatorial problem can result. If there are m_k measurements in the current frame and n tracks, then the number of permutations possible is given by :-

$$\frac{(m_k)!}{(m_k - n)!}$$

This can soon lead to a large number of association possibilities, even with a modest numbers of measurements and targets being tracked.

This problem can be overcome by the use of 'validation gating'. Validation gating limits the search space in which a measurement can be correctly hypothesised as originating from a particular target. The validation gate is the region in which the correct measurement will occur with a high probability, and is similar to limiting the search space in the SSD correlation window. When tracking a target's position, the validation gate is centred around its predicted position. The size of the gate is determined by the covariance of the predicted position and by the probability required to include the correct measurement.

The validation gate can be defined from the prediction outputs of the Kalman filter as

$$\nu'(k)\ S^{-1}(k)\ \nu(k) < \gamma$$

where $\nu(k)$ is the Kalman filter innovation, $S(k)$ is the innovation covariance, and the constant γ is chosen from chi-squared distribution tables which gives a probability of the correct measurement occurring within the validation gate. The constant γ is typically chosen to give a probability mass of 99%. Any measurement within the validation gate can be associated with the target being tracked, others outside cannot. A two-dimensional example is shown in Figure 6.2 for two targets, T_1 and T_2. The measurements z_1, z_2, z_3, are validated for target T_1, and the measurements z_2, z_4 are validated for T_2. The measurement z_5 is not validated for either.

There remains however, a strategy to determine which of the validated measurements should be associated with which of the targets under track. The set of validated measurements may consist of the correct measurement, measurements from other targets or from false alarms, including background noise and clutter. Several methods exist for the data association strategy, including a simple nearest-neighbour approach, an optimal Bayesian approach considering all possible associations through the time, and simplifications considering the previous n frames or just the current frame. Some of these approaches are described below.

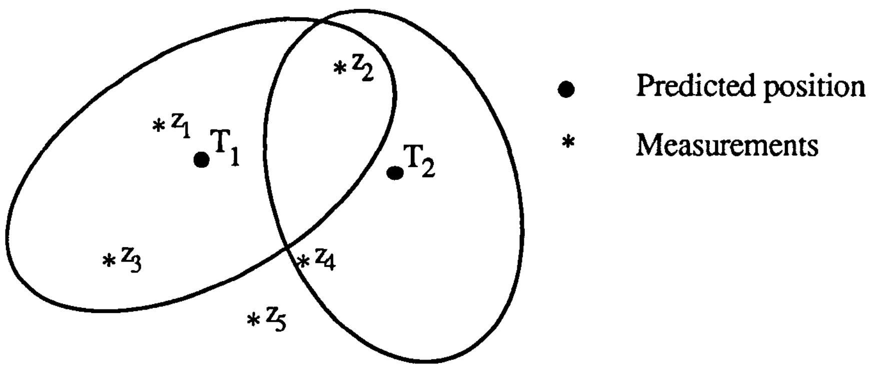


Figure 6.2 - Example of validation gating in two dimensions.

6.2.2.1 The Nearest Neighbour Standard Filter

The simplest approach for data association in low density tracking situations is to use a technique known as the Nearest Neighbour Standard Filter (NNSF) [Bar-Shalom88]. In the NNSF, the measurement nearest to the predicted position is taken as being the correct measurement. The Mahalanobis distance, between the predicted object position, $\hat{z}(n|n-1)$, and each of the detection measurements, $\{z_i(n)\}$, in the validation gate, is computed. The measurement, corresponding to that giving the minimum Mahalanobis distance, is chosen as the correct measurement. The Mahalanobis distance is similar to the validation gate definition :-

$$D^2(z_i(n)) = [\, z_i(n) - \hat{z}(n|n-1)\,]' \; S^{-1}(n) \; [\, z_i(n) - \hat{z}(n|n-1)\,]$$

Thus, the error covariance information, S(n), is used such that those dimensions in which there is small ambiguity are given a greater weighting than those with a larger ambiguity. The major shortfall of the NNSF occurs when the nearest measurement is not the correct measurement. This can lead to a significant effect on the tracking performance.

6.2.2.2 The optimal Bayesian approach

The optimal Bayesian approach considers the combinations of all measurements from time zero to the present time, rather than the associations of the latest set of measurements. The total number of measurement histories at time n is :-

$$L_n = \prod_{j=1}^{n} (1 + m_j)$$

where m_j is the number of measurements at time j. Note that the extra 1 within the above equation is due to the possibility that none of the measurements is correct. A separate Kalman filter is used to produce state and covariance estimates for each possible measurement history. A weighted Bayesian estimate can be obtained from the conditional mean of the state. At time n this is given by :-

$$\hat{x}(n|n) = \sum_{i=1}^{L_n} \hat{x}^i(n|n)\,\beta^{n,i}$$

where $\beta^{n,i}$ is the probability that the ith measurement history is the correct one. The computation and memory requirements increase exponentially.

Sub-optimal algorithms use an N-scan method, combining measurement histories which are identical over the previous N time periods. The number of expected measurement histories required to be processed and stored is approximately :-

$$L_n = \prod_{j=1}^{N} (1 + E[m_j])$$

where $E[m_j]$ is the expected number of measurements in the jth frame.

6.2.2.3 The Probabilistic Data Association Filter

The algorithm corresponding to the situation where the N-Scan is equal to zero, is the Probabilistic Data Association Filter (PDAF). This was first introduced by Bar-Shalom [Bar-Shalom75]. The PDAF takes a weighted summation of the innovations, between all validated measurements and the predicted position, for each target. The weighted innovation used in the Kalman filter is given by :-

$$v(n) = \sum_{i=1}^{m_n} \beta_i(n) v_i(n)$$

where m_n is the number of measurements within the validation gate at time n, $v_i(n)$ is the innovation between the prediction and the ith measurement, and $\beta_i(n)$ is the probability that the ith measurement is the correct measurement. The error covariance associated with the updated estimate also differs from that of the standard Kalman filter. The updated covariance, $\mathbf{P}(n|n)$, in the PDAF is given by :-

$$\mathbf{P}(n|n) = \beta_0(n)\mathbf{P}(n|n-1) + [1-\beta_0(n)]\mathbf{P}_c(n|n) + \mathbf{P}_a(n)$$

where $\mathbf{P}(n|n-1)$ is the predicted covariance weighted by the probability, β_0, that none of the measurements is the correct one. $\mathbf{P}_c(n|n)$ is the covariance given by the standard Kalman filter weighted by the probability of the correct measurement existing, and $\mathbf{P}_a(n)$ increases the covariance representing the ambiguity in the measurement origin. This is given by :-

$$\mathbf{P}_a(n) = \mathbf{W}(n)\left[\sum_{i=1}^{m_n}\beta_i(n)v_i(n)v_i'(n)-v_i(n)v_i'(n)\right]\mathbf{W}'(n)$$

where $\mathbf{W}(n)$ is the Kalman gain. The probabilities, $\beta_i(n)$ and $\beta_0(n)$, are given by :

$$\beta_i(n) = \frac{e_i}{b+\sum_{j=1}^{m_n} e_j} \qquad \text{and} \qquad \beta_0(n) = \frac{b}{b+\sum_{j=1}^{m_n} e_j}$$

where

$$e_i = \exp\left\{-\frac{1}{2}v_i'(n)S^{-1}v_i(n)\right\}$$

and the constant b is given by :

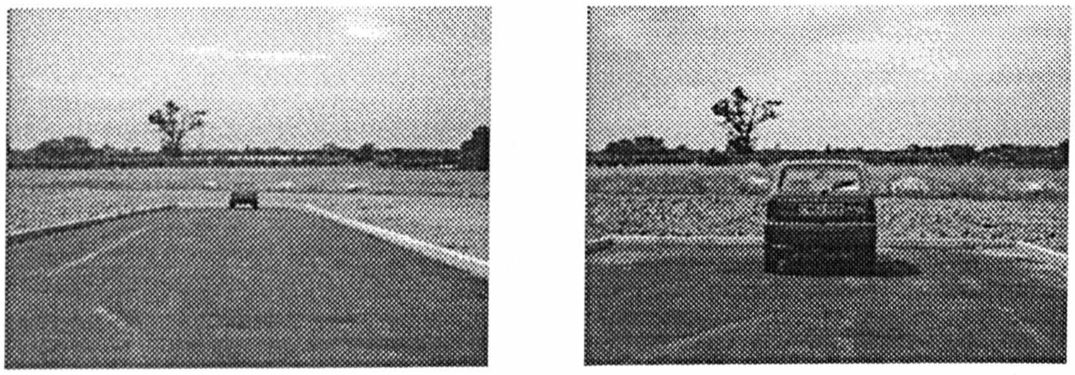
$$b = \left(\frac{2\pi}{\gamma}\right)^{n_z/2} m_n C_{nz}\frac{(1-P_D P_G)}{P_D}$$

Further γ is a chi-squared number, n_z is the dimension of the measurements, $\mathbf{z}$, and C_{nz}, P_D, P_G are constants.

Two tracking applications are now considered. The first is concerned with the tracking of an object in a low density environment. The second considers a general multi-target tracking situation which may arise in both computer vision and target tracking scenarios. The NNSF data association technique is used in the first application and the PDAF is used in the second.

6.3 Tracking Application 1 - Object Tracking

The first application is concerned with the tracking of an object across the image plane as a camera moves towards it. Tracking is performed on the measured position of an object on the image plane, in the x and y dimensions. The known forward component of the camera motion is used in a novel object size-tracking formulation through the image sequence. The size formulation enables estimates of the objects distance from the camera and its physical size to be made.



a) Range = 39.5m b) Range = 9.5m

Figure 6.3 - Two images from a sequence approaching a car.

The image processing used on each image within the sequence follows the classical processing flow of enhancement, detection, segmentation, feature extraction and classification. The early stages are global filtering operations, and the later stages are local to individual objects found within the image. The detection and segmentation stages of the image processing are described in Section 6.3.2. Both techniques are size based, requiring an estimate of the objects image size.

The processing flow is shown in Figure 6.4. The first operation on the nth image is a global edge orientation calculation followed by a global object detection whose predicted size, $\Delta\hat{z}(n|n-1)$, is used from previous frames. The set of detected object's centroids, $\{z_d(n)\}$, are compared with the predicted position of the object, $\hat{z}(n|n-1)$, and one detection, $z(n)$, is selected using the NNSF data association filter. The segmentation operation uses the detection centroid and the size prediction. From the segmentation a centroid, $z_s(n)$, and size, $\Delta z(n)$, are measured. These measurements are incorporated into the tracking filters to produce estimates of the object's location and size, and predictions for the next frame.

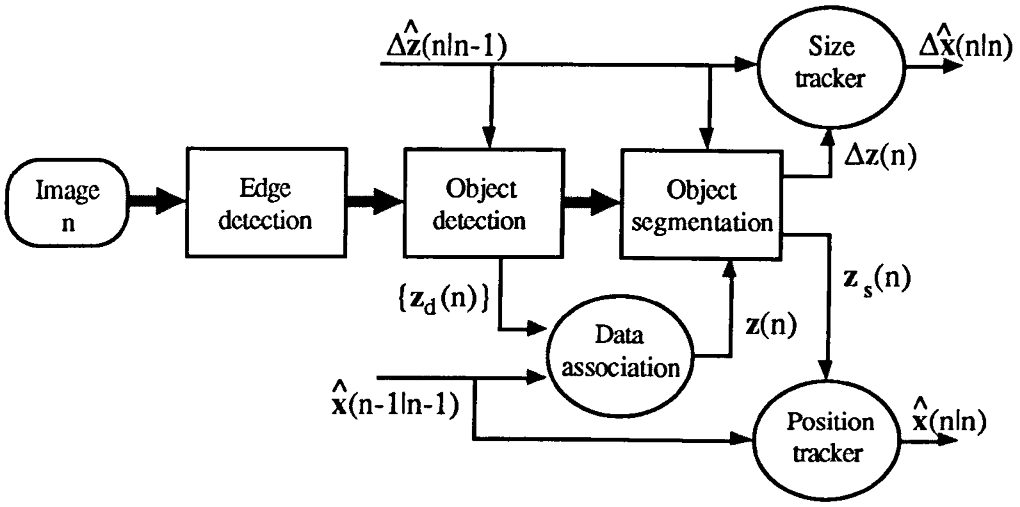


Figure 6.4 - Processing flow showing image processing, size and position tracking.

The measurement noise variances are also incorporated into the Kalman filters, but are not shown in Figure 6.4. The tracking models used for the size and position tracking are described below in Section 6.3.1. This is followed by the main stages of the image processing, that is, the object detection and object segmentation stages.

6.3.1 Object tracking models

The object tracking uses two separate filters, the first for the tracking of the object's position across the image plane and the second to track the object's size. The size tracking is used when the object changes its depth from the image plane as is the case when the camera is moving. The tracking models used are described below.

6.3.1.1 Position tracking model

The tracking of the objects consists of an image position and velocity model in both x and y dimensions. Image plane co-ordinates are used for the models. The position tracking model here, was given in Section 6.2.1, but is enlarged to incorporate both x and y dimensions within a single model.

The initialisation of the position tracking filter is performed using the centroids of the object segmentation in the first two images. The first centroid is used as the state estimate of the position, and the difference between the two centroids, divided by the frame time, is used as the estimate of velocity. The initial covariance matrix, **P**(0), uses the centroid measurement variance, σ_{vx}^2 (from the image processing) :-

$$\mathbf{P}(0) = \begin{bmatrix} \sigma_{vx}^2 & \frac{\sigma_{vx}^2}{T} \\ \frac{\sigma_{vx}^2}{T} & \frac{2\sigma_{vx}^2}{T^2} \end{bmatrix}$$

6.3.1.2 Size tracking model

The tracking of an object's size is central to both the object detection and object segmentation algorithms used in the image processing. The tracking of size has been applied to a number of image sequences [WSTL90, Atherton91], and its formulation is described below. The relationship between the image size of an object and its physical size can be seen in Figure 6.5. Note that size of an object here, refers to its width, or height, for instance.

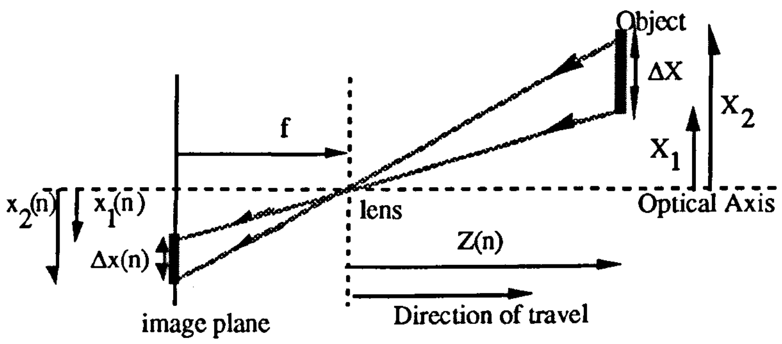


Figure 6.5 - The Imager showing the relationship between the size of an object and its depth from a camera.

Using similar triangles the size of the object on the image plane is related to its physical size by :-

$$\Delta x(n) = \frac{\Delta X}{Z(n)} fk = \frac{\Delta X}{Z(0) - S(n)} fk$$

where the distance between the object and camera plane in frame n is Z(n) metres, or the initial range, Z(0), minus the distance travelled towards the object, S(n). The physical size of the object is ΔX metres, the object size on the image plane is $\Delta x(n)$ pixels, k is a constant relating pixel distances to metres, and f is the focal length of the camera in metres.

The use of relative position (size) removes the need to find the focus of expansion, making the technique immune to small random rotations of the camera. The variation in depth of the object is assumed small with respect to the distance from the camera. A similar formulation was used by Williams [Williams88] but considers only two successive frames at a time.

The relationship between the image size measurements and the range to the object is non-linear and requires the use of the Extended Kalman filter for recursive estimation. By using the inverse size, the tracking becomes linear but requires a transformation of each measurement. The state-space representation of this requires two states, a and b :-

$$y(n) = \frac{1}{\Delta x(n)} = \frac{Z_0}{kf\Delta X} - \frac{S(n)}{kf\Delta X} = a + bS(n)$$

The system is modelled by the two states, a(n) and b(n), with a single measurement of the inverse image size :-

$$\mathbf{x}_s(n) = \begin{bmatrix} a(n) \\ b(n) \end{bmatrix} = A_s\,\mathbf{x}_s(n-1) + \begin{bmatrix} 0 \\ 1 \end{bmatrix} v_y(n)$$

$$= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} a(n-1) \\ b(n-1) \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} v_y(n)$$

$$z(n) = \mathbf{h}(\mathbf{x}_s(n)) + w_s(n)$$

where

$$\mathbf{h}(\mathbf{x}_s(n)) = \frac{1}{a(n) + S(n)b(n)}$$

The additive plant noise process, $v_y(n)$, is approximated by zero mean Gaussian noise with variance $\sigma^2_{vs}(n)$ representing the noise in the inverse image size due to uncertainties in the forward camera motion. The additive measurement noise process, $w_s(n)$, has variance $\sigma^2_{ws}(n)$ and varies with true image size.

The size Kalman filter requires an initial estimate of both the states, $x_s(0)$, and the covariance matrix, P(0). The state is a vector function of the estimates of the initial distance between the object and camera, Z_0, and the object's physical size, ΔX. The covariance matrix for x is:

$$P(0) = J_f^T P_b J_f$$

where P_b is the covariance matrix of $\mathbf{z} = [Z_0, \Delta X]$:

$$P_b = \begin{bmatrix} p_{\Delta X} & 0 \\ 0 & p_{Z_0} \end{bmatrix}$$

J_f is the Jacobian of the vector function with respect to $\mathbf{z}$, and $p_{\Delta X}$, p_{Z0} are the variances in the initial estimates of the object physical size and range respectively.

The initial covariance matrix for the size Kalman filter is given by :-

$$P(0) = \begin{bmatrix} \left(\frac{Z_0}{kf\Delta X}\right)^2 \left(\frac{p_{\Delta X}}{\Delta X^2} + \frac{p_{ZZ}}{Z_0^2}\right) & -\frac{Z_0}{(kf\Delta X)^2}\frac{p_{\Delta X}}{\Delta X^2} \\ -\frac{Z_0}{(kf\Delta X)^2}\frac{p_{\Delta X}}{\Delta X^2} & \frac{1}{(kf\Delta X)^2}\frac{p_{\Delta X}}{\Delta X^2} \end{bmatrix}$$

The size tracking model described has been given in terms of a single size parameter such as an object's width or an object's height. However, the model could be enlarged so as to incorporate two or more size measures.

6.3.2 Object detection and segmentation processing

The object image processing flow used here consists of the following stages. Initially, a Sobel edge detection is performed to produce edge orientation and magnitude information. This is used in a size based detection operation. The output of the detection undergoes a component labelling [Cypher90], a centroid calculation and an NNSF data association operation. The associated centroid is used by the segmentation operation. The output of the segmentation is a measurement of the objects position and size, which are then used to update the tracking filters. The sized based image processing for object detection and object segmentation are described below.

6.3.2.1 Object Detection

The object detection method used in this application is based upon the Hough Transform. Although this is a method for line detection, it can be extended to find more complex geometric shapes as shown by Duda and Hart [Duda72]. They gave a parametric representation of circles such that peaks in the resulting parameter space represented the centre of circles. This is commonly referred to as the Circular Hough Transform, and the parametric representation is given by :-

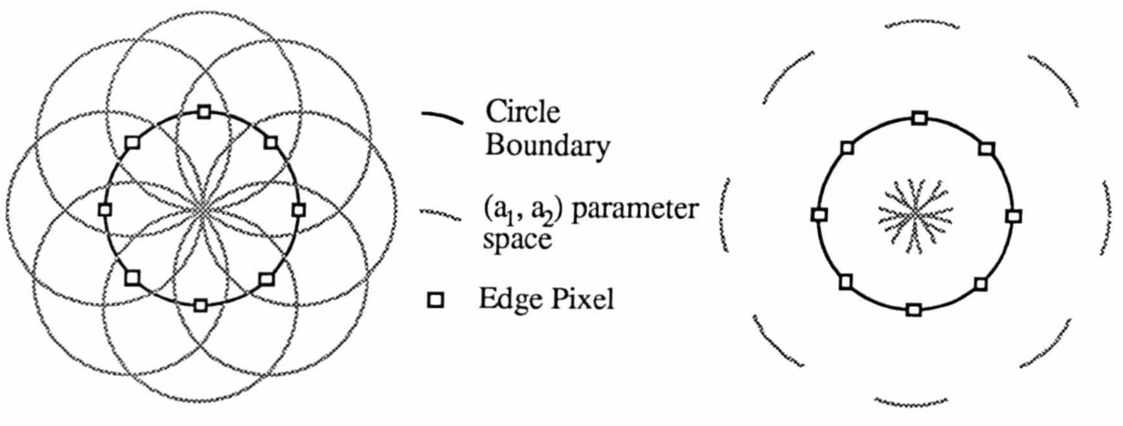
$$(x_i - a_1)^2 + (y_i - a_2)^2 = r^2$$

where (x_i, y_i) are image edge points, (a_1, a_2) is the centre of the circle, and r is the radius of the circle. The resulting parameter space is three dimensional. Each edge point contributes a circle of a radius r into each two dimensional plane of (a_1, a_2).

The parameter space of the circular Hough transform remains local to each edge point, thus simplifying parallel implementations. However, it is computationally expensive since the operation is performed for each edge point at all radii. A modification is to limit the contribution of each edge point in the parameter space to an arc around its orientation [Kimme75]. Although this modification reduces the complexity, the algorithm still has to be repeated for different values of the radius, r.

An example of the circular Hough transform, at a single radius, is shown in Figure 6.6a, and the modification using edge orientation is shown in Figure 6.6b. Eight edge points are highlighted on both circles. In both cases, a peak in the parameter space occurs at the centre of the circles. This peak represents a maximum of intersections of the circles drawn from each edge point. The modified circular Hough Transform shown in Figure 6.6b uses both the edge orientation and the orientation + π. This enables an object which is either lighter, or darker, than the background to produce the same peak in the parameter space.

Further modifications can be made to the Circular Hough transform to reduce its complexity. The method of Minor and Sklansky [Minor81] reduces the three dimensional parameter space to two dimensions while also quantising the orientations to eight directions. This method is known as the Spoke filter. It makes much use of bit-operations and is suited to bit-serial SIMD arrays [Atherton90].



a) Circular Hough Transform b) Using edge orientation

Figure 6.6 - Example detection of circles using the circular Hough transform.

The Spoke filter propagates a spoke out into the parameter space (a_1, a_2) in the direction of the edge point (and the orientation + π) between a minimum radius, R_{min}, and a maximum radius, R_{max}. A peak in the Spoke parameter space represents the centre of an object, which is approximately circular, and has a size within the range of sizes specified by the spoke radii. The values of R_{min} and R_{max} can be taken to be a function of the size tracking filter. In the processing used here, it was assumed that R_{min} = (size estimate)/4 and R_{max} = 1.2*(size estimate)/2.

The spoke filter is shown in Figure 6.7, where the centre point in the parameter space can be contributed to from any of the edge points within the shaded region.

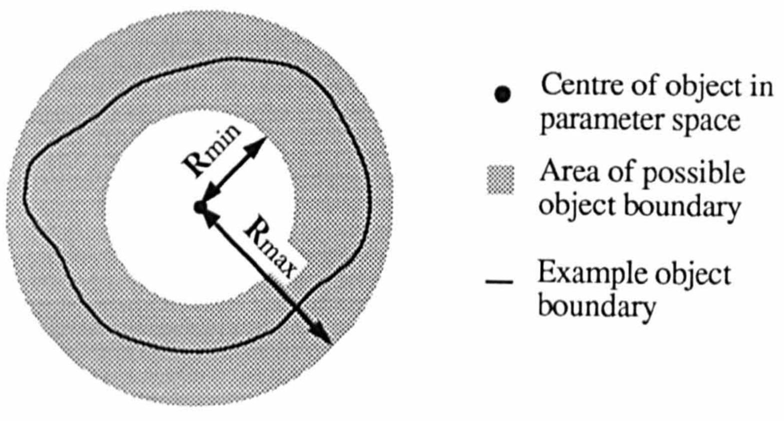
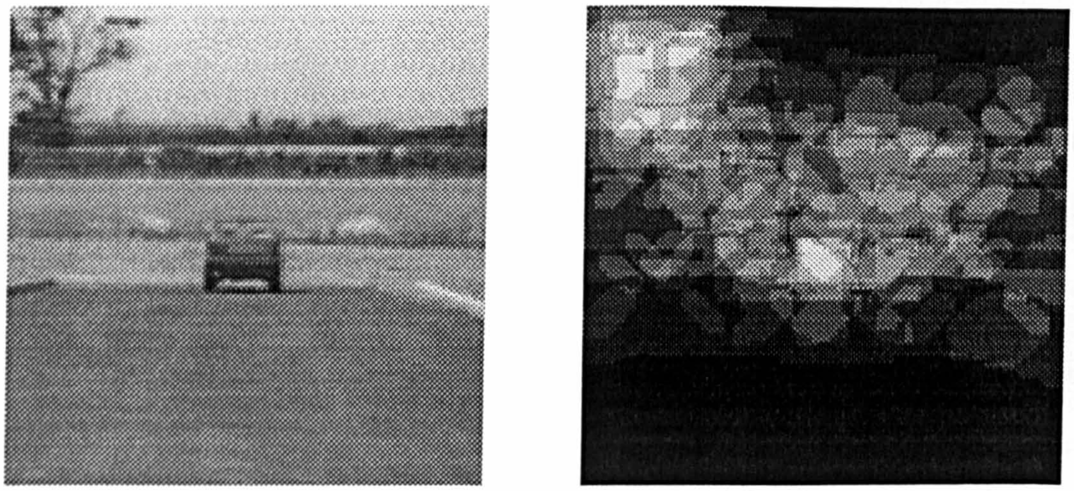

R_{min}
R_{max}
● Centre of object in parameter space
▨ Area of possible object boundary
Example object boundary

Figure 6.7 - Edge points which contribute to a single parameter space location in the Spoke filter.

The Spoke filter as implemented by Minor [Minor81] uses an edge detected image which undergoes a threshold operation to retain the top T percent of edge points (typically T = 10%). A maximum of eight spoke intersections can occur in the parameter space, one for each of the quantised orientations. The regions within the parameter space, containing a seven or eight spoke intersections, can be used as detected object positions. When the contrast of the object is not solely lighter (nor darker) than its background, spokes can be radiated in both the direction of their edge orientation, and their edge orientation + π.



a) Original image b) Output from the Spoke filter

Figure 6.8 - Example of the Spoke filter.

An example of the Spoke filter is shown in Figure 6.8. The image shown in Figure 6.8a is the original image and the image in Figure 6.8b is the output from the Spoke filter. Eight Spoke intersections are shown as white and zero intersections are shown as black. It can be seen that there are eight Spoke intersections at the centre of the car and also at the tree in the background.

Approximations in the orientation output from the edge detector can be made to simplify the computation requirements. The city block method of $|\Delta x| + |\Delta y|$ can be used for the approximate edge magnitude. The quantised edge orientation ideally represents 45 degrees of the total orientation range requiring an arctan calculation. However, an approximation can be made requiring six comparisons and twelve conditional statements, shown below. Note that the variables `up`, `dn`, `ri`, `le`, `yp`, `xp` are logicals and $\Delta x, \Delta y$, ax and ay are byte values. The resulting quantisations represent a range in θ of either 53.2 or 36.8 degrees depending upon the orientation.

```
up = (Δx ≥ 0),     dn = (Δx < 0),    ri = (Δy ≥ 0),     le = (Δy < 0)
ax = abs(Δx),      ay = abs(Δy),     yp = (ay ≥ 2ax),   xp = (ax ≥ 2ay)

            if (ri and up)
                 if (yp)              then  θ = 1
                 else if (xp)         then  θ = 3
                 else                       θ = 2
            if (ri and dn)
                 if (yp)              then  θ = 5
                 else if (xp)         then  θ = 3
                 else                       θ = 4
            if (le and up)
                 if (yp)              then  θ = 5
                 else if (xp)         then  θ = 7
                 else                       θ = 6
            if (le and dn)
                 if (yp)              then  θ = 1
                 else if (xp)         then  θ = 7
                 else                       θ = 8
```

6.3.2.2 Object Segmentation

A segmentation operation follows the detection, and is localised to the regions in the image in which an object has been detected. The segmentation algorithm used has many similarities to the spoke detection filter above. It uses edge magnitude and orientation information and is able to segment objects which are approximately convex in shape within a given range of size. This segmentation technique is termed the Spoke Segmentor [Atherton90]. A similar technique has been reported by Golston and Moss [Golston90].

The centroid of the detection, produced by the Spoke filter, is used as the starting point of the spoke segmentor. From this point a number of spokes are radiated out at angles of $2\pi/N_S$ across the edge image (where N_S = number of segmentation spokes). These spokes are active only from a distance r_{min} to r_{max} from the detection centroid. The maximum edge magnitude, along the active region of the spoke, is taken as an object boundary point.

The orientation of the boundary point within a segmentation spoke angle is expected to be either in the same direction, or different by π, from the angle of the spoke. The two angles are possible so that objects lighter, or darker, than the background can be segmented.

A set of boundary points result from this segmentation process, at most one for each of the segmentation spokes. In high clutter situations some of these points may not lie on the object boundary but instead on other edge features which may be close to, or within, the object. Post processing can reduce these effects, in the form of a 3x1 median filter [Golston90] which removes spike differences on the boundary points.

The median filter can be performed by collating the boundary points into a 1D array, indexed by spoke segmentation angle. These values represent the distance between the centroid and the most likely boundary at each angle. The median filter is performed on this array, assuming that the array is circular i.e. that the N_sth value is adjacent to the first value. This operation can be performed using the Vector mode of the M-SIMD array within the WPM.

A convex hull operation on the resulting boundary points produces an approximate boundary contour. The convex hull can be thought of as stretching an elastic band around the set of points. Only those points which lie on the contour of the band are part of the convex hull. The convex hull of the boundary points can be used to extract features of the object for classification purposes, or to extract the object's pixels from the background.

A simplified convex hull algorithm may be used since the ordering of the points from the segmentation is known, i.e. they are indexed in theta. A suitable algorithm is given by [Chen89] which considers every boundary point along with its two neighbouring points in the clockwise direction. For a candidate convex hull point, the gradient of the line between the previous point and itself must be greater than that between itself and the next point. That is :-

$$\frac{y_1 - y_0}{x_1 - x_0} > \frac{y_2 - y_1}{x_2 - x_1} \qquad \text{or} \qquad (y_1 - y_0)*(x_2 - x_1) > (y_2 - y_1)*(x_1 - x_0)$$

where (x_0, y_0) is the previous point, (x_1, y_1) is the candidate point and (x_2, y_2) is the next point. Points which do not satisfy this are eliminated from the list. The algorithm iterates until all points remaining lie upon the convex hull, i.e. no points are eliminated in an iteration.

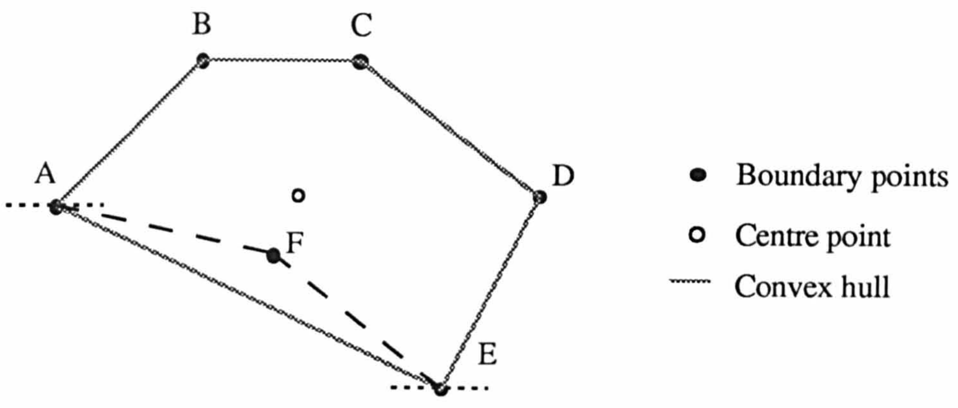
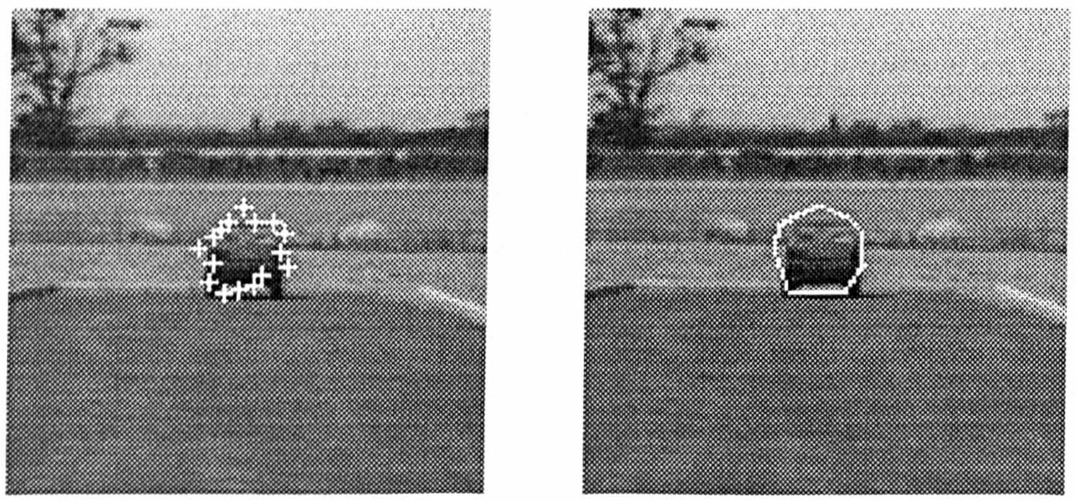


Figure 6.9 - Example of the Convex hull algorithm.

An example is shown in Figure 6.9 for 6 boundary points - A..F. On the first iteration, the gradient of AB is larger than that of BC, so point B is not eliminated. However, the gradient of EF is less (more negative) than that of FA and so point F is eliminated. Only two iterations of the algorithm is required in this example.

The image from Figure 6.8a is used to illustrate the Spoke segmentation and the convex hull operation in Figure 6.10. Figure 6.10a shows the position of 16 boundary points using the centroid of the detected position of the car from Figure 6.8b. The convex hull of these boundary points is shown in Figure 6.10b.



a) Boundary point output b) Convex hull of boundary points

Figure 6.10 - Example output from the object segmentation.

6.4 Tracking Application 2 - Generic Target Tracking

The second tracking application can be considered as representative of a generic multiple target tracking application. It considers the situation where the targets being tracked are viewed as small regions on the image plane, e.g. from space-based platforms, in defence applications, or in air traffic control situations. The number of targets can range from 10's to 10,000's and more, especially in the space based systems when background events are also considered. The processing flow follows that shown in Figure 6.11. The image processing applied to each input image is application specific, producing point measurements which represent the positions of possible targets.

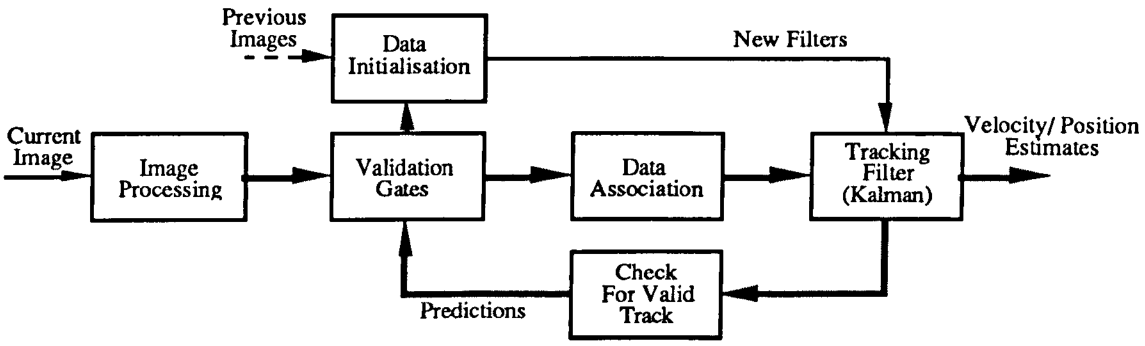


Figure 6.11 - Processing flow in a generic multiple-target tracking environment.

This situation differs from the first example application, in that an increase in complexity occurs for the tracking operations, but the image processing requirements are typically less. One tracking filter is required for the position of each of the targets being tracked. Each has a data association operation between the predictions (output from the tracking filters) and the measurements in each new image frame.

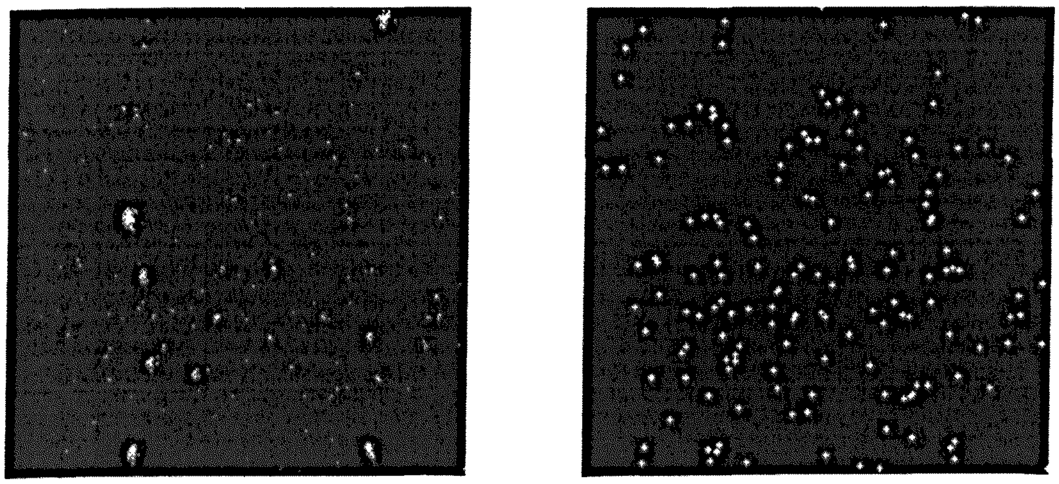
The tracking models, used in this example application for the position tracking, are the same as those used in the first application. That is, a position and velocity tracker, in both x and y dimensions combined within a single model. The PDAF filter was used for the data association between the measurements and the predictions from each filter. However, other models of tracking and data association methods are equally applicable. Example image processing operations, which can be applied to input imagery to produce point measurements, are described below.

6.4.1 Image processing operations

The first is a simple thresholding technique which can be used when a uniform background exists, which is darker than any object in the image (e.g. space based systems). The second classifies each image pixel dependent upon its relative value in comparison to its neighbours in order to find local maximums. The third detects corner features within images. The output from each method produces point positions which may be incorporated into tracking filters.

6.4.1.1 Image thresholding

A simple threshold of the image may be undertaken if the objects contained within it are solely lighter or darker than the background, thus segmenting the objects. Each isolated object may then be uniquely labelled and the centroid of each calculated. The resultant image contains pixels which are set at the centroid of each object found, forming point measurements. An example is shown in Figure 6.12 and depicts a single image from a simulated sequence of a camera passing through a meteor shower. The original image is shown in Figure 6.12a and the processed image with point measurements is shown in Figure 6.12b.



a) Original image b) Processed image

Figure 6.12 - Example image from a meteor sequence.

6.4.1.2 Monotonicity operator

The point extraction method used by Kories [Kories86], finds the local maxima of image pixel values and has been used to estimate displacement vector fields between frames. It uses an operator, termed a 'monotonicity operator', which is invariant under grey-scale transformations. The operator classifies each pixel into one of eight categories dependent upon the number of neighbouring pixels (8-way connected) which have a lower grey-scale value. The eight neighbouring pixels are chosen by the mask, shown in Figure 6.13. Each neighbouring pixel used is separated from the centre pixel by a distance L horizontally and/or vertically.

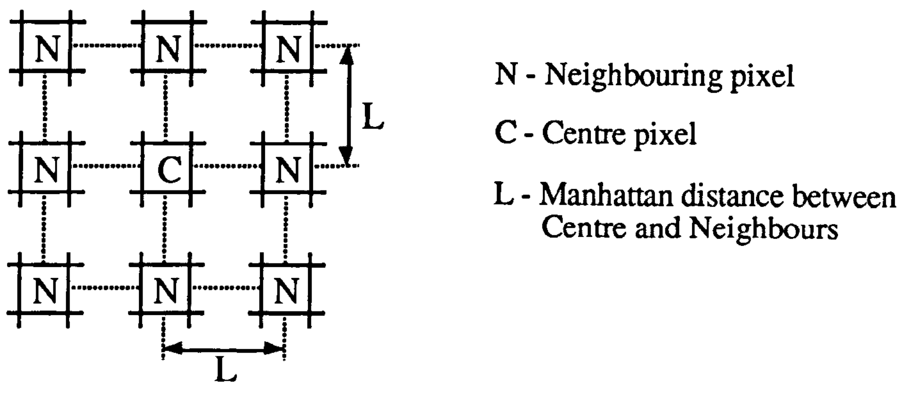


Figure 6.13 - The monotonic operator of Kories.

A pixel which is a local maximum receives a classification of eight under this scheme. Adjacent pixels in the same category undergo a component labelling process and centroid calculation. Regions which are of size less than 10 pixels are eliminated. The resultant centroids are matched between frames using their classification, area, and position as feature descriptors. Although the method of Kories only considers two consecutive frame for the displacement calculation, the method could easily be extended to image sequences and, using appropriate Kalman filter models, an estimation of the image displacements made.

6.4.1.3 Corner detection

Corners are convenient image features whose position is invariant to both image grey-scale and image rotation. A combined edge and corner detector has been described by Harris [Harris88] and also by Noble [Noble87]. Harris calculates the image gradients at each pixel location in both the horizontal and vertical directions, denoted by I_x and I_y, using the simple convolution masks of [-1 0 1] and $[-1\ 0\ 1]^T$ respectively. The values of I_x^2 and I_y^2 and I_xI_y are calculated and smoothed using an nxn Gaussian convolution kernel. The values are used to from a 2x2 matrix, **M** :-

$$\mathbf{M} = \begin{bmatrix} I_x^2 & I_xI_y \\ I_xI_y & I_y^2 \end{bmatrix}$$

The matrix **M** effectively describes the shape of the auto-correlation function, about each pixel within the image, with the eigenvalues of the matrix being proportional to the principal axis of the auto-correlation function. A corner is found if both the eigenvalues are large, thus indicating that the auto-correlation function is sharply peaked. A measure, R, is used by Harris to give a *cornerness* and an *edgeness* factor :

$$R = \text{Det}(\mathbf{M}) - k\ \text{Trace}(\mathbf{M})$$

Note that the Det(**M**) is the product of the two eigenvalues of **M** and the Trace(**M**) is the summation of them. A corner is indicated if the value of R is positive and an eight-way local maximum (in a 3*3 search). An edge is indicated if the value of R is negative and is a local minima in horizontal or vertical directions depending upon the largest of the local edge gradients (I_x, I_y) respectively. An example output from the Harris corner detector is shown in Figure 6.14. The black regions indicate both corners (black points) and lines (black lines). The coloured regions indicate locations of possible corners (dark grey) or possible lines (light grey).

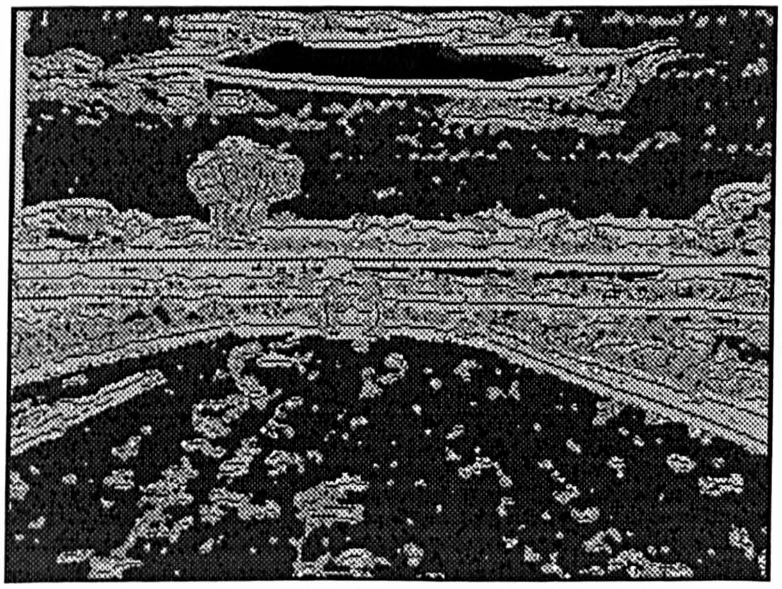


Figure 6.14 - Example image showing the categories of the Harris Corner detector.

The extraction of corners by Harris has been used to estimate camera ego-motion [Harris87] and also to extract 3D information from images using corner and edge information [Stephens89].

6.5 Performance on the WPM

The performance of the example tracking applications on the WPM is considered in two parts: firstly, the image processing, which is required to produce the features that are tracked; followed by the data association and the implementation of the tracking Kalman filters. The communication bandwidths within the WPM are such that it is impracticable to move image data between the M-SIMD and MIMD processor arrays and as such imposes the restriction that the images may only be processed on the SIMD array. The data association and tracking may be performed on either the M-SIMD or the MIMD arrays.

6.5.1 The image processing

The image processing used in both the tracking applications is listed in Table 6.1. All the operations listed are performed at each pixel location, in a data parallel fashion. The Spoke segmentation takes advantage of the WPM's M-SIMD operational autonomy and count capability to calculate peak edge magnitude along each of its radiating spokes. The convex hull calculation uses the output of the peak edge locations, arranged in vector mode on the SIMD array, within a WPM Cluster.

The image processing time, for the object tracking application, is the summation of the Sobel filter, the Spoke filter, the Spoke segmentor, the component labelling, the convex hull, and the centroid calculations. The image processing time for the generic tracking application is taken as the summation of the threshold, the component labelling and the centroid calculation. Note however, that the local processing of the spoke segmentation, component labelling, convex hull and centroid calculation are performed per object within a Cluster. Thus in the second application, where many centroids may need to be calculated within a Cluster, the time must be scaled by the expected number of targets within that Cluster.

Operation	Time (μs)	Comments
Sobel filter	44	City-block magnitude, and approximate orientation estimate
Spoke filter	65	R_{min}=5, R_{max}=8 [Atherton90]
Spoke segmentation	220	r_{min}=5, R_{max}=8, N_s=16 [Nudd92a]
Connected Component labelling	20	Regions size 12 pixels [Atherton90]
Convex hull	71	three iterations of the point elimination
Centroid Calculation	11	See Section 5.4.2.3
Threshold	3	
Monotonicity Operator	71	L=4
Corner detection	1200	assuming 16-bit intermediates

Table 6.1 - Image processing operations on the M-SIMD level of the WPM.

The time taken for the image processing operations on the M-SIMD array, within the WPM, is 431μs for the first application, and between 14μs and 1211μs for the second. In the first application it is assumed that there is at most one object, which fits fully, within a Cluster. In the second application it is assumed that the processing consists of either a threshold operation, or a corner detection operation, followed by, at most, a single centroid calculation within each Cluster.

6.5.2 Tracking operations

One tracking filter is required per target under track. This can potentially lead to a large number of filters. Each filter also requires a data association operation between the predictions and the measurements found on the image plane. The data association operation and the tracking filters are described below.

The data association used in the first application is the NNSF where a small number of objects are tracked, and the PDAF in the second application where many targets are tracked. However, the data association requires the prediction, $\hat{z}(n|n-1)$ and the prediction error covariance, S(n), from the Kalman filter. The NNSF returns the innovation term, ν(k), which is incorporated into the Kalman filter. The PDAF returns a weighted innovation term, $\nu(k)$, an associated error covariance term, $\mathbf{P}_a(k)$, and the probability of in-correct data association, β_0. The data-flow of these parameters is shown in Figure 6.15. It is not clear where the processing of the Kalman filters should take place. This in itself is a data parallel problem since many filters are processing in parallel. The processing of the Kalman filters is discussed later.

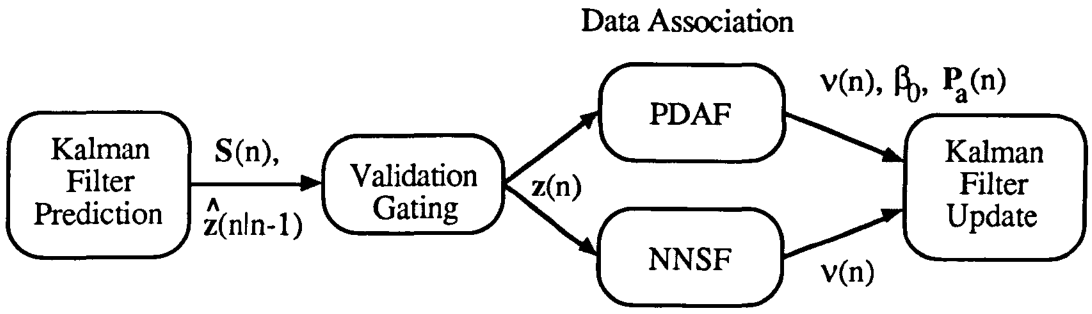


Figure 6.15 - The parameters passed between the Validation gating, PDAF or NNSF data association, and the tracking Kalman filters.

6.5.2.1 The data association

Both the NNSF and PDAF data association filters are used to associate the point measurements in the current frame with the targets already under track. The association is limited to the validation gate of the target, as discussed in Section 6.2.2. Note that the validation gate for each target under track is different. The validation of the measurements may be performed on the M-SIMD array.

If it is assumed that all SIMD PEs are labelled with their (x,y) position within the image, then the test to validate measurements may be performed in parallel across the array. This is done by simply broadcasting the predicted position and covariance from the tracking filters across the array. PEs which contain a point measurement, and lie within the validation gate, are validated measurements for the track under consideration. Subsequently, the association innovation, $\nu(k)$, and the innovation variance can be calculated.

On a conventional SIMD array, only one validation/association operation can be performed at once and so the operation is performed sequentially for all targets under track. However, the M-SIMD partition within the WPM, enables each operation to be limited to a small region of the whole measurement data. This enables the speed of the validation/association operation to be increased, by a factor equal to the number of Clusters within the WPM. This situation is shown in Figure 6.16a for four such validation regions.

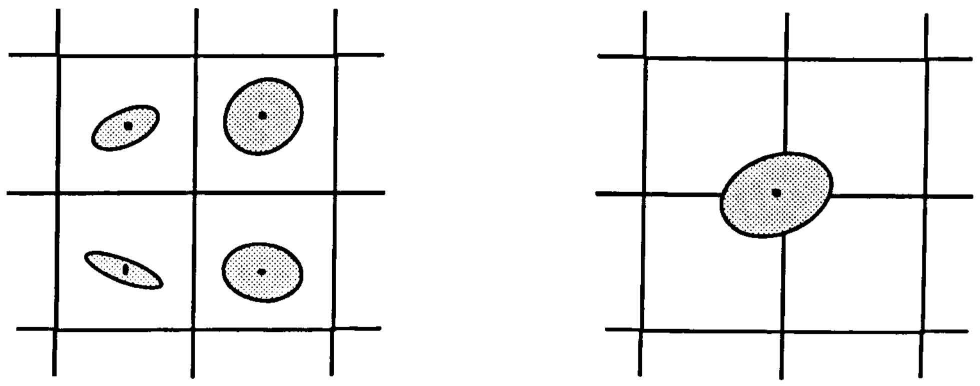


Figure 6.16 - Validation gates mapped onto four Clusters within the WPM.

The number of calculations operating in parallel across the M-SIMD array depends upon the size of the validation gate (defined by the chi-squared number and the covariance output from the tracking filter) and its position relative to the Cluster boundaries. The example shown in Figure 6.16b shows a validation gate across four Clusters. In such situations, all four Clusters may be synchronised so that they perform the same operation in parallel using the same validation gate parameters. The maximum speedup of M-SIMD partitioning over that of the SIMD will rarely be achieved, but would typically lie within the maximum and a quarter of this maximum. The maximum is the number of Clusters in the M-SIMD machine.

The innovation for the NNSF directly follows from the validation test by taking the minimum validation measure (see Section 6.2.2.1). The weighted innovation and variance term for the PDAF (see Section 6.2.3) can be calculated on the Transputer after the extraction of the validated measurements from the M-SIMD array.

6.5.2.2 The processing of the Kalman filters

The second major computational task required for target tracking is that of the Kalman filter processing, one per target under track and one iteration per time frame. Each Kalman filter calculation involves matrix/vector computations as shown in Appendix C. The largest matrix is of size nxn, where n is the number of states within the model. Each matrix operation requires the use of floating point arithmetic.

An analysis of the number of floating point operations required for one iteration of a tracking filter has been given by Pattipatti [Pattipatti90]. The total number of calculations required are repeated below, where n is the number of states within the model, and m is the number of measurements within the measurement model. For this, it was assumed that a subtraction takes the same amount of time as an addition, and a division takes the same amount of time as a multiplication (all in floating point).

$$\text{Multiplications :} \quad \frac{3}{2}\left(n^3 + n^2\right) + m\left(3\frac{n^2}{2} + 9\frac{n}{2} + 4\right)$$

$$\text{Additions :} \quad \frac{3}{2}\left(n^3 - n\right) + \frac{m}{2}\left(3n^2 + 5n + 1\right)$$

Further operations are required to incorporate the output of the PDAF with the update of the covariance matrix.

Although it is useful to know the number of operations to be performed at the floating point level, it does not illustrate any under-lying structure in the data which could be exploited by parallel operations. The Kalman filter equations form a set of matrix vector operations. The data parallelism, contained within the matrices, can be exploited on an array processor using matrix techniques as shown in Section 5.5. The number, type, and dimensions of matrix operations required on a single iteration of a Kalman filter are shown in Table 6.2. The operations are split into separate matrix addition, subtraction, multiplication and inversion operations, and include such operations as matrix outer products. The number of PEs required for each operation, such that one PE is used for a single matrix element is shown, along with the complexity of the operation, i.e. the number of iterations required for the matrix multiplication and inversion.

Matrix sizes (1^{st}, 2^{nd})	PEs required	add	subtract	multiply	invert
n^2, n^2	n^2	1	1	2	
n^2, nl	n^2			1	
n^2, nm	n^2			2	
nm, mn	n^2			1	
mn, nm	n^2			1	
mn, nl	n^2			1	
nm, m^2	nm			2	
nm, ml	nm			1	
m^2, m^2	m^2	1			1
nl, nl	nl	1			
ml, ml	ml		1		

Table 6.2 - Matrix/vector operations required on a single iteration of a Kalman filter with n states and m measurements.

In addition, two matrix transpositions are also required, on matrices of sizes n^2 and nm. To simplify the analysis it is assumed that the number of system states, n, is larger than the number of measurements, m, and that each matrix operation utilises either a single PE or a patch of n^2 PEs. This means that operations such as the transpose of a nm matrix is the same as the transpose of an n^2 matrix, and a multiplication between an n^2 and nm matrix is the same as that between two n^2 matrices. It is further assumed that there is an operational dependency upon the individual matrix operations such that no more than one matrix operation may take place in parallel for a single filter. Thus, the total time taken to perform one iteration of a Kalman filter is the summation of the individual times for each operation given in Table 6.2.

The time taken to perform a number of Kalman filter operations on a linear system with four system states (n=4), and two measurements (m=2), is shown in Figure 6.17, for both the Transputer and the SIMD array within a Cluster. The time taken on the SIMD array of the WPM is shown for two data mappings, the one per PE mapping and the Sheet mapping, both are discussed in Section 5.5. The time for the calculation of the filter on the SIMD array is based on the matrix performance figures from

Section 5.5. The use of the EKF with its linearisation operation, or the use of the PDAF data association requiring a modified co-variance term, will add to the processing requirements shown in Figure 6.17.

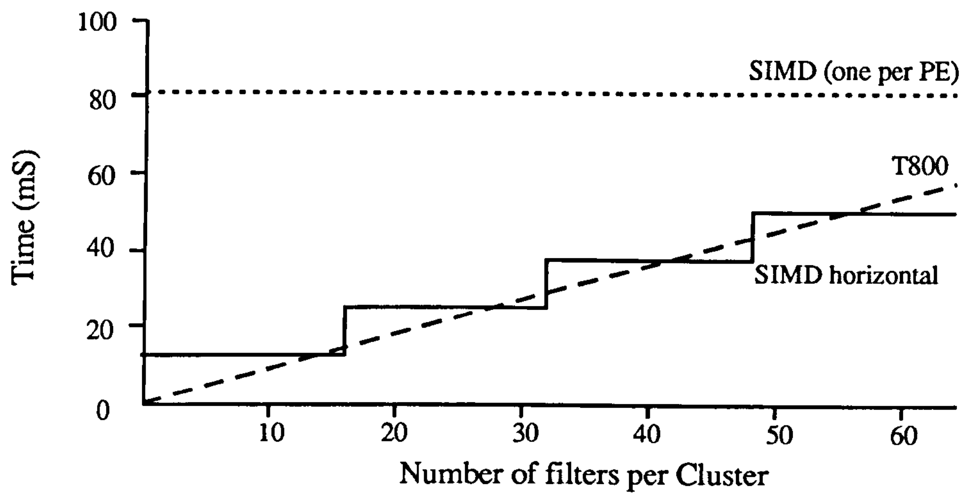


Figure 6.17 - Time taken to perform a number of Kalman filter operations, n=4, m=2, on a WPM Cluster and a T800 Transputer.

The time taken for the first 16 filter operations is a constant on the WPM, since the full utilisation of the SIMD array occurs when 16 filters, each using 16 PEs, are being calculated. The time taken to perform the same operation on the Transputer increases linearly with the number of filters to be calculated. It is interesting also to note that the time taken on either the WPM Cluster or the T800 Transputer is approximately the same over the numbers of filters considered. However, as the number of filters approaches full utilisation in the one PE per filter mode, the PE array achieves approximately twice the speed of the Transputer.

The two example applications, considered previously, had different requirements for the number of tracking filters. In the first example, only a couple of filters where required for each object being tracked. In the second, a single filter was required for each target under track. Typically, there are only a few objects in the entire image in the first example, but in the second, anything between 0 and 16 filters per Cluster can be typically required. Thus, for the first application it is clear that the filter calculations should be perfomed on the Transputers within the WPM. For the second application however, it is not clear where the filters should be processed.

It is quite probable that in a real tracking environment, some sort of manoeuvre tracking capability may be required, altering the tracking models from 1st order to a 2nd order, and vice-versa. An example of this can be seen in the tracking performed by Pfeiffer [Pfeiffer89] where different models are used dependent upon the distance of the targets from the imager. In such cases, the processing of the Kalman filter becomes a control parallel, rather than data parallel, problem with each filter requiring differing processing due to the different filter models. For instance, if two different models are required within N filters, the time taken on the SIMD array would double. For the Transputer it would remain the same if extra computation, due to the different dimensionalities of the tracking models, is ignored.

The processing required in the tracking applications is thus divided between the SIMD and MIMD levels of the WPM. The image processing and data validation operations are performed on the Multiple-SIMD array, and the data association and tracking filters are performed on the MIMD array. The operations on the two processor arrays are performed simultaneously. A filter produces predictions which are passed to the SIMD array for data validation. The results are then passed back to the Transputer. While the data validation is taking place on the SIMD array, the Transputer can calculate the predictions for the next target, after updating the previous targets filter with the previous data association results.

Operation	WPM level	Application 1 (μs)	Application 2 (μs)
Image processing, global	M-SIMD	109	3 to 1200
Image processing, local, per target	M-SIMD	322	11
Validation Gating per target	M-SIMD	512	496
data association per target	MIMD	0	31
Kalman filters per target	MIMD	912	1182

Table 6.3 - Timings for the image processing and tracking operations in the two tracking applications.

The times for the image processing and tracking operations are shown in Table 6.3. The M-SIMD array has a clock speed of 10MHz and the MIMD processor used was a

T800 Transputer with a clock speed of 25MHz. The data association and Kalman filter computations on the Transputer, occur concurrently with the data validation on the M-SIMD array. Note that the time given for the tracking operations is per target. When used in a system with a frame time of 40ms, the number of targets that can be tracked, within each Cluster, is 44 using the NNSF data association, or 33 using the PDAF.

6.6 Summary

The two tracking applications considered have illustrated the requirement for three different forms of processing, namely global data parallel, local data parallel and numeric. The global processing of the image data takes place in an application specific manor to produce image features which can then be tracked. A data association operation is undertaken on the processed image features and the predictions from targets under track, and requires local data parallel processing. The tracking filters for each target require precision numeric calculations on small data sets. Control parallel processing is well suited to this operation. Each of these processing types can be matched to the processing levels within the WPM, on the M-SIMD level operating in its 'synchronised' mode, on the M-SIMD level operating in its local autonomous mode, or on the MIMD level.

The extraction of image features typically leads to areas of the image requiring further processing. In the first example application, a local region of the image required further processing after the object detection, for segmentation and feature extraction. In the second example, image features consisted of points which where typically sparse and unevenly distributed across the image plane. Both examples leads to different intermediate data forms on the M-SIMD array of the WPM.

For the efficient processing, each of the intermediate data forms would ideally be evenly distributed across the whole M-SIMD array. Even distribution of the data across a processing resource is commonly referred to as load-balancing. This is the subject of the next chapter where two data forms are considered for load-balancing across the WPM, that of small regions of iconic data or sub-images and that of sparse data.

Chapter 7

Load-Balancing on the WPM

7.1 Introduction

The efficient utilisation of resources is of prime importance in real-time processing systems and even more so in vision systems which involve high bandwidths of data. In parallel processor systems there are in general six sources of overhead which can affect the speedup of a parallel application implementation [Singh92]. These are :-

- inherently serial sections (Amdahl's law [Amdahl67])
- redundant work
- overhead of managing the parallelism
- synchronisation overhead
- load imbalance
- communication.

All these factors affect the performance of applications to varying degrees. A parallel architecture can influence the degree to which these factors are important through suitable hardware support, although the first factor is solely a feature of the application. However, a general programmable architecture is not designed for any specific application and so each of the factors should be minimised as much as possible. In data-parallel applications, including image analysis, the main factors are the load imbalance and the communication overheads.

For example, load imbalance can occur when part of the image contains a clustering of relevant features. These features may require separate processing, as was the case with the object segmentation processing of Section 6.3, and the data association process of Section 6.5. In such situations, the processing time taken can be much greater than if the processing were partitioned such that each Cluster, in the WPM, is required to perform the same amount of processing, i.e. load balanced. The load imbalance can occur dynamically, i.e. at run-time, within the processing of a particular algorithm (e.g. the data association). It can also result at an algorithmic level, for example through the processing of different objects in an image using different algorithms.

The communication overheads may be kept to a minimum by the preservation of data locality within the parallel processor. The time taken to perform communication between data elements is assumed proportional to the distance between them in terms of processors. In the iconic sense this would require adjacent pixels to remain adjacent after the load-balancing, thus preserving spatial locality. Similarly, with sparse data local relationships need to be preserved, although in this case spatial locality is not necessarily needed.

An important factor in the design and use of a load balancing algorithm is the cost, or time, that the algorithm takes to be performed. The cost of the load balancing has to be outweighed by the benefit, or improvement in processing time, achievable. The load-balancing condition to be satisfied is

$$\max [\ T_j(\text{norm})\ ] - \max [\ T_j(\text{balanced})\ ] > T_{\text{load}}$$

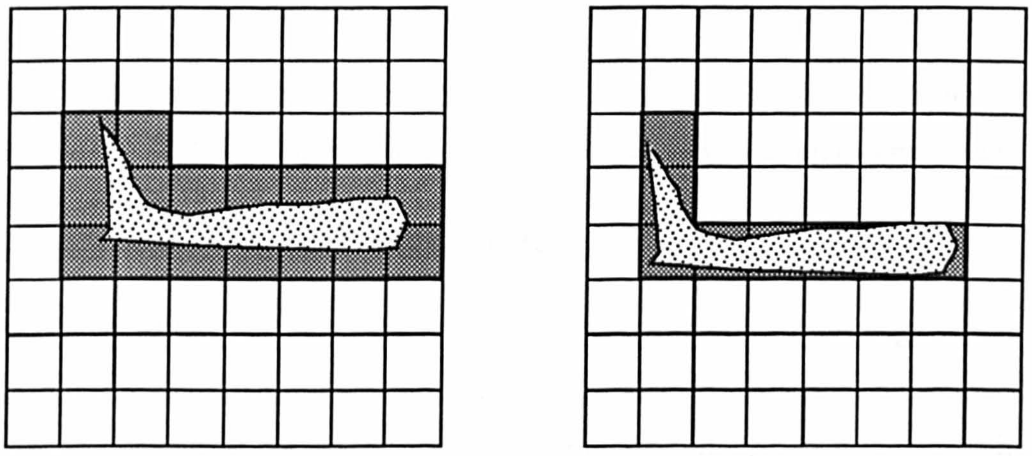
where T_j(balanced) is the time taken by the load-balanced processes on processor j, T_j(norm) is the time taken for the unbalanced processes on processor j, and T_{load} is the time taken for the load-balancing process.

Techniques for load-balancing on the M-SIMD array within the WPM are discussed in this chapter by the illustration of two different types of applications which require quite different load-balancing algorithms. The first is an iconic situation involving sub-images which are required to be tessellated across the whole processor array for efficient processing. The second deals with the rearrangement of sparse data which can occur in the target tracking domain as described in Section 6.5. Although the load-balancing operations are described in terms of the M-SIMD array within the WPM, they are equally applicable to the MIMD Transputer array.

The load-balancing of sub-images results in a two phase process, the first of which ensures that the minimum number of processors are used to process a particular shaped sub-image - this is described in Section 7.2. The second phase is to assign the sub-images to Clusters, within the processor array, minimising any unused areas. This conforms to the classical bin-packing problem with has a known complexity of Non-Polynomial (NP). The methodologies generally applied to this type of problem are described in Section 7.3 and the suitability for use on the WPM described. A method for the compression of sparse data, while preserving local data relationships, is examined in Section 7.4. This leads to the formulation of a modified algorithm for the greater preservation of locality between sparse data elements.

7.2 Minimising Clusters Used for Processing

For the efficient processing of sub-images on a fixed computation topology, such as the WPM, each sub-image area must be optimised such that the number of occupied Clusters for each and every one is kept to a minimum. An example sub-image of a synthetic aeroplane is shown in Figure 7.1. Each of the smaller squares represents the SIMD patch of a WPM Cluster (forming an 8x8 patch M-SIMD machine).



(a) Original area (b) Area after optimisation

Figure 7.1 - The area of an aeroplane on an M-SIMD machine.

The number of occupied SIMD patches, used for the aeroplane, is reduced to 8 in the optimised case against 16 in the original. The benefits of this optimisation are two fold: a greater number of idle SIMD patches can be used for the processing of other objects, and the communication overhead in the active areas is reduced.

The minimum number of SIMD patches used for processing such an object can be found by a naive method of masking the sub-image region, repeatedly shifting and performing a count of the number of occupied SIMD patches on each shift. The shift occupying the least number of patches would thus be chosen. However, a total of 16^2 shifts and counts are required - one for each possible position of the sub-image area. This is computationally expensive. A vote algorithm, with much lower computational overhead can be used and is described below.

The vote algorithm consists of two stages -

1) Each SIMD patch calculates a set of shifts, that can be performed on the sub-image area, to free itself of it. This is done for each of the directions of North, South, East, West, and each diagonal although only if such a shift exists.

2) The shifts calculated from step 1 are broadcast to the other SIMD patches allowing them to vote on the effects that each shift would produce on the image region. A count of the number of patches that would be freed, and the number of new patches that would be occupied, is performed. The difference between the two is the effective number of SIMD patches freed, the maximum represents the optimum mapping of the object across the M-SIMD array.

Most of the operations required, for the vote algorithm, use information from neighbouring SIMD patches thus keeping necessary communication to a minimum.

7.2.1 Calculation of Image Shifts

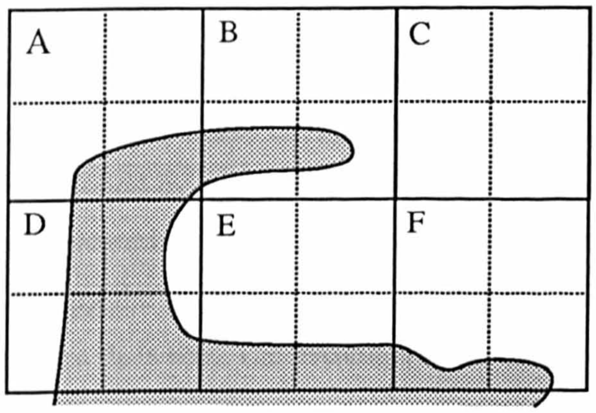
An SIMD patch may be either partially or fully covered with a sub-image, as can be seen in Figure 7.1. The patches which are only partially covered can be transformed to a free patch, available for the processing of another sub-image, by a suitable shift of the object in one of the eight directions. The calculation of the shift required to free an SIMD patch is as follows.

Each of the sub-image boundary PEs are labelled as being on the North, if the PE is on the northern boundary of the object and is in the bottom half of the SIMD patch, or

South if the PE is on the southern boundary of the object and is in the top half of the PE, and so on for east and west. Similarly, concave corner PEs are labelled with their respective corners if for instance a NE boundary PE is in the SW quadrant of the SIMD patch and there is no other occupied PE to the right and above it.

Consider the north edge PEs. The maximum of these, i.e. the top of the object within the bottom half of the SIMD patch, represents the required shift south to free the SIMD patch. If the image contains concavities, the South shift may result in part of the sub-image being shifted in from the north Cluster. A check is required on adjacent Clusters to see if any part of the sub-image will be shifted in. Similar operations are repeated for each of the remaining directions.

For example, part of a sub-image region is shown in Figure 7.2 mapped across six Clusters, A to F. The shift directions which free each Cluster from the image region are also shown. Note that, after the checking of neighbouring Clusters, the south shift from Cluster E would no longer be valid. This is because part of the sub-image from Cluster B would be shifted into E, and therefore would not be freed.



Shifts	Cluster
N	-
S	A,B,E,F
E	-
W	-
N&E	-
N&W	-
S&W	F
S&E	A

Figure 7.2 - An object across five Clusters showing the possible shift directions that can take place to free each Cluster of it.

The result of these calculations is a table of shifts, calculated within each Cluster, which will move the sub-image out of its SIMD boundary. The calculations of these shifts can be performed on each of the SIMD patches in parallel using their M-SIMD operational capability. Each of the shifts are then broadcast to the other SIMD patches covered by the object so that a voting process may be performed to determine whether or not the shift will free further Clusters.

7.2.2 Voting on an image shift

Consider the voting for one requested shift (S_x, S_y), from one of the list of calculated shifts within each of the other sub-image SIMD patches. S_x and S_y represent the requested shifts in the horizontal and vertical directions respectively. For a diagonal shift they are both non-zero. Each sub-image SIMD patch then votes on this, the result being either a *CLEAR*, *NO_CHANGE* or *OCCUPY*; meaning that the shift will either free up the patch, neither frees or occupies another patch, or will occupy another patch in addition to itself respectively.

The vote algorithm uses a notation of '0' and '1' to denote the state of an unoccupied and occupied SIMD patch respectively, and a subscript of '-' and '+' to denote the situation before and after the requested shift. Each SIMD patch is addressed as (X, Y) within the M-SIMD machine, with the origin being the top-left patch. The algorithm is illustrated below for a requested shift in the North-East direction but it can easily be changed for other directions.

For a North-East shift the patch (X,Y-1) could be occupied by part of the object shifted from the patches (X,Y), (X-1,Y) and (X-1,Y-1). The possible occupancy can easily be found by a masking operation and an associative response within each of these neighbouring patches. If a neighbouring patch contains part of the object which will occupy the patch (X, Y-1) after the shift, a 1_+ is sent to the patch (X, Y-1), otherwise a 0_+ is sent. The masking required is:-

For patch (X,Y)	If PE(a, b) is occupied then send a '1_+'
For patch (X-1,Y-1)	If PE(c, d) is occupied then send a '1_+'
For patch (X-1,Y)	If PE(c, b) is occupied then send a '1_+'

where $1 \le a \le (16-S_x)$, $(16-S_y) \le b \le 16$,
$(16-S_x) \le c \le 16$, $1 \le d \le (16-S_y)$

An example is shown in Figure 7.3 with part of a sub-image initially mapped across three SIMD patches. The shift shown is in a North-East direction with $S_y > S_x$.

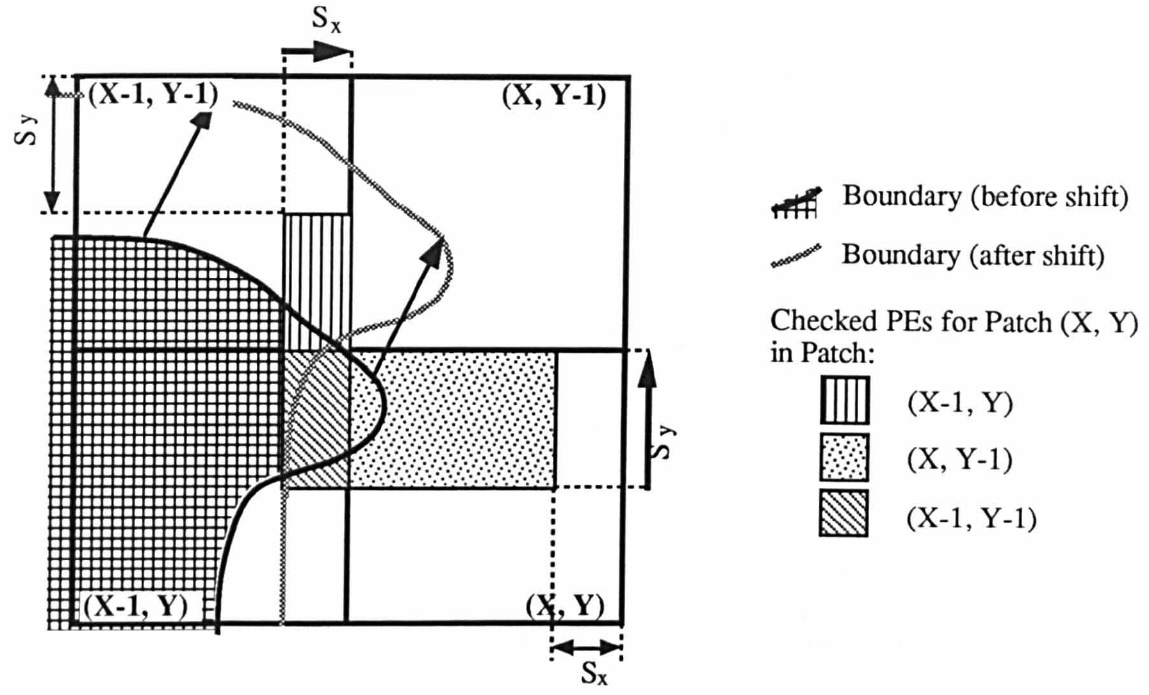


Figure 7.3 - Example of a North-East shift on part of a region mapped across four Clusters.

The voting of patch (X,Y-1) is dependent upon the values received from its neighbours and its own occupied status according to the following:

If the patch had status '0_-' and only '0_+' are received then vote *NO_CHANGE*
If the patch had status '1_-' and a '1_+' is received then vote *NO_CHANGE*
If the patch had status '0_-' and a '1_+' is received then vote *OCCUPY*
If the patch had status '1_-' and a '0_+' is received and at least one of the requested shifts (S_x, S_y) is contained within one of the calculated shifts for the patch then vote *CLEAR*

The votes are calculated for each SIMD patch in parallel using their M-SIMD capability, and then sent to a single Cluster, such as the upper-left Cluster containing the sub-image, for accumulation. The difference between the *CLEAR* and *OCCUPY* votes is the effective number of patches freed on that shift. The maximum of this, over all requested shifts, gives the shift for the optimum sub-image mapping which can then used to shift the sub-image.

In the example shown in Figure 7.3, three SIMD patches are initially occupied. After the shift, SIMD patch (X, Y) becomes freed but patch (X, Y-1) becomes occupied. These two patches vote *CLEAR* and *OCCUPY* respectively, as shown in Table 7.1. The shift results in an equal number of *CLEAR* and *OCCUPY* votes for the part of the sub-image shown, i.e. there is no change in the number of occupied patches.

	Occupied status		
Patch	Before	After	Vote
(X, Y-1)	0_-	1_+	*OCCUPY*
(X-1, Y-1)	1_-	1_+	*NO_CHANGE*
(X, Y)	1_-	0_+	*CLEAR*
(X-1, Y)	1_-	1_+	*NO_CHANGE*

Table 7.1 - Voting of the SIMD patches shown in Figure 7.3.

The main requirements of the vote algorithm is concerned with the broadcasting of the requested shifts and receiving the resultant votes. However, requested shifts only occur on the boundary SIMD patches, typically one or two per Cluster containing part of the sub-image boundary. Thus, the complexity of the voting algorithm is $O(\sqrt{N})$ where N is the number of SIMD patches occupied by the sub-image. It is estimated that the calculation of the requested shifts and the voting on a single shift is 160 cycles and 40 cycles respectively on the WPM.

The possible decrease in the number of Clusters required for any one sub-image can be a factor of 4. This is the case when a single 16x16 sub-image is mapped, initially across four Clusters, but after a suitable shift is mapped across only one. The actual decrease obtained is dependent upon the application. The decrease in the required number of Clusters is effectively the same as the speedup that can be obtained after the load-balancing operation.

7.3 The Bin Packing of Clusters

The problems associated with the scheduling of jobs in a parallel system is a variant of the classical bin-packing problem with known complexity of NP. Therefore, optimal solutions are expensive to calculate, if at all feasible. Approximations and heuristics have been employed for the solution of such problems over the past decade and have had performances approaching that of the optimum packing as the number of jobs increases [Dowsland92]. The scheduling of image patches across the M-SIMD array of the WPM conforms to the same problem.

Each of the jobs that arises within the WPM is a sub-image, assumed to be of size (p, q) horizontally and vertically and each with a processing time of t. Thus each job is represented by a three part identifier, $J_i = (p_i, q_i, t_i)$ where $i \in \{1 .. n\}$ and n is the number of such jobs. These sub-images can arise at any location across the WPM array, and if the input image is larger than the SIMD array of the WPM, or many images are mapped across the array at any one time, many sub-images can arise in the same Cluster. An example of the distribution of the sub-images is shown in Figure 7.4 for four images, or four tiles, mapped across an 8x8 Cluster WPM.

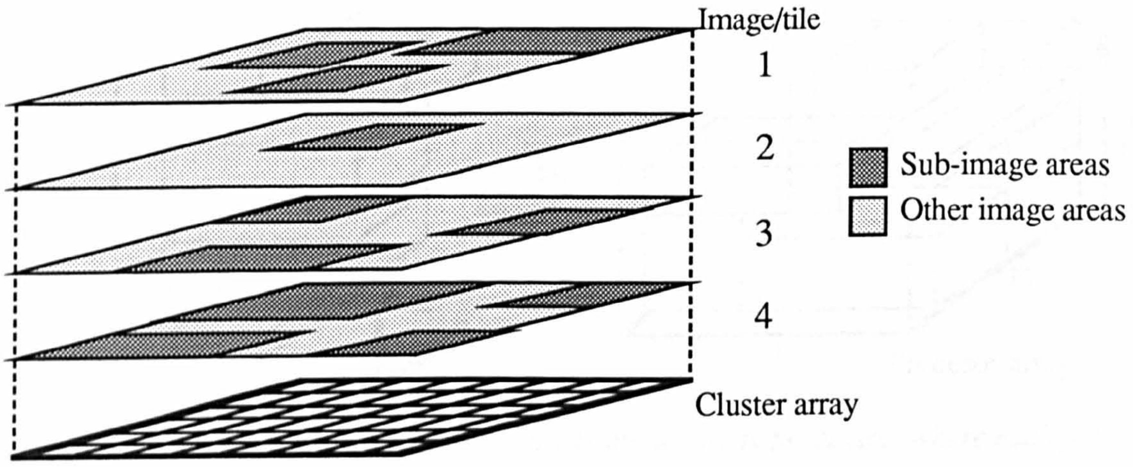


Figure 7.4 - Example distribution of sub-images across the WPM, from four images.

7.3.1 Three-dimensional bin packing

Examples of 3-dimensional packing includes those of container loading [Gehring90], where the goal is to maximise the load of the container, or conversely to minimise the volume wastage. The load-balancing problem here can be stated as the scheduling of Jobs J_i across the array such that the total processing time from start to finish is minimised. Generalising this into a packing problem, the jobs (p_i, q_i, t_i) can be treated as 3-dimensional boxes which are required to be packed into a three dimensional space representing the processor array dimensions and time [Li89]. The jobs are packed so as to minimise the height in the 3rd dimension, i.e. to minimise the processing time.

An example of 3-dimensional packing is shown in Figure 7.5 for a set of jobs mapped onto an array processor. The overall processing time is represented by the dotted line - the volume difference between this and the individual jobs is wasted processor time which, ideally, should amount to zero. Each processor in the array is the smallest functional block with can perform independent processing - in the case of the WPM the processors represent a Cluster, both a 16x16 SIMD array and an MIMD processor.

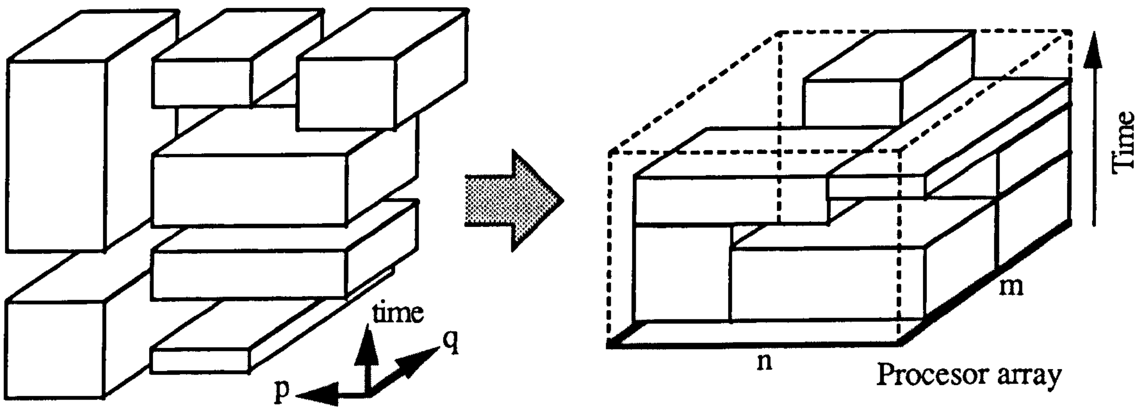


Figure 7.5 - Example of a set of jobs mapped onto an array processor, where each job requires (p_i, q_i) processors and t_i time.

The complexity of the packing problem is further increased when each of the jobs are considered as being malleable [Turek92]. For example, each sub-image of processor size (p_i, q_i) with time requirements, t_i, can be rearranged to fit into a single M-SIMD array of size 16x16 PEs using tiling operations. The resulting blocks would then be of size (16, 16) with an associated increase in processing time of $\frac{p_i}{16} * \frac{q_i}{16} * t_i$. Note that this is approximately correct if the differences in communication time between the two data mappings is ignored - see Section 5.3.

An example of this is shown in Figure 7.6 for a 4x4 Cluster processor array with a total of sixteen jobs. In Figure 7.6a each of the jobs have been arranged such that they are all the size of a single Cluster, but with the time required proportional to the number of image tiles contained within each Cluster. In Figure 7.6b each of the jobs are arranged such that they utilise the maximum number of PEs possible, i.e. a single pixel per PE. In this example, the sub-images range from a size of 2x2 Clusters to 4x4 Clusters, in steps of 2x2 Clusters.

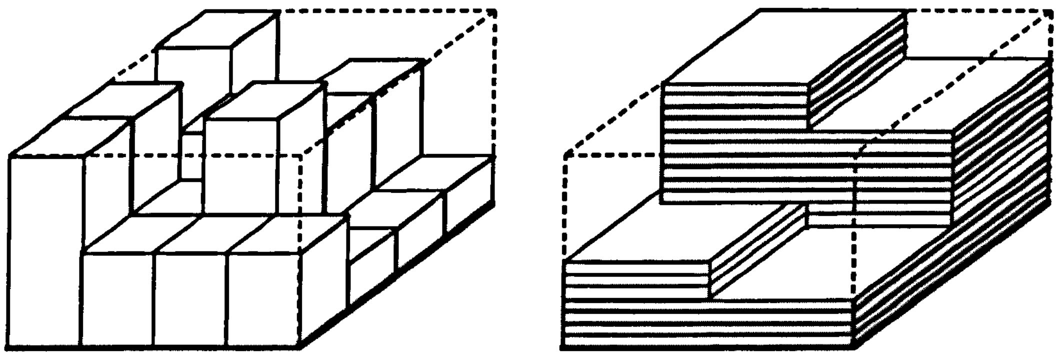

a) each mapped to a 16x16 SIMD patch b) each SIMD PE contains one pixel

Figure 7.6 - Example of the malleability of the mapping of sub-images.

The time required for both mappings shown is the same, but as can be seen, the mapping in Figure 7.6b, could be further load-balanced by placing the top few square tiles in the hole to the lower right. This would reduce the overall time taken in comparison to Figure 7.6a. However, the easiest data mapping that can be load-balanced occurs when the jobs are of the same size, such as is the case in Figure 7.6a. In this case each job is the size of a single Cluster, and tessellation is easy. But, it requires at least the same number of jobs as the number of Clusters, and ideally each would have the same time requirements.

A further factor, not considered above, is the possibility of time dependency between processes. One process may only be able to be performed after the completion of another. This factor further adds to the complexity of the packing problem and has not been considered here.

In a typical application, the third dimension of time cannot always be assumed to be known at the start of any job. Thus, the bin packing degrades from a 3-dimensional to a 2-dimensional problem. The dimensions now represent only the size of the sub-image, i.e. the number of PEs (p_i, q_i) required for each job and the malleability of the sub-image sizes is no longer a consideration.

7.3.2 Two-dimensional bin packing

The optimum solution of the packing problem can be found be using an exhaustive search procedure which tries each of the possible combinations of the sub-images, keeping the combination which minimises any wasted processors. This leads to an NP complexity. Since the number of sub-images within any one frame can enter double figures, a combinatorial solution can not be calculated within the required processing time of one frame period.

A formulation of the two dimensional packing problem has been described by Kroger [Kroger91] using a graph-theoretical model. The algorithm used is an iterative one and has been implemented on an array of 32 T800 Transputers. The time for each iteration of the algorithm, for the test cases considered by Kroger, ranged from 0.16s to 2.3s. The number of iterations required was typically 5000 - a worst time requirement of 11500s (3 hours). Such an algorithm cannot be applied within the time constraints of an image analysis system with a typical frame time of 40ms.

Approximations to the optimal solution are thus required. One common approximation is that of shelf-packing. Shelf packing considers the blocks to be placed in strips across the width of the area being packed. Each shelf is given a height equal to the height of the tallest block placed on that shelf. Subsequent shelves start above the first shelf. One shelf algorithm is the Next-Fit rule (NFL) [Baker83] in which the blocks are placed on the bottom shelf left-justified until the next block cannot fit on it. A new shelf is then begun and the process continues for this new shelf.

An example of the NFL shelf algorithm is shown in Figure 7.7a. Hofri [Hofri80] showed that the efficiency of this algorithm approaches 66% of the optimum packing as the number of blocks -> ∞. If the blocks are initially sorted into height order, an algorithm such as the First Fit Decreasing Height (FFDH) [Coffman80] can be used. The efficiency of this algorithm approaches 100% of the optimum packing as the number of blocks -> ∞ [Coffman90]. An example of the FFDH is shown in Figure 7.7b with the same blocks as those in Figure 7.7a. The FFDH algorithm is more efficient than the NFS algorithm in this and most other cases.

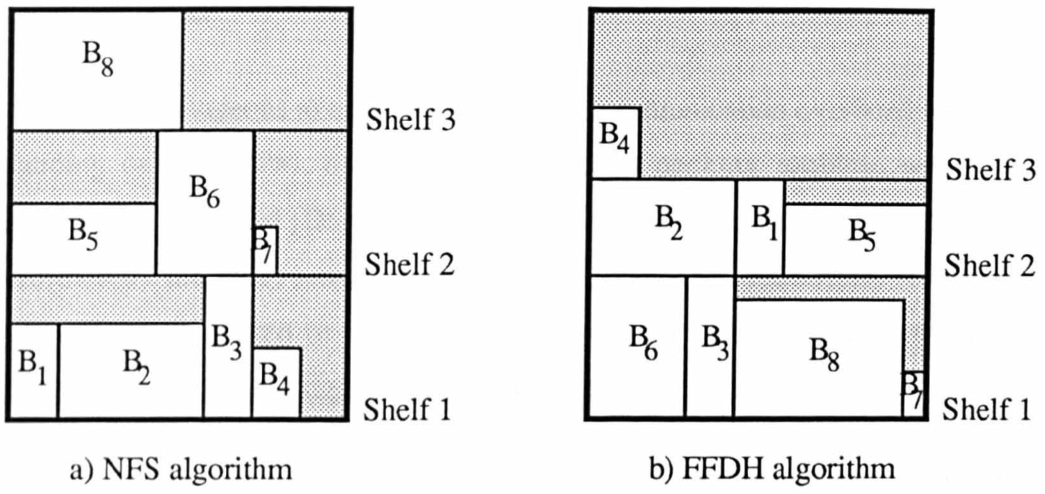


Figure 7.7 - Example of the shelf packing of eight blocks.

The shelf algorithms shown above can be directly applied to the mapping of the two dimensional sub-images onto the M-SIMD array within the WPM. The shape of the sub-images need not be rectangular. However, if they are 'padded out' to become rectangular, the efficiencies of the shelf algorithms will be reduced since the padded areas would lead to inefficiencies. The number and size of sub-images, to be mapped across the WPM array, can be such that they cannot all be fitted onto the array at once. In such cases, the initial mapping should minimise the wasted processor space so as to process as many sub-images as possible. When a job finishes, a remaining job of a suitable size can then take its place.

The movement of any data around a processor array is costly in terms of communication time. For the WPM, the communication times between Clusters was discussed in Section 5.2 where it was shown that by using the Clusters M-SIMD array internal wrap-around facility, the communication for large distances is greatly reduced over that of a conventional SIMD array. The cost is approximately 24 cycles for each Cluster shifted across, and 272 cycles for the final Cluster. Thus, the cost in moving a sub-image with top left corner in Cluster (x, y) to a Cluster (x', y') is :-

$$((x'-x) + (y'-y))*24 + 544 \text{ cycles}$$

The maximum distance any sub-image would move is the distance from one corner of the Cluster array to the centre, when using the torus structure of the WPM as a whole. This is a distance of (P+Q)/2 Clusters, where (P, Q) are the number of Clusters forming the WPM.

Thus, it is advantageous to minimise the image communication overhead whilst load-balancing on the WPM. Such a constraint on a packing problem was recently described by Dowsland [Dowsland90] for a robot arm. The objective of the packing performed by the robot arm, was to make as few moves as possible. However, the application was limited to only four different sized blocks, which were initially assumed to be physically separated from the area to be packed.

A novel heuristic algorithm has been developed here for the packing of sub-images which are initially within the processor array to be packed. The algorithm uses an ordered list of the sub-images, similar to the FFDH shelf algorithm above, but ordered in terms of area rather than height. This is termed the Fit in Place Decreasing Size algorithm (FPDS) below. The algorithm is as follows.

```
Order the sub_images so that the area of sub_image(i) >= sub_image(i+1)
FOR (i = 0, i < number of sub_images) do
    IF the processors that contain sub_image(i) are free THEN
        assign sub_image(i) to these processors
    ELSE_IF there is a set of processors that can contain sub_image(i) THEN
        assign sub_image(i) to these processors
```

The algorithm will assign each sub-image to the processors over which it lies, if they are free. Otherwise, the sub-image will be assigned to a spare set of processors, closest to the current position of the sub-image, which could accommodate it. This first spare set is found by spiralling out from the position of the top-left corner of the sub-image. Shifting of the image data, in the FPDS, is then only performed if necessary.

7.3.3 Comparison between the FFDH and FPDS algorithms

Both the FFDH and the FPDS algorithms have been used on a set of test data which was generated using a pseudo-random generator. It was assumed, for the simulation, that there were up to eight separate images mapped across a WPM processor array of 8x8 Clusters, similar to that shown in Figure 7.4. This mapping is effectively a subset of the mapping that could arise when an image is mapped onto a smaller processor array, although in this case the image is eight times larger. In addition, it was assumed that the sub-images were rectangular, with each side having a uniform distribution between 16 and 48 pixels in steps of 16 pixels. The comparisons shown use averages from a Monte-Carlo trial of 100 runs.

The number of Clusters utilised, within the WPM array after the use of both the FFDH and the FPDS packing algorithms, is shown in Figure 7.8. The average size of a sub-image is 2x2 Clusters (32x32 pixels) which results in the possible full utilisation of the array occurring when the number of sub-images is larger or equal to 16. When full utilisation is not possible, the number of utilised Clusters is linear. The utilisation of the WPM array when no load-balancing takes place is also shown in Figure 7.8. In this case, sub-images are only assigned to a set of Clusters if there is no shifting involved, and the Clusters have not already been assigned.

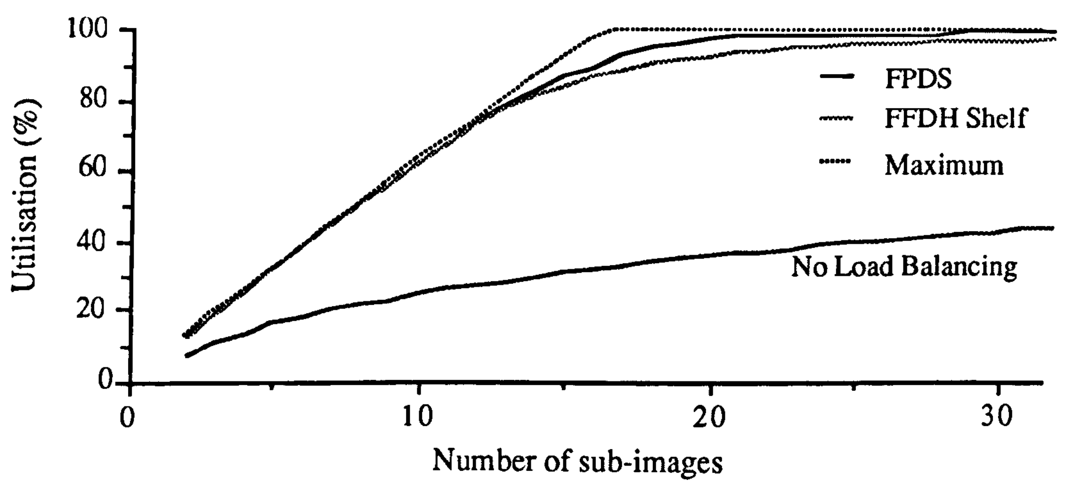


Figure 7.8 - The utilisation of the WPM using the FFDH shelf and FPDS load-balancing algorithms.

The resulting utilisation of the FPDS algorithm is very similar to that of the FFDH algorithm over the range of sub-images considered, out performing it by 5% in places. The utilisation without any load-balancing taking place is typically 50% of that with either of the load-balancing algorithms. Similarly, the number of sub-images assigned across the array, as shown in Figure 7.9, is similar for the two algorithms. However, the actual sub-images assigned, in both algorithms, are not necessarily the same ones for a given trial.

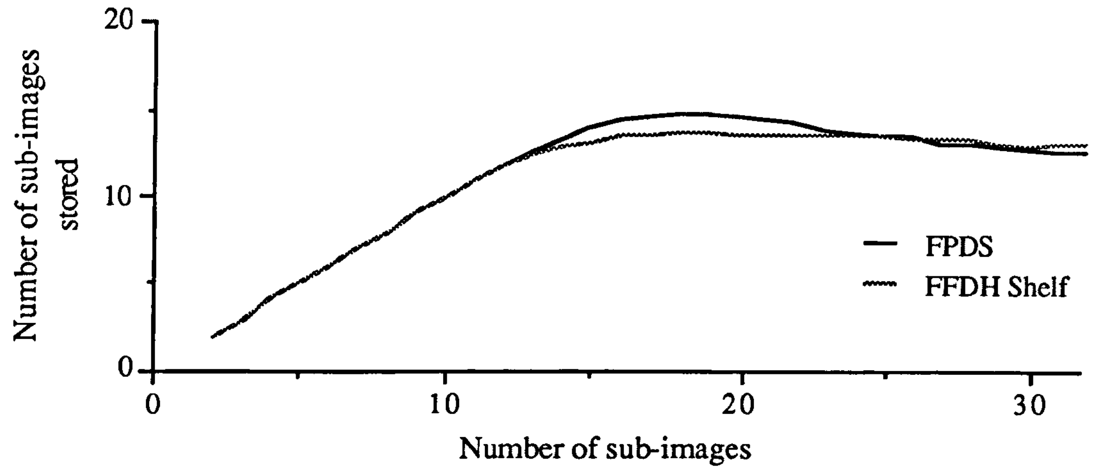


Figure 7.9 - The number of sub-images assigned across the WPM using the FFDH shelf and FPDS algorithms.

The FFDH and the FPDS algorithms have been shown to result in similar utilisation of the WPM above. However, the amount of data communications required for each algorithm is of prime importance. The communication requirements are shown in Figure 7.10. It can be seen that the FFDH algorithm typically requires twice as many data communications as the FPDS algorithm. This is as expected, since the FPDS algorithm was designed to reduce the amount of data communications required.

Two types of communications can take place: a type 2 shift is the last shift over any Cluster, required for communication in any one direction, and a type 1 shift is the preceding shifts in the same direction. These are separated due to the differing costs associated with each. A type 1 shift costs 24 cycles, and a type 2 shift costs 272 cycles (see Section 5.2). Note that the necessary synchronisation time between Clusters has been ignored. This would inflate the overall costs for any inter-Cluster shifting.

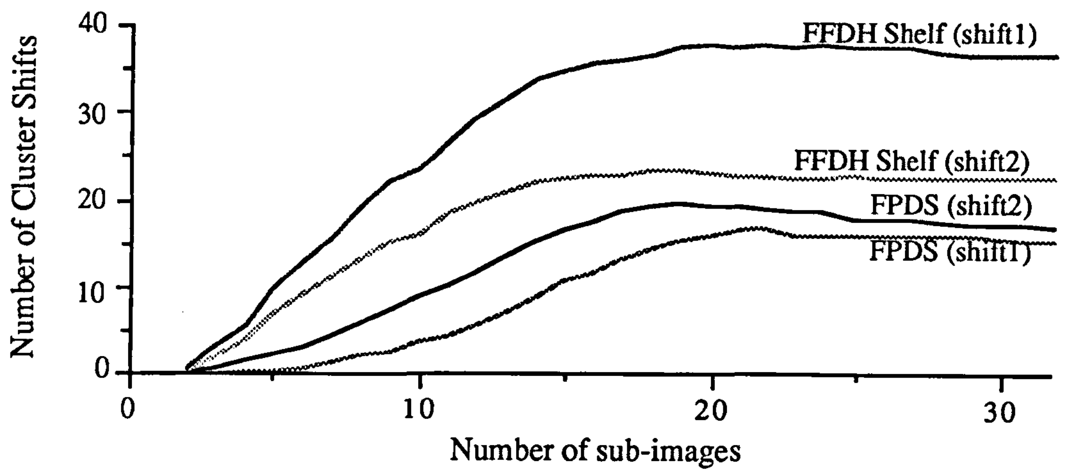


Figure 7.10 - The communications required using the FFDH shelf and FPDS algorithms.

The cost of the data communications, for both algorithms, is shown in Figure 7.11. The curves in Figure 7.11 are simply the summation of those in Figure 7.10 for each algorithm, weighted by the cycle counts required for each type of shift. It can be seen that the FPDS algorithm can be performed in less time than the FFDH algorithm. This reduction in time required for the data communications, resulting from the load-balancing operation means that increased time is available for the sub-image processing. However, the cost of the load balancing algorithm itself must also be added to this overall figure. This cost is analysed in the next section.

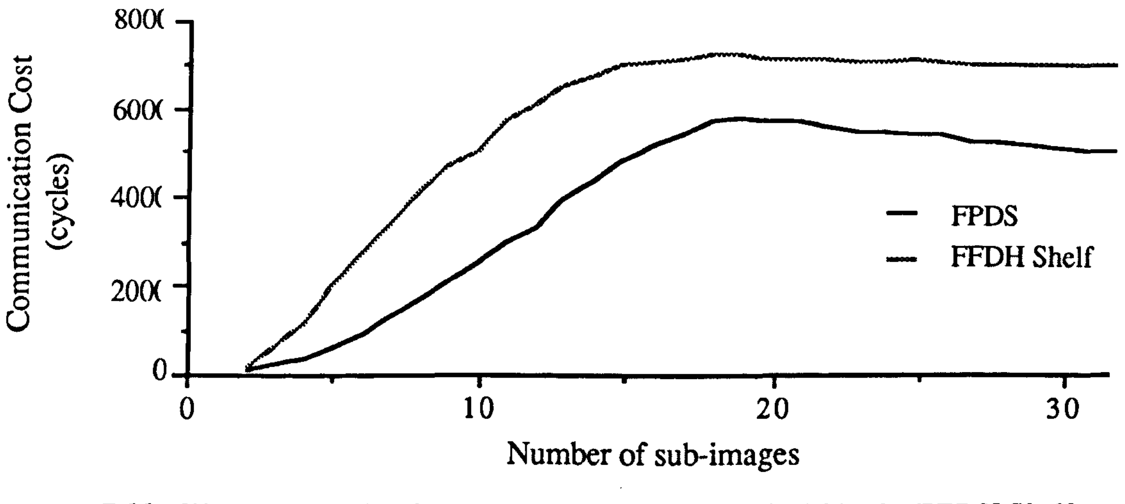


Figure 7.11 - The time cost for the communications required within the FFDH Shelf and FPDS algorithms.

7.3.4 Implementation of the load-balancing algorithms

Both the FFDH and the FPDS algorithms require the sub-image parameters of position (x_i,y_i), and width, height (w_i,h_i) in Clusters, i.e. pixels/16, to be communicated to a single Cluster in which the algorithm is to be performed. For simplicity, this is assumed to be the top-left Cluster. Similarly, the Cluster communicating each of the sub-image parameters will be the top-left one containing that sub-image.

Each algorithm requires a sorted ordering of these parameters. The FFDH requires the list to be sorted into height order, and the FPDS requires a size, (width*height), ordering. Such an ordering can be obtained by the use of a bin-sort, in O(n) time. The number of bins is known and is small. For the FFDH, the number of bins is the height of the input image, and in the FPDS the number of bins is the size that a sub-image can take. The height of a sub-image has to be less than or equal to the height of the input image and similarly for the size. For a 128x128 image, (8x8 Clusters), a total of 8 bins are required for the FFDH and 64 for the FPDS.

Once all the parameters have been received from each sub-image, and the bin-sort performed, the largest sub-image for either algorithm can be found as the first entry in the bins. An assignment message is sent to the Clusters that contain this largest sub-image, either in-place for the FPDS algorithm or in the bottom left corner for the FFDH algorithm. If the sub-image is not already in the place specified by this message then a shifting process begins as described earlier. While the shifting takes place, the next assignment can be calculated. Hence, for the FFDH algorithm, the latency time, to perform the algorithm is simply that taken to find the tallest sub-image. The remaining assignments can then be overlapped with the required shifting time.

The situation is somewhat different for the FPDS algorithm due to the first assignment requiring no shifting. In fact, the first sub-image required to be shifted is that which overlaps a Cluster in which an assignment has already taken place. Subsequent iterations of the algorithm can be overlapped with this shifting.

The time taken to perform both the load-balancing algorithms is shown in Table 7.2 for a total of 16 sub-images. The time taken on a single T800 Transputer, and a

Transputer with 16x16 SIMD array, is shown. It can be seen that both algorithms take less using both the Transputer and SIMD array. This is due to the bit operations, in the form of masks, used by both algorithms. However, the FFDH algorithm may be performed on a single Transputer, within the time taken for the resulting communications on the M-SIMD array (Figure 7.11). The FPDS cannot, it requires use of both processors if little overhead is required when compared to the M-SIMD communication time. The main overhead of the FPDS algorithm occurs on a clash, when the Clusters for a region have already been assigned to a previous sub-image.

	Transputer T800 (25MHz)		Transputer T800 + SIMD array	
Algorithm	1 iteration (μs)	Total (μs)	1 iteration (μs)	Total (μs)
FFDH shelf	40	640	2.5	40
FPDS heuristic	45 (if no clash) 160 (if clash)	2414	2 (if no clash) 15 (if clash)	110

Table 7.2 - Time taken for the load-balancing algorithms.

The times in Table 7.2 do not include the communication times for transmission of the required sub-image parameters across the array prior to the bin-sorting, nor does it include the transmission times of the assignment messages. Such timings can only be obtained when the algorithms have been implemented on a full sized working prototype of the WPM.

7.4 Load-Balancing of Sparse Data

For applications which contain sparse data sets during their computation, a different kind of load-balancing can be applied. Such situations arise in the target tracking domain, as described in Section 6.5 and in other many-body computations. In tracking applications, the data can remain 'in place' across the processor array and thus preserving the spatial arrangement between data elements. The position of the data elements is specified by the position of the PE containing it. However, unless the distribution of the data is bordering upon that which occurs in image processing, i.e. dense data, there will be a low utilisation of the processor array. For example, if one in ten PEs contain a data element in the sparse set then the maximum utilisation of the processor array that can occur is 10%.

The utilisation of the processor array can be increased however, by re-arranging the data or effectively compressing the area in which they are located across the processor array. This compression may result in only part of the processor array needing to be used on that part of the application, the rest freed for other processing. Alternatively, it may be the case that it is no longer necessary to tile the data set across the array, enabling the compressed data set to be fully processed.

When re-arranging the sparse data it may be required to preserve data-locality and inter-relationships. For example, this is required in the problem of finding near-neighbours of each data point. If the data remains in place, on the processor array, the search space required simply spirals out from the data point in a similar way to a convolution operation. A maximum distance is typically given which defines the size, S, of the search space, thus giving the algorithm a complexity O(S), i.e. independent of the number of data elements in the set. If, however, the data has lost all spatial locality information, all points must be checked with all other points resulting in a complexity of $O(N^2)$, where N is the number of data elements.

A data re-arrangement algorithm is required which effectively compresses the data set while preserving the locality between data points. An algorithm designed for this purpose is the Monotonic Lagrangian Grid (MLG) [Boris86]. An investigation into this algorithm has lead to a new partitioned version described in the next section.

7.4.1 The Monotonic Lagrangian Grid

The Monotonic Lagrangian Grid was designed for the compression of a sparse data set while preserving local relationships between data elements. It was originally called the Monotonic Logical Grid and later renamed the Monotonic Lagrangian Grid (MLG) [Picone90]. The scenario assumed, for the application of this data mapping, was that of a dynamical system, in which the data points within the data set could move to slightly different locations over one time frame. The method has been applied to the target tracking domain, within a simulation environment, using the BEAST (Battle Engagement Area Simulator/Tracker), on a Connection Machine, CM2 [Kolbe90].

The mapping of the data set is aimed at ensuring that the elements which are adjacent before the mapping are close within the MLG. The indexing in the MLG for two dimensions requires two conditions :-

$$x(i, j) \leq x(i + 1, j) \qquad 1 \leq i \leq n_x - 1$$
$$y(i, j) \leq y(i, j + 1) \qquad 1 \leq j \leq n_y - 1$$

where x(i, j) and y(i, j) denotes the x and y coordinates of the data points in the MLG location (i, j), and n_x, n_y is the size of the MLG in the x and y dimensions respectively. The latter can be equal to the square root of the total number of data points. The data stored in each location within the MLG is the (x, y) position of the data point in the original data space, in addition to any other parameters associated with each data point. The construction of the MLG can be extended to further dimensions, depending upon the dimensionality of the data being considered.

An MLG of N^2 data elements can be constructed in the following steps :

a) order the data in terms of their y-coordinate from the lowest to the highest
b) partition this ordering into N equal sets, indexing the first with j = 1, the second with j = 2 etc.
c) order the data within each partition in terms of their x-coordinate, indexing the first with i = 1, the second with i = 2 etc.

An example MLG ordering, on a set of 16 randomly distributed data points, is shown in Figure 7.12. The original space in which the data existed is shown on the left, the relationships that the MLG algorithm produces is shown in the middle, and the resulting 4-connected MLG is shown on the right. Note that the resulting neighbours in the MLG are the same as those in the original data.

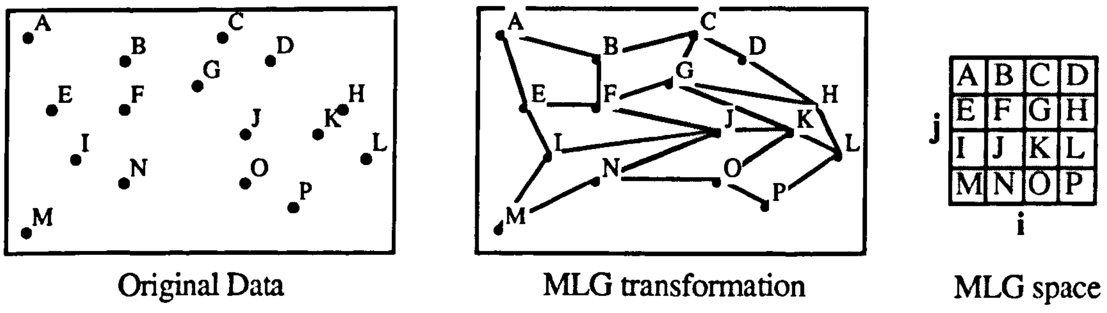
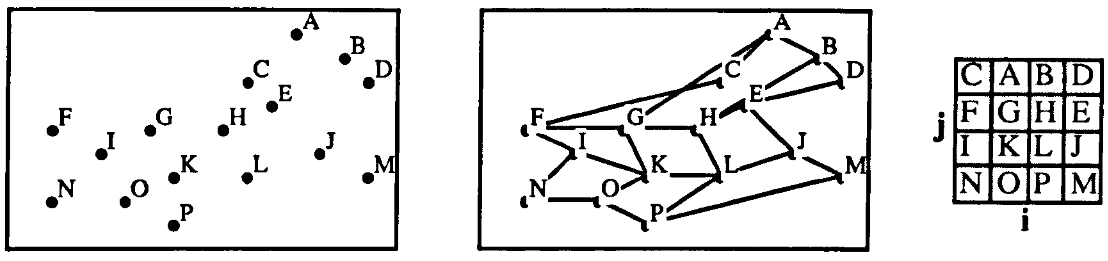


Figure 7.12 - Example of the MLG mapping on a set of 16 data elements.

The example shown in Figure 7.12 considers data which is ideally distributed for the MLG algorithm, the resultant MLG contains the relationships between the data elements that were required. A further example for a distribution of 16 data points is shown in Figure 7.13. This time however, the resulting MLG mapping does not preserve all local relationships. For example, the data elements C and E are adjacent in the original data but are at opposite ends in the MLG.

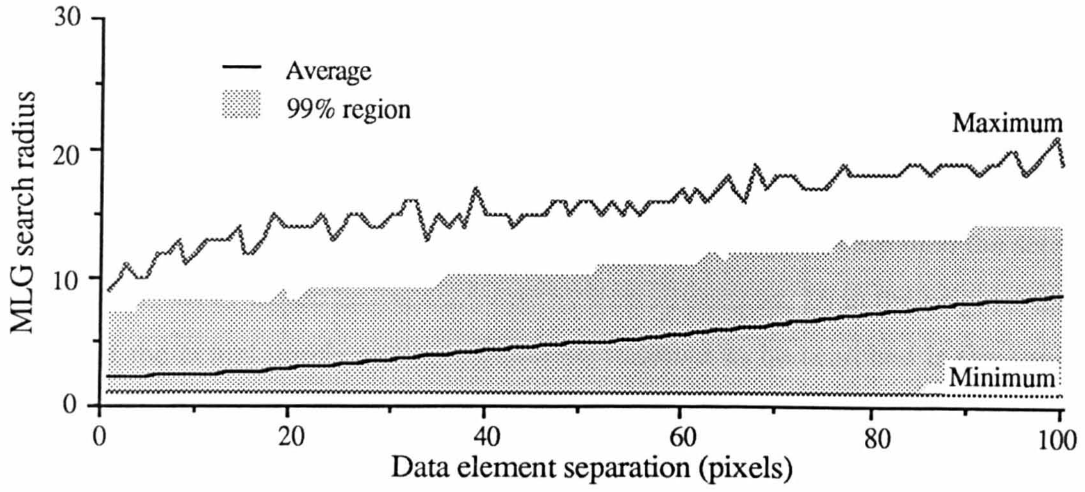


Original Data MLG transformation MLG space

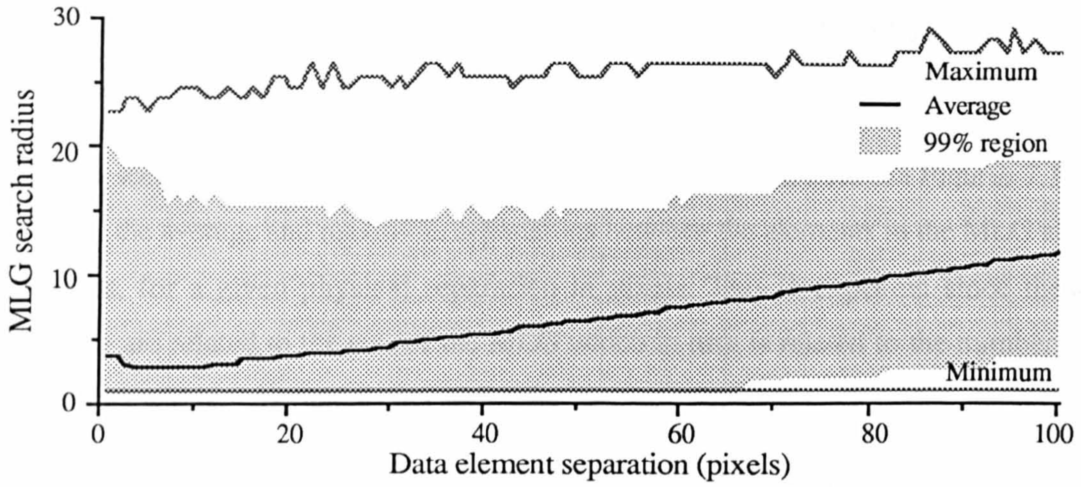
Figure 7.13 - Further example of the MLG mapping resulting in poor locality between data elements.

One of the worst data sets that could be encountered for the MLG mapping is that of N^2 data points along a diagonal line from one corner of the original data space to the other [Patterson91]. The first N elements would be placed in the first line of the MLG, the second N in the second line etc. A dis-continuity would occur between the last element in each row and the first element of the next row within the MLG. Thus, the maximum separation between any two data elements, e.g. the n^{th} and $n^{th}+1$ elements, would be a distance of N apart in the MLG.

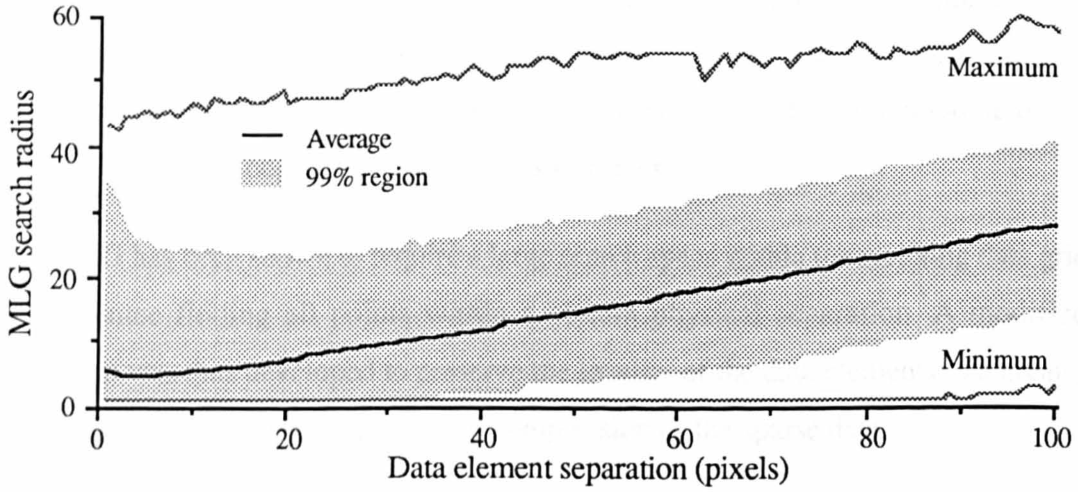
The MLG mapping was used on three different densities of data points for images of size 256x256. These images were generated from pseudo-random number generators giving an approximately even distribution of data points across the image. The number of data points placed within the images, for three test cases, was 500, 1000 and 5000 respectively. The distribution of distances between any two points on the image, compared to the distances between any two points within the MLG, is shown in Figure 7.14. These comparisons were drawn from a Monte-Carlo run of twenty images for each of the three densities.



(a) 500 data points on a 256x256 image



(b) 1000 data points on a 256x256 image



(c) 5000 data points on a 256x256 image

Figure 7.14 - Original vs. MLG separation on randomly distributed data.

The maximum and minimum curves on the graphs denote the bound on the search radius within the MLG, required in order to guarantee finding all data points for a given separation. The shaded region shows the search radius needed to find 99% of the data points for a given separation. For example, if all points within a physical distance of 16 pixels where required to be found in the 500 point case then the search radius in the MLG would be 14, (or 9 for the 99% region) vs. 16 in the original data. Note that the actual number of data elements required to be searched is the square around each data point. If s is the search radius, then $(2s+1)^2-1$ elements need to be searched up to a maximum of the number of data elements within the data grids. It can be seen that the average MLG separation increases linearly with the physical separation. The upper bound however, tends towards the size of the MLG data array, i.e. the square root of the total number of data elements, i.e. 23, 32 and 71 for the three cases respectively.

It can be seen that as the density of the data elements increases, between Figures 7.14a to 7.14c, the average and maximum separations between the elements in the MLG also increases, for a given physical separation in pixels. This is expected since these distances are related to the size of the MLG which in turn is related to the number of data points within the original data set.

The results indicate that a large area of the MLG must be searched if all data elements, separated by a certain distance, are required. The 99% region shown may be used to limit the search area when only approximate results are required. The region denoted by the minimum, maximum bounds and the 99% region are statistics for the evenly distributed data that was used here. For other distributions of data, similar statistics can be obtained as long as the data is available off-line.

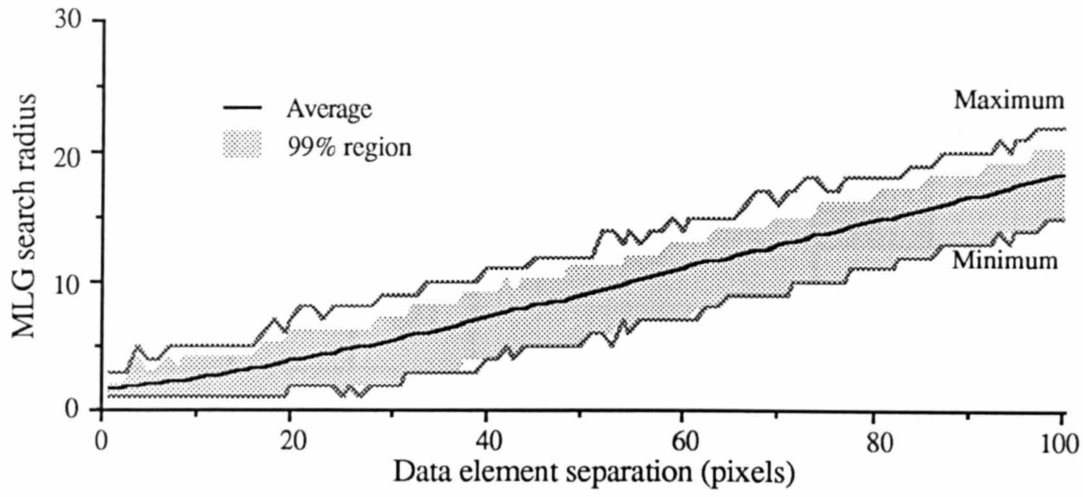
The MLG has been shown to require a large search space within the resulting data grid to guarantee finding all points within a certain physical separation. A modified algorithm was thus developed to preserve the locality of the data elements to a greater extent, while also performing a similar compression of the sparse data.

7.4.2 The Partitioned Monotonic Lagrangian Grid

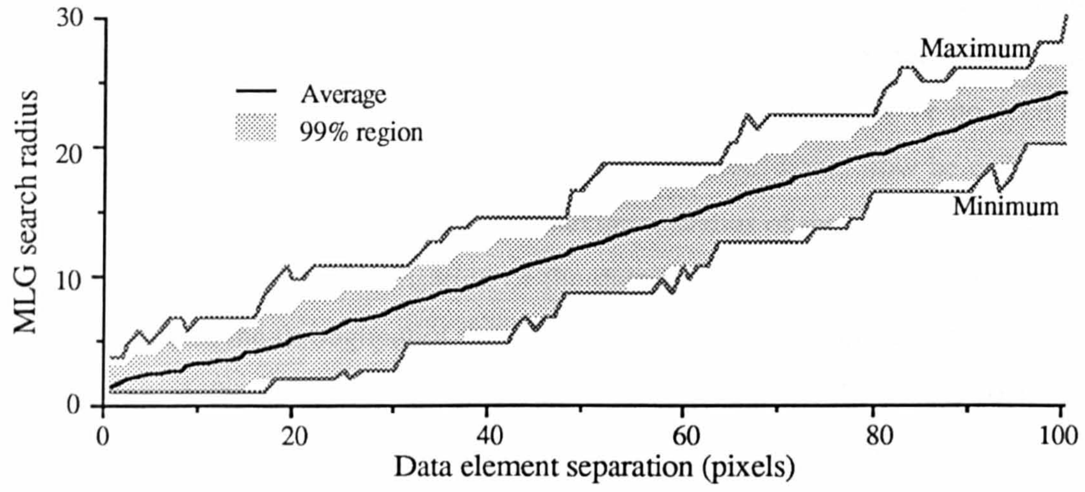
The solution used was to partition the original data space into a set of square regions, perform the MLG algorithm on each region and then join the resulting MLG blocks together to form the whole MLG. This algorithm is termed the Partitioned Monotonic Lagrangian Grid (PMLG). The size of the PMLG blocks, BxB, is chosen such that the distance between any two data elements separated by R in the original data space will be separated by a maximum distance of (2B-1) in the resulting PMLG. This is achieved by making the block size equal to the maximum of the number of data elements contained in any one of the RxR regions in the original data space. A further advantage of this partitioning is to localise the computation required for the formation of the PMLG. By making the boundaries of each of the partitioned regions to be the boundaries of the Clusters within the WPM, each block may be formed within each Cluster in parallel, thus making the value of R to be 16.

The PMLG was constructed using the same data sets as for the MLG above. The resulting separation between data points, compared with their image separations, is shown in Figure 7.15 for the three different densities of data points. It can be seen that the separation between data points, within the PMLG, are now within tighter bounds compared to the MLG separations shown in Figure 7.14. In fact, the maximum and minimum separations are defined by the block sizes. For instance all the data elements separated by a distance of R in the original data set (16 in this case) are now contained within the same or neighbouring blocks. Similarly if the separation required is between 16 and 32 only every second neighbouring block need be searched.

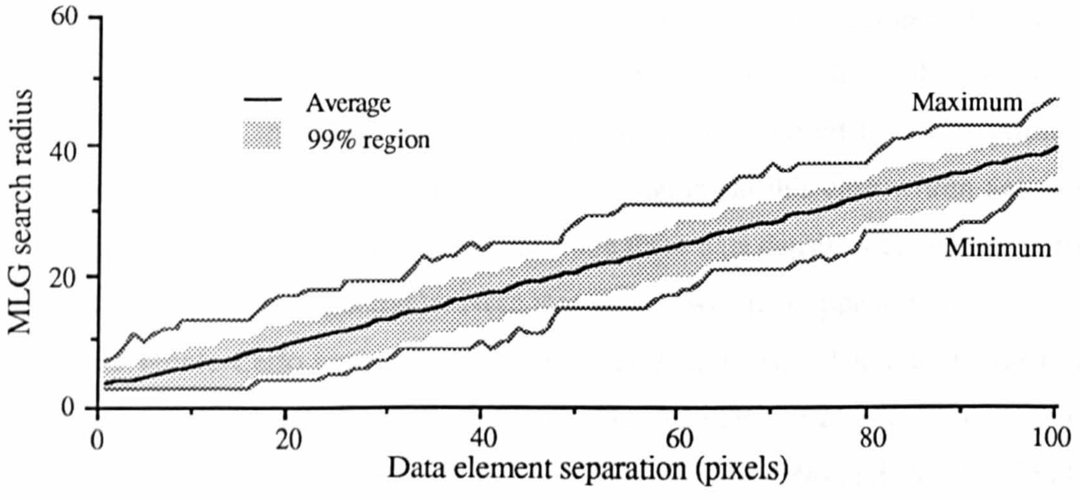
A further impact of this approach is that the PMLG blocks from neighbouring Clusters can be copied between Clusters, as shown in Figure 7.16. Such a capability is advantageous when optimisation of the number of processors is not important. The copying of data between Clusters allows the same data to be used within different operations on each of these Clusters. This process is very easily performed, only requiring shifting when the size of each PMLG block is 25 or less. In this case, all nine blocks can be placed within the 256 element SIMD array within a Cluster.



(a) 500 data points on a 256x256 image



(b) 1000 data points on a 256x256 image



(c) 5000 data points on a 256x256 image

Figure 7.15 - Original vs. PMLG separation on randomly distributed data.

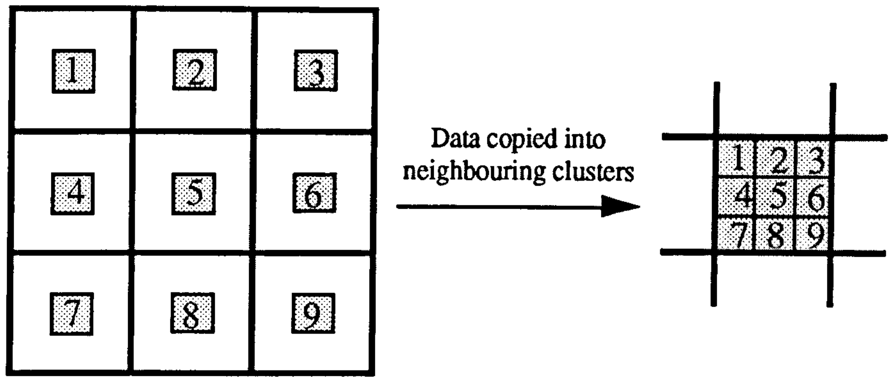


Figure 7.16 - Mapping nine compressed data sets to a single Cluster.

In the target tracking application considered in Section 6.5, the data association took place with measurements lying within the validation gate. The validation gate can lie within a Cluster, or across Clusters, depending upon the centre of the gate and its size. By copying the data from all neighbouring Clusters, there would be no need for the synchronisation of four Clusters, all the data required would now be contained within a single Cluster (assuming that the maximum radius of the validation region is less than 16 pixels). The localisation of the data association operation to a single Cluster increases the throughput of these operations that can be performed across the WPM array. A factor of four increase can be obtained, when the data association operations required four Clusters before the PMLG mapping and only one after.

7.4.3 Comparison between the MLG and the PMLG

The disadvantage of the PMLG compared to the MLG is that the compression of the data points is reduced. The data remains sparse, to some extent, due to the size of the PMLG blocks and the fact that the overall PMLG is specified by the maximum number of data elements within any one block region (in this case within any one Cluster). The PMLG block sizes required for the three data densities considered were (9,16), 16 and (36,49) respectively. Note that where two sizes appear, both occurred within the Monte-Carlo testing for that particular data density. The total number of such blocks for a 256x256 image in all cases was $(256/R)^2 = 256$ for $R = 16$, thus making the total size of the PMLG to be : (2304, 4096), 4096 and (9216, 12544) respectively. In comparison, the size of the MLG remains the same as the number of data elements, making a square data array, to give sizes of 529, 1024 and 5041 for the three data densities.

Since the size of the PMLG is governed by the peak number of data points occurring within any one Cluster, when uneven distribution of the data occurs the size of the resulting PMLG will be larger than that for the evenly distributed data. A set of simulated images where generated with an uneven distribution of data. Two parts of the image had a density three times greater than the rest of the image. An example image is shown in Figure 7.17.

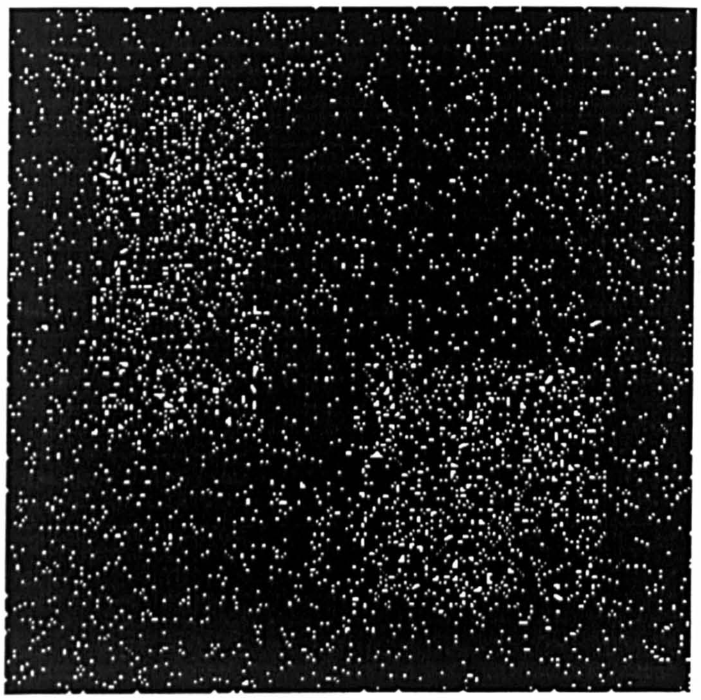


Figure 7.17 - Example of unevenly distributed data, the lighter regions are three times as dense as the darker regions.

The MLG and PMLG were used to form the compressed data arrays on data distributed as shown in Figure 7.17, with 1000 and 5000 data points. The distance distributions were obtained as before, for the evenly distributed data above, from a Monte-Carlo run of ten images. These are not included here since they were found to be very similar to the previous results for the evenly distributed data of Figures 7.14 and Figure 7.15. The size of the PMLG increased, due to the block size increasing to 25 and 64 for the two data densities respectively.

A comparison can be made between the MLG and the PMLG in one of two ways. The first compares the search spaces required to find the all data elements within a certain image distance, e.g. 16 pixels. The time taken to perform such an algorithm is proportional to this search space. The second compares the effective processor operations required to find neighbours within the same image distance. The search performed is across a total of $(2s+1)^2-1$ data elements, where s is the search radius from Figures 7.14 and 7.15 for the MLG and PMLG respectively. The number of processors needed for this search is assumed to be the same as the size of the MLG grids. Thus the effective number of processor operations is given by :-

(search area * MLG grid size).

The MLG required that all the elements within its grid need to be searched for a search radius of 16, resulting in $(\text{MLG grid size})^2$ processor operations required. The PMLG required the search of adjacent blocks only - a search radius of $s = (2B-1)$. The size of the PMLG grid was $256B^2$ hence the total number of processor operations required is:

$$((4B - 1)^2 - 1)*256B^2 = 2048B^3*(2B - 1)$$

A comparison of the size of the PMLG and the MLG is given in Table 7.3 for the three densities of points. Also given is the search space required for each to find all data points within an image distance of 16 pixels, and the resultant number of processor operations required for this, in units of 10^5 operations. The last two columns give the ratios of the search space size and the processor operations between the MLG and the PMLG.

In most cases, the PMLG needs less processor operations than the MLG, except in the low data density situations. If the number of processors is not a concern, i.e. when there are sufficient to process the entire MLG data grids, then the computation time is most important. This is proportional to the search space. The ratio of the search space between the MLG and the PMLG shows that the search on the PMLG is between 9.5 and 2.3 quicker than on the MLG. In the original image, without any compression, the same search operation would use 256x256 processors with a search size of $31^2-1 = 960$ to give an overall count of 629.1 ($*10^5$) operations. Both the search space and operation counts for the MLG and the PMLG are lower than this.

# points	PMLG grid (block) size	PMLG search space (PEs)	PMLG ops ($\times 10^5$)	MLG grid size	MLG search space (PEs)	MLG ops ($\times 10^5$)	Ratio: MLG / PMLG search space	Ratio: MLG / PMLG ops
500	2304 (9)	120	2.8	523	523	2.8	4.4	1.0
	4096 (16)	224	9.2	523	523	2.8	2.3	0.3
1000	4096 (16)	224	9.2	1024	1024	10.5	4.6	1.1
	6400 (25)	360	23.0	1024	1024	10.5	2.8	0.5
5000	9216 (36)	528	48.7	5041	5041	254.1	9.5	5.2
	12544 (49)	728	91.3	5041	5041	254.1	6.9	2.8
	16384 (64)	960	157.3	5041	5041	254.1	5.3	1.6

Table 7.3 - Processor operations to find all data points within a distance of 16 pixels from each other.

7.4.4 Construction of an MLG on an SIMD array

The MLG algorithm was implemented on a single Cluster within the WPM, which allows a PMLG block to be constructed for any number of data points contained within a 16x16 space. The construction of the MLG scales linearly with the number of data points within the SIMD array. When an MLG block is required to be constructed over more than one Cluster, such as is the case in the construction of the MLG for the entire data set across an image, there will be communication and associative response combination overheads as discussed in Chapter 5.

The time taken to construct a PMLG block within a Cluster is shown in Table 7.4 - note that blocks can be constructed within each Cluster in parallel. The time taken to construct the MLG is small when compared to the total processing time available within the frame period. A speedup of four can be achieved by the use of the PMLG for the data association operation in the target tracking detailed in Section 6.5.

Block size	4	5	6
Time (μs)	238	257	276

Table 7.4 - Time taken for the construction of a PMLG block on the SIMD array within a WPM Cluster.

7.5 Summary

The load balancing techniques described in Sections 7.2, 7.3 and 7.4 can each be performed within the frame period of an image analysis system. The time taken for the three methods, the optimisation of the number of patches required for the processing of a sub-image, the packing of rectangular sub-image patches across the WPM array using the FPDS algorithm, and the compression of a sparse data set using the PMLG algorithms are given in Table 7.5. The performance advantages gained by the use of each load-balancing technique is also shown in this table. The iconic techniques enable an increase in the utilisation of the Clusters within the WPM. The sparse method enables a decrease in the search space required to find nearest neighbours. This technique also allows the replication of data between Clusters which improves the throughput of algorithms such as data association in target tracking.

Technique	Time	Advantages
Patch minimisation	0.1ms	4 * reduction in the number of SIMD patches used
FPDS Bin packing	0.5ms	2 * improvement on utilisation of PE array
PMLG	0.3ms	less PEs required for the compressed data set, Speedup of 4 on data association operations

Table 7.5 - Time taken, and advantages of, the load-balancing techniques on the WPM

The time taken for each method is a small percentage of the overall time available for processing, but the advantages gained from such techniques enable a greater utilisation of the array. This can also be viewed, for both image or sparse data operation, as a decrease in the processing time per operation.

The performance of the load-balancing operations have been described in terms of the M-SIMD array within the WPM. It is likely that the data on the Transputer MIMD array may also need to be load-balanced in some applications. Load-balancing could equally be applied in such cases. Additionally, there could be a dependency between the MIMD array and the M-SIMD array, thus increase the complexity of the load-balancing operation. This has not been considered here.

The methods described can also be applied to non-image based applications. In fact such load-balancing techniques could form part of a library which the programmer could call upon when a certain structure within the data is known to exist, or arises from the processing that is to be performed on that data. Such algorithms can be further optimised if further information on the structure of the data can be assumed.

The most convenient way in which to use the load-balancing techniques described here would be for the compiler, or the run-time operating system, to automatically invoke them, and not by the applications programmer. Thus, when the structure of the data deems it necessary and it is detected by the run-time system, a load-balancing technique could be automatically brought into use and dispensed with afterwards. Such automatic capabilities of a parallel compiler/run-time system do not exist at the present time.

Chapter 8

Conclusions

This thesis describes, a novel heterogeneous architecture, the Warwick Pyramid Machine (WPM). The design of this machine arose from an examination of the requirements of image analysis. The processing requirements include the need for data-parallel computation at a low level (for raw pixel processing), and for more flexible independent processing of small data sets (at a higher, or symbolic, level). Furthermore, a typical image analysis processing flow requires, within the low-level processing, both global and local data parallelism. Local data parallelism enables regions of the image to be addressed in an object dependant way.

The WPM contains separate processor configurations for the different processing requirements. It consists of a massively parallel Multiple-SIMD (M-SIMD) processor array, for low-level operations, and an MIMD processor, for high level operations. The M-SIMD array can operate in a synchronised SIMD mode, mimicking the action of a conventional SIMD array, or in a local autonomous mode where each SIMD patch can perform its own operation. Thus, the WPM supports both global and local data-parallel operations.

The two arrays within the WPM are tightly coupled, with each MIMD processor being associated with a single SIMD patch. Communications between the two levels takes place through the use of shared memory, and associative response mechanisms within the SIMD patch. The partitioning of the associative mechanisms, to each SIMD patch, results in multiple data-paths between the M-SIMD and MIMD arrays.

In the review contained within Chapter 2, the various options available in the construction of a parallel architecture were described. These included memory structure, processor interconnection topologies and control structure. Existing parallel machines were described, and each generally conformed to one of the two control structures, either SIMD or MIMD. Several research machines, combining SIMD and MIMD processor types were described in Chapter 3. These architectures enabled only global data-parallel and control-parallel operation.

The design and implementation of the prototype WPM is described in Chapter 4. Commercial processors were used, for both the SIMD and the MIMD arrays, in its implementation. The architecture is modular, based around the concept of a Cluster. A Cluster represents a vertical slice through the machine. Two prototype Clusters have been built enabling the communication and synchronisation mechanisms, between SIMD patches, to be tested.

The performance of the WPM, in terms of its constituent components is also described in Chapter 4. Although the peak performances enable a comparison to be made between different machines, they are not necessarily representative of the performance achievable in real applications. The peak figures can be thought of as an upper bound on the performance of the machine.

A comparison of the performance of the M-SIMD array, within the WPM, with that of a conventional SIMD architecture is given in Chapter 5. This was examined for both the possible peak advantages obtainable in specific operations and those achievable in example numerical matrix operations. It was found that the M-SIMD array performed better than a conventional SIMD array, in many instances, for the specific operations of local object processing, associative responses, and data communications. Further, the performance of matrix operations was found to be greater on the WPM M-SIMD array for specific sizes of matrices.

The mapping of an application across the multiple processing levels of the WPM is illustrated using example tracking operations in Chapter 6. Two different types of tracking applications were considered, namely that of object tracking and the tracking of feature points. The two applications differed in the density of objects across the image and resulted in the use of different algorithms in the tracking processing flow.

In the first tracking application, image processing operations where used to detect, segment, and extract information about objects form the image plane. This information was incorporated into two tracking filters, one for position and the other for size. The second application considered only the position tracking of points which resulted from a global image processing operation.

The efficient utilisation of a parallel machine is a prime factor in achieving near-peak performances. In the WPM, the utilisation of the Clusters needs to be considered at two levels. The first is to ensure that the component arrays within Clusters are being utilised. This requires the efficient partitioning of the application, between the SIMD and MIMD processors, as is seen in the tracking applications in Chapter 6. The second is to ensure that all Clusters are being utilised. This second case leads to a load-balancing problem as discussed in Chapter 7.

A comparison between the performance of machines should not be looked at in isolation. Other factors, such as gate count and clock speeds, should also be taken into account. The prototype WPM uses a low level of integration being based on the AMT Distributed Array Processor [AMT88] for the SIMD array, and Transputers for the MIMD array. The former dates back to 1973, and the latter back to 1987. The Cluster controller is also implemented using a low level of integration, and represents 10% overhead in hardware complexity when compared to a combination of SIMD and MIMD arrays [Nudd92b]. Thus, the performance improvements achieved are a direct result from this slight increase in hardware complexity.

The use of both SIMD and MIMD arrays, within the Warwick Pyramid Machine, results in increased performance on applications including target tracking. The target tracking required both image processing and the numeric processing involved in the update of Kalman filters. These operations require both data parallel and control parallel processing.

The partitioning of the SIMD array, in terms of its local autonomy and associative responses, to form an M-SIMD array results in better performance in a number of algorithms. The target tracking applications benefited from the local autonomy, and the matrix operations benefited from the associative response mechanisms for routing.

The effect of the local autonomy, within an M-SIMD machine for local data parallel processing, becomes increasingly important when compared to SIMD arrays as they scale in size. Additionally, the routing performance increase, such as that in matrix operations, will become more predominant as the computational time decreases. This could be a result of using multiple-bit processing elements. Thus, the use of M-SIMD arrays will achieve greater performance increases if the processor array size is scaled, or if the computational performance of the SIMD processors improves, compared to those within the WPM.

The size of the SIMD patch within a Cluster is a major factor in the performance improvements achievable on local data-parallel processing. The best performance improvements occur when the size of the data set is equal to, or a multiple of, this SIMD patch size. For a general purpose image/numerical analysis machine, no optimum SIMD patch size exists. An appropriate SIMD patch size can be chosen however, if the machine is to be dedicated to a particular subset of problems, a result of apriori knowledge about the local data parallel processing required.

The load-balancing algorithms resulted in an improvement between a factor of two and four for the utilisation of the WPM Cluster array, whilst also requiring little overhead. The use of a vote algorithm, a heuristic packing algorithm, and a partitioned Monotonic Lagrangian Grid have all resulted in improved load-balancing operation over conventional algorithms. These techniques might be equally applicable to other parallel machines.

A novel parallel architecture has thus been designed, implemented, and performance measurements obtained. The Warwick Pyramid Machine supports several styles of processing, both local and global data-parallel operation, and control parallel. The structure of the machine outperforms architectures from which it is derived. The analysis, contained within this thesis, adds to the ever increasing field of parallel processing.

Suggestions for further work

The WPM could be designed into an integrated form, it is feasible to consider that all the components of a Cluster may be incorporated into a single chip. This chip would form the basic building block for the whole machine. The use of a more sophisticated SIMD processor, for example a multiple-bit PE, would increase the performance of the SIMD processor array. Additionally, the use of current technology would result with a faster clock speed, further increasing the performance of a Cluster.

The SIMD patch size within the Cluster is a factor in the performance improvements achievable on the M-SIMD array. The control structure, between the Cluster controllers and the SIMD arrays, could be reconfigureable. This would mean that the controllers would not provide the instruction stream for same PEs throughout an operation, but be reconfigured depending upon the size of the objects being processed. This is similar in some respects to the design of PASM, detailed in Chapter 3, but on a larger scale.

Further complexities in the hardware system should not propagate through to the software system, which needs to be designed to be as easy as possible to use. Ideally one would like a software system which would automatically parallelise code, and in the case of the WPM, partition parts of the code across the different processor levels. At run-time, techniques for load-balancing could automatically be used when feasible. Automatic parallelisation of the software will probably result in the greater usage of parallel machines.

Bibliography

Almasi89	Almasi, G.S., and Gottlieb, A., Highy Parallel Computing, Addison-Wesley, 1989.
AMD86	Advanced Micro Devices, *AM29116 High-Performance 16-Bit Bipolar Microprocessors,* October 1986.
AMD87	Advanced Micro Devices, *Am29C331 CMOS 16-Bit Microprogram Sequencer,* September 1987.
Amdahl67	Amdahl, G.M., "Validity of the single processor approach to achieving large scale computing capabilities", In Proc. AFIPS Spring Joint Computer Conf., Atlantic City, NJ, 1967, pp. 483-485.
AMT88	Active Memory Technology Ltd., *DAP Series Technical Overview,* March 1988.
AMT90	Active Memory Technology Ltd., *Introducing the DAP/CP8 range,* April 1990.
Anandan87	Anandan, P., *Measuring Visual Motion From Image Sequences,* Ph.D. dissertation, University of Massachusetts at Amherst, May 1987.
Asthana89	Asthana, A., Jagadish, H.V., and Mathews, B.T., "Impact of Advanced VLSI Packaging on the Design of a Large Parallel Computer", In Proc. of the International Conf. on Parallel Processsing, Pensylvania, August 1989, pp. 323-327.
Atherton90	Atherton, T.J., Nudd, G.R., Clippingdale, S.C., Francis, N.D., Kerbyson, D.J., Packwood, R.A., and Vaudin, G.J., "Detection and Segmentation of Blobs using the Warwick Multiple-SIMD Architecture", In Parallel Architectures for Image Processing, SPIE, February 1990, pp. 96-104.
Atherton91	Atherton, T.J., Kerbyson, D.J., and Nudd, G.R., "Passive Estimation of Range to Objects From Image Sequences", In *British Machine Vision Conference 1991*, Mowforth, P. (Ed), Spinger-Verlag, Glasgow, September 1991, pp. 343-346.

Baker83	Baker, B.S., and Schwarz, J.S., "Shelf Algorithms for Two-Dimensional Packing Problems", SIAM J. Comput., Vol. 12, No. 3, 1983, pp. 508-525.
Ballard82	Ballard, D.H., and Brown, C.M., Computer Vision, Prentice Hall, 1982.
Bar-Shalom75	Bar-Shalom, Y., and Tse, E., "Tracking in a Cluttered Environment with Probabilistic Data Association", Automatica, Vol. 11, September 1975, pp. 451-460.
Bar-Shalom88	Bar-Shalom, Y., and Fortmann, T.E., Tracking and Data Association, Academic Press, Mathematics in Science and Engineering, Vol. 179, 1988.
Barnes68	Barnes, G.H., Brown, R.M., Kato, M., Kuck, D.J., Slotnick, D.L., and Stokes, R.A., "The Illiac IV Computer", IEEE Trans. on Computers, Vol. 17, No. 8, August 1968, pp. 746-757.
Barr89	Barr, D.C., "Computing Surface", In Proceedings in Supercomputing, Vol. 1, ASFRA, February 1989, pp. 37-43.
Batcher80	Batcher, K.E., "Design of a Massively Parallel Processor", IEEE Trans. on Comput., Vol. 29, No. 9, September 1980, pp. 836-840.
Beal90	Beal, D., and Lambrinoudakis, C., "Floating Point Support for SIMD Array Processors", Tech. Rept 511, Queen Mary and Westfield College, Dept. of Computer Science, November 1990.
Blevins90	Blevins, D.W., Davies, E.W., Heaton, R.A., and Reif, J.H., "BLITZEN: A Highly Integrated Massively Parallel Machine", J. of Parallel and Distributed Computing, February 1990.
Boris86	Boris, J., "A Vectorized 'Near Neighbours' Algorithm of Order N Using a Monotonic Logical Grid", J. of Computational Physics, Vol. 66, 1986, pp. 1-20.
Bowen82	Bowen, B.A., and Brown, W.R., VLSI Systems Design for Digital Systems Processing, Prentice-Hall, Englewood Cliffs, NJ, 1982.
Broomell83	Broomell, G., and Heath, J.R., "Classification Categories and Historical Development of Circuit Switching Topologies", ACM Computing Surveys, Vol. 15, No. 2, June 1983, pp. 95-133.
Broida86	Broida, T.J., and Chellappa, R., "Kinematics and Structure of a Rigid Object from a Sequence of Noisy Images", In. Proc. of Workshop on Motion: Representation and Analysis, IEEE, Charleston, South Carolina, May 1986, pp. 95-101.

Burt91	Burt, P.J., "Image Motion Analysis Made Simple and Fast, One Component at a Time", In *British Machine Vision Conference 1991*, Mowforth, P. (Ed), Springer-Verlag, 1991, pp. 1-8.
Cantoni86	Cantoni, V., "I.P. Hierarchical Systems: Architectural Features", In *Pyramidal Systems for Computer Vision*, Springer-Verlag, Cantoni, V., and Levialdi, S. (Eds), 1986, pp. 21-39.
Cantoni87	Cantoni, V., and Levialdi, S., "PAPIA: A Case History", *Parallel Computer Vision*, Academic Press, Uhr, L. (Ed), 1987, pp. 3-13.
Chambers92	Chambers, T., "Parallel Computers: The Performance Quest", In Computer Performance Evaluation '92, Pooley, R., and Hillston, J. (Eds), 1992, pp. 131-135.
Chen89	Chen, C., "Computing the Convex Hull of a simple Polygon", Pattern Recognition, Vol. 22, No. 5, 1989, pp. 561-565.
Choudhary92	Choudhary, A., and Ranka, S., "Parallel Processing for Computer Vision and Image Understanding", IEEE Computer, Vol. 25, No. 2, February 1992, pp. 7-10.
Christy90	Christy, P., "Software to Support Massively Parallal Computing on the MASPAR MP-1", In Proc. of COMPCON, IEEE, San Francisco, February 1990, pp. 29-33.
Clermont87	Clermont, P., and Merigot, A., "Real-time synchronization in a multi-SIMD massively parallel machine", In Proc. Workshop on Computer Architectures for Pattern Analysis & Machine Intelligence, IEEE, October 1987, pp. 131-136.
Coffman80	Coffman, E.G., Garey, M.R., Johnson, D.S., and Tarjan, R.E., "Performance bounds for Level-oriented Two-Dimensional Packing Problems", SIAM J. Comput., Vol 9, No. 4, 1980, pp. 808-826.
Coffman90	Coffman, E.G., and Shor, P.W., "Average-case analysis of cutting and packing in two dimensions", European J. of Operational Research, Vol. 44, 1990, pp. 134-144.
Cypher90	Cypher, R., Sanz, J., and Snyder, L., "Algorithms for image component labelling on SIMD mesh-connected computers", IEEE Trans. on Computers, Vol. 39, No. 2, February 1990, pp.276-281.
Danielsson81	Danielsson, P.E., "Getting the Median Faster", CGIP, Vol. 17, 1981, pp. 71-78.
Danielsson90	Danielsson, P., and Seger, O., "Generalized and Separable Sobel Operators" In *Machine Vision for Three-Dimensional Scenes*, Academic Press, 1990, pp. 347-379.

Deriche90 Deriche, R., and Faugeras, O., "Tracking Line Segments", Image and Vision Computing, Vol. 8, No. 4, November 1990, pp. 261-270.

Diefendorff92 Diefendorff, K., and Allen, M., "Organization of the Motorola 88110 Superscalar RISC Microprocessor", IEEE Micro, Vol. 12, No. 2, April 1992, pp. 40-63.

Dowsland90 Dowsland, K.A., "Efficient automated pallet loading", European Journal of Operational Research, Vol. 44, 1990, pp. 232-238.

Dowsland92 Dowsland, K.A., and Dowsland, W.B., "Packing Problems", European J. Operational Research, Vol. 56, 1992, pp. 2-14.

Du91 Du, L., Sullivan, G.D., and Baker, K.D., "3D Grouping by Viewpoint Consistency Ascent", In *British Machine Vision Conference 1991*, Mowforth, P. (Ed), Springer-Verlag, 1991, pp. 45-53.

Duda72 Duda, R.O., and Hart, P.E., "Use of the Hough Transformation To Detect Lines and Curves in Pictures", Communications of the ACM, Vol. 15, No. 1, January 1972, pp. 11-15.

Duff88 Duff, M.J.B., "Some Considerations on the Limitations of Image Processing Computer Architectures", In Proc. IAPR Workshop on Computer Vision, Tokyo, Japan, October 1988, pp. 1-5.

Duller89 Duller, A.W., Storer, R.H., Thomson, A.R., Dagless, E.L., Pout, M.R., Marriot, A.P., and Goldfinch, J., "Design of an Associative Processor Array", IEE Proceedings, Pt. E, Vol. 136, No. 5, September 1989, pp. 374-382.

Duncan90 Duncan, R., "A Survey of Parallel Computer Architectures", IEEE Computer, Vol. 23, No. 2, February 1990, pp. 5-16.

Faddeev59 Faddeev, V.N., Computational Methods of Linear Algebra, Dover Publications, 1959.

Feng81 Feng, T., "A Survey of Interconnection Networks", IEEE Computer, December 1981, pp. 12-27.

Flynn66 Flynn, M.J., "Very High-Speed Computing Systems", Proc. of the IEEE, Vol. 34, No. 12, December 1966, pp. 1901-1909.

Foster71 Foster, C.C., and Stockton, F.D., "Counting Responders in an Associative Memory", IEEE Trans. on Computers, December 1972, pp. 1580-1583.

Fountain87 Fountain, T.J., Processor Arrays, Academic Press, 1987.

Fountain88a Fountain, T.J., "Introducing Autonomy to Processor Arrays", In *Machine Vision*, Academic Press, 1988, pp. 31-58.

Fountain88b Fountain, T.J., Matthews, K.N., and Duff, M.J.B., "The CLIP7A Image Processor", IEEE Trans. on PAMI, Vol. 10, No. 3, May 1988, pp. 310-319.

Francis89 Francis, N., "PSIM", VLSI Group Memo, Computer Science Department, University of Warwick, August 1989.

Francis91 Francis, N.D., *Parallel Architectures for Image Analysis*, Ph.D. dissertation, Department of Computer Science, University of Warwick, September 1991.

Gehring90 Gehring, H., Menschner, K., and Meyer, M., "A Computer-based heuristic for packing pooled shipment containers", European J. Operational Research, Vol. 44, 1990, pp. 277-288.

Gehringer88 Gehringer, E.F., Abullarade, J., and Gulyn, M.H., "A Survey of Commercial Parallel Processors", Computer Architecture News, Vol. 16, No. 4, September 1988, pp. 75-107.

Gibbons88 Gibbons, A., and Wojciech, R., Efficient Parallel Algorithms, Cambridge University Press, 1988.

Golston90 Golston, J.E., Moss, R.H., and Stoecker, W.V., "Boundary detection in skin tumour images: an overall approach and a radial search algorithm", Pattern Recognition, Vol. 23, No. 11, November 1990, pp. 1235-1247.

Grand93 Grand Challenges 1993 : High Performance Computing and Communications. The FY 1993 U.S. Research and Development Program, National Science Foundation, Washington DC.

Grinberg84 Grinberg, J., Nudd, G.R., and Etchells, R.D., "A Cellular VLSI Architecture", IEEE Computer, Vol. 17, No. 1, January 1984, pp. 69-81.

Harris87 Harris, C.G., "Determination of Ego-motion from Matched Points", In Proc. of the Third Alvey Vision Conf., Cambridge, September 1987, pp. 189-192.

Harris88 Harris, C., and Stephens, M., "A Combined Corner and Edge Detector", In Proc. of the Fourth Alvey Vision Conf., University of Manchester, 1988, pp. 147-151.

Harris92 Harris, A., "Putting the right numbers into HDTV', Electronics world + wireless world, No. 1675, June 1992, pp. 481-485.

Heel88	Heel, J., "Dynamical systems and Motion Vision", Tech. Rept 1037, A.I. memo, MIT, April 1988.
Hennessy90	Hennessy, J.L., and Patterson, D.A., Computer Architecture a Quantitative Approach, Morgan Kaufmann, 1990.
Hennessy91	Hennessy, J.L., "Computer Technology and Architecture : An Evolving Interaction", IEEE Computer, Vol. 24, No. 9, September 1991, pp. 18-29.
Hillis85	Hillis, W.D., The Connection Machine, MIT Press, Cambridge, 1985.
Hofri80	Hofri, M., "Two-dimensional packing: Expected performance of simple level algorithms", Information and Control, Vol. 45, 1980, pp. 1-17.
Hord90	Hord, R.M., Parallel Supercomputing in SIMD Architectures, CRC Press, 1990.
Horn81	Horn, B.K.P., and Schunck, B.G., "Determining optical flow", Artificial Intelligence, Vol. 17, 1981, pp. 185-203.
Howarth88	Howarth, R.M., and Francis, N.D., "Cluster Programming Language : Definition and user manual", Tech. Rept RR 125, Department of Computer Science, University of Warwick, 1988.
Hwang80	Hwang, K., and Ni, L.M., "Resource Optimization of a Parallel Computer for Multiple Vector Processing", IEEE Trans. on Computers, Vol. 29, No. 9, September 1980, pp. 831-836.
Hwang85	Hwang, K., and Briggs, F.A., Computer Architecture and Parallel Processing, McGraw-Hill Int., 1985.
Hwang90	Hwang, K., Panda, D.K., and Haddadi, N., "The USC orthogonal multiprocessor for image processing with neural networks", In Parallel Architectures for Image Processing, SPIE, San Diego, February 1990, pp. 70-85.
Hwang91	Hwang, K., Alnuweiri, H.M., Kumar, V.K.P., and Kim, D., "Orthogonal Multiprocessor Sharing Memory with an Enhanced Mesh for Integrated Image Understanding", CVGIP: Image Understanding, Vol. 53, No. 1, January 1991, pp. 31-45.
Inmos85	Inmos Ltd., *Transputer Reference Manual,* 1985.
Inmos91	Inmos Ltd., *The T9000 Transputer,* 1991.

Jesshope87	Jesshope, C., Rushton, A., Cruz, A., and Stewart, J., "The Structure and Application of RPA - A Highly Parallel Adaptive Architecture". In *Highly Parallel Computers.* Elsvier Science, Reijns, G.L., and Barton, M.H. (Eds), 1987, pp. 81-92.
Jesshope89	Jesshope, C.R., O'Gorman, R., and Stewart, J.M., "Design of SIMD microprocessor array", IEE Proceedings, Pt. E, Vol. 136, No. 3, May 1989, pp. 197-204.
Judge92	Judge, T.R., Andonov, D., Kerbyson, D.J., and Bryanston-Cross, P.J., "Parallel Two Dimensional Phase Unwrapping", To be published.
Kalata84	Kalata, P.R., "The Tracking Index: A Generalized Parameter for Alpha-Beta-Gamma Target Trackers", IEEE Trans. AES, Vol. 20, March 1984, pp. 174-182.
Kalman60	Kalman, R.E., "A New Approach to Linear Filtering and Prediction Problems", Trans. ASME, J. Basic Engineering, Vol. 82, March 1960, pp. 34-45.
Kaplan63	Kaplan, A., "A search memory subsystem for a general purpose computer", In AFIPS Conf. Proc., Baltimore, Md, 1963, pp. 193-200.
Kerbyson92	Kerbyson, D.J., Atherton, T.J., and Nudd, G.R., "An MSIMD Architecture for Feature Tracking", In IEE Colloquium on Medium Grain Distributed Computing, London, March 1992.
Kimme75	Kimme, C., Ballard, D., and Slansky, J., "Finding Circles by an Array of Accumulators", Communications of the ACM, Vol. 18, No. 2, February 1975, pp. 120-122.
Kodak90	Kodak Ltd, *Kodak EKTAPRO Motion Analyzers,* 1990.
Kolbe90	Kolbe, R.L., Boris, J.P., and Picone, J.M., "Battle Engagement Area Simulator/Tracker", Tech. Rept 6705, Naval Research Lab, Washington DC, October 1990.
Kories86	Kories, R., and Zimmermann, G., "A Versatile Method for the Estimation of Displacement Vector Fields from Image Sequences", In Proc. Workshop on Motion: Representation and Analysis, IEEE, Kiawah Island, May 1986, pp. 101-106.
Kroger91	Kroger, B., Schwenderling, P., and Vornberger, O., "Genetic Packing of Rectangles on Transputers", In Transputing '91, Welch, P., Stiles, D., Kunii, T.L., and Bakkers, A. (Eds), IOS Press, 1991, pp. 593-608.

Kuehn85 Kuehn, J.T., Siegel, H.J., Tuomenoksa, D.L., and Adams, G.B., "The Use and Design of PASM", In *Integrated Technology for Parallel Image Processing*, Academic Press, Levialdi, S. (Ed), 1985, pp. 133-153.

Lambrinoudakis91 Lambrinoudakis, C., *A Cost-Effective Architectural Progression for Enhancing the Numeric Performance and General Purpose Capability of SIMD arrays*, Ph.D. dissertation, Queen Mary and Westfield College, September 1991.

Lawrie75 Lawrie, D.H., "Access and alignment of data in an array processor", IEEE Trans. on Computers, Vol. 24, No. 12, December 1975, pp. 1145-1155.

Lea88 Lea, R.M., and Bolouri, H.S., "Fault tolerance : step towards WSI", IEE Proc. Pt. E, Vol. 135, No. 6, November 1988, pp. 289-297.

Lea91 Lea, R.M., and Jalowiecki, I.P., "Associative Massively Parallel Computers", Proceedings of the IEEE, Vol. 79, No. 4, April 1991, pp. 469-479.

Lee87 Lee, S.Y., and Aggarwal, J.K., "Parallel 2-D Convolution on a Mesh Connected Array Processor", IEEE Trans. on PAMI, Vol. 9, No. 4, July 1987, pp. 590-594.

Lenoski92 Lenoski, D., Laudon, J., Joe, T., Nakahira, D., Stevens, L., Gupta, A., and Hennessy, J., "The DASH Prototype: Implementation and Performance", In Proc. 19th Int. Sym. on Computer Architecture, 1992, pp. 92-103.

Levialdi88 Levialdi, S., "Computer Architectures for Image Analysis", In Proc. 9th Int. Conf. on Pattern Recognition, Rome, Italy, November 1988, pp. 1148-1158.

Levitan87 Levitan, S.P., Weems, C.C., Hanson, A.R., and Riseman, E.M., "The UMass Image Understanding Architecture", In *Parallel Computer Vision*, Academic Press, Uhr, L. (Ed), 1987, pp. 215-248.

Li89 Li, K., and Cheng, K.H., "Job-Scheduling in Partionable Mesh Connected Systems", In Int. Conf. on Parallel Processing, 1989, pp. 65-72.

Liddell87 Liddell, H.M., and Parkinson, D., "Mapping Large Scale Computational Problems on a Highly Parallel SIMD Computer", In Proc. Third SIAM Conf. on Parallel Processing for Scientific Applications, Los Angeles, December 1987.

Lim87	Lim, H.S., and Binford, T.O., "Survey of Parallel Computers", In Proc. Image Understanding Workshop, Los Angeles, Califonia, February 1987, pp. 644-654.
LSI89	LSI Logic, *Shortform Catalog,* November 1989.
Maresca88	Maresca, M., Lavin, M.A., and Li, H., "Parallel Architectures for Vision", Proc. of the IEEE, Vol. 76, No. 8, August 1988, pp. 970-981.
Marr82	Marr, D., Vision, W.H. Freeman, 1982.
Marslin91	Marslin, R., Sullivan, G.D., and Baker, K.D., "Kalman Filters in Constrained Model Based Tracking", In *British Machine Vision Conference 1991*, Mowforth, P. (Ed), Springer-Verlag, 1991, pp. 371-374
Matthies89	Matthies, L., Kanade, T., and Szeliski, R., "Kalman filter-based algorithms for estimating depth from image sequences", International Journal of Computer Vision, Vol. 3, 1989, pp. 209-236.
Maybeck79	Maybeck, P.S., Stochastic models, estimation and control, Vol. 1, Academic Press, 1979.
Minor81	Minor, L.G., and Sklansky, J., "The Detection and Segmentation of blobs in infrared images", IEEE Trans. SMC, Vol. 11, No. 3, March 1981, pp. 194-201.
Nass92	Nass, R., "Massively Parallel System Delivers 68,500 MIPS", Electronic Design, Vol. 40, No. 21, October 1992, pp. 89-90.
Nevatia82	Nevatia, R., Machine Perception, Prentice-Hall, 1982.
Nickolls90	Nickolls, J.R., "The Design of the MasPar MP-1: A Cost Effective Massively Parallel Computer", In Proc. of COMPCON, IEEE, San Francisco, February 1990, pp. 399-402.
Nicole88	Nicole, D.A., "ESPRIT Project 1085 Reconfigurable Transputer Processor Architecture", In Proc. CONPAR, Manchester, September 1988, pp. 12-39.
Noble87	Noble, J.A., "Finding Corners", In Proceedings of the Third Alvey Vision Conference, Cambridge, September 1987, pp. 267-274.
Nudd88	Nudd, G.R., Howarth, R.M., Atherton, T.J., Francis, N.D., Vaudin, G.J., and Walton, D.W., "A Heterogeneous Architecture for Parallel Image Processing", In UK Information Technology, Swansea, July 1988, pp. 495-499.

Nudd89	Nudd, G.R., Atherton, T.J., Howarth, R.M., Clippingdale, S.C., Francis, N.D., Kerbyson, D.J., and Packwood, R.A., "WPM : A Multiple-SIMD architecture for image processing", In Proc. 3rd IEE Conf. on Image Processing and Its Applications, Warwick 1989, pp. 161-165.
Nudd91	Nudd, G.R., Kerbyson, D.J., Atherton, T.J., Francis, N.D., Packwood, R.A., and Vaudin, G.J., "A Massively Parallel Heterogeneous VLSI Architecture for MSIMD Processing", In *Algorithms and Parallel VLSI Architectures*, Elsevier North Holland, 1991.
Nudd92a	Nudd, G.R., Francis, N.D., Atherton, T.J., Kerbyson, D.J., Packwood, R.A., and Vaudin, G.J., "A Hierarchical Multiple-SIMD Architecture for Image Analysis", Machine Vision and applications, Vol. 5, No. 2, May 1992, pp. 85-103.
Nudd92b	Nudd, G.R., Atherton, T.J., and Kerbyson, D.J., "An Heterogeneous M-SIMD Architecture for the Kalman Filter Controllerd Processing of Image Sequences", In Proc. CVPR, IEEE, Champaign, Illinois, June 1992, pp. 842-845.
Nudd92c	Nudd, G.R., Kerbyson, D.J., and Atherton, T.J., "Pyramid Architectures for SDI Sensor Processing". In *Signal Processing Monograph*, B-K Dynamics, Palmer, P. (Ed), Chapter 7.2, 1992.
O'Gorman89	O'Gorman, R, *Design and Application of the RPA II*, Ph.D. dissertation, University of Southampton, Department of Electronics and Computer Science, 1989.
Page89	Page, I., "Graphics + Vision = SIMD + MIMD (A novel dual-paradigm approach)", In *Parallel Processing for Computer Vision and Display*, Addison-Wesley, Dew, P.M., Earnshaw, R.A., and Heywood, T.R. (Eds), 1989, pp. 89-103.
Parsytec88	Parsytec GmbH, *Parallel processing in industry*, Aachen, Germany, 1988.
Parsytec91	Parsytec GmbH, *Beyond the Supercomputer - Parsytec GC*, Aachen, Germany, 1991.
Pass85	Pass, S., "A VLSI Array Processor for Image and Signal Processing". In *Advanced Signal Processing*, Peter Peregrinus, 1985, pp. 218-20.
Patterson80	Patterson, D.A., and Ditzel, D.R., "The case for the reduced instruction set computer", Computer Architecture News, Vol. 8, No. 6, October 1980, pp. 23-33.

Patterson91 Patterson, M., Personal Communication, University of Warwick, February 1991.

Pattipatti90 Pattipatti, K.R., Kurien, T., Lee, R., and Luh, P.B., "On Mapping a Tracking Algorithm Onto Parallel Processors", IEEE Trans. on AES, Vol. 26, No. 5, September 1990, pp. 774-790.

Peterson91 Peterson, C., Sutton, J., and Wiley, P., "iWarp: A 100-MOPS LIW Microprocessor for Multicomputers", IEEE Micro, Vol. 11, No. 3, June 1991, pp. 26-29 and 81-87.

Pfeiffer89 Pfeiffer, C., and Ciplickas, "A Conceptual Design of the MSX On-Board Signal and Data Processor", Tech. Rept J0045-1, Internal Document, Hughes Aircraft Company, July 1989.

Pfister87 Pfister, G.F., Brantley, W.C., George, D.A., Harvey, S.L., Kleinfelder, W.J., McAuliffe, K.P., Melton, E.A., Norton, V.A., and Weiss, J., "An Introduction to the IBM Research Parallel Processor Prototype (RP3)", In *Experimental Parallel Computing Architectures*, Elseveir, Dongarra, J.J. (Ed), 1987, pp. 123-140.

Picone90 Picone, J.M., Lambrakos, S.G., and Boris, J.P., "Timing Analysis of the Monotonic Logical Grid for Many-Body Dynamics", SIAM J. Sci. Stat. Comput., Vol. 11, No. 2, March 1990, pp. 368-388.

Pratt78 Pratt, W., Digital Image Processing, Wiley, 1978.

Procter91 Procter, B., "Technology and Market Trends", In Proc. of Parallel Computing workshop, University of Newcastle, September 1991, pp. IX.1 - IX.7.

Rattner91 Rattner, J., "The New Age of Supercomputing", In Distributed Memory Computing, Bode, A. (Ed), Spinger Verlag, 1991, pp.1-6.

Reddaway73 Reddaway, S.F., "DAP - a Distributed Array Processor", In 1st Annual Symposium on Computer Architecture, Gainsville, Florida, December 1973, pp. 61-65.

Reddaway79 Reddaway, S.F., "The DAP Approach", Infotech State of the Art Report, Vol. 2, 1979, pp. 185-205.

Reddaway85 Reddaway, S.F., "Median Filtering on the DAP", Tech. Rept CM98, Active Memory Technology, Reading, England, October 1985.

Reddaway88 Reddaway, S.F., "Mapping Images onto Processor Array Hardware", In *Parallel Architectures and Computer Vision*, Clarendon Press, Page, I. (Ed), 1988, pp. 299-314.

Reddaway90	Reddaway, S.F., "Signal Processing on a Processor Array", In *Massively Parallel Computing with the DAP*, Pitman, Parkinson, D., and Litt, J. (Eds), 1990, Chapter 4, pp. 55-75.
Reeves80	Reeves, A.P., "On Efficient Global Information Extraction Methods for Parallel Processors", CVGIP, Vol. 14, 1980, pp. 159-169.
Reeves84	Reeves, A.P., "Parallel Computer Architectures for Image Processing", CVGIP, Vol. 25, 1984, pp. 68-88.
Rosenfeld82	Rosenfeld, A., and Kak, A.C., Digial Picture Processing, Vol. 1, Academic Press, 1982.
Rosenfeld88	Rosenfeld, A., Ornelas, J., and Hung, Y., "Hough Transform Algorithms for Mesh-Connected SIMD Parallel Processors", CVGIP, Vol. 41, 1988, pp. 293-305.
Rushton89	Rushton, A., Reconfigurable Processor-Array: a bit-sliced parallel computer, Pitman, 1989.
Schaefer87	Schaefer, D.H., Ho, P., Boyd, J., and Vallejos, C., "The GAM Pyramid", In *Parallel Computer Vision*, Academic Press, Uhr, L. (Ed), 1987, pp. 15-42.
Schalkoff89	Schalkoff, R.J., Digital Image Processing and Computer Vision, Wiley & Sons, 1989.
Segal90	Segal, A., "Heterogeneous Parallel Processor for a Model-Based Vision System", In Applications of Optical Engineering: Proc. of OE/Midwest, SPIE, 1990, pp. 601-614.
Seitz85	Seitz, C.L., "The Cosmic Cube", Communications of the ACM, Vol. 28, No. 1, January 1985, pp. 22-33.
Shu88	Shu, D.B., and Nash, J.G., "Minimum spanning tree algorithm on an image understanding architecture", In Hybrid Image and Signal Processing, SPIE, 1988, pp. 212-228.
Siegel81	Siegel, H.J., Siegel, L.J., Kemmerer, F.C., Mueller, P.T., Smalley, H.E., and Smith, S.D., "PASM : A Partionable SIMD/MIMD System for Image Processing and Pattern Recognition", IEEE Trans. on Computers, Vol. 30, No. 12, December 1981, pp. 934-947.
Simmons89	Simmons, M., Koskela, R., and Bucher, I., Instrumentation for Future Parallel Computing Systems, Addison Wesley, ACM Frontier Series, 1989.

Singh91	Singh, A., "Incremental Estimation of Image-Flow Using a Kalman Filter", In IEEE Workshop on Visual Motion, Princeton, New Jersey, October 1991, pp. 36-43.
Singh92	Singh, J.P., Holt, C., Totsuka, T., Gupta, A., and Hennessy, J.L., "Load Balancing and Data Locality in Hierarchical N-body Methods", Tech. Rept CSL-TR-92-505, Stanford University, Computer Systems Lab, 1992.
Slorach88	Slorach, F., and Alsford, J.R., "A RAM Based CMOS Histogrammer Integrated-Circuit", IEEE Trans. on Nuclear Science, Vol. 35, No. 1, 1988, pp. 209-212.
Sorenson85	Sorenson, H.W., Kalman Filtering: Theory and Applications, IEEE Press, 1985.
Stephens89	Stephens, M., and Harris, C., "3D Wire-Frame Integration from Image Sequences", Image and Vision Computing, Vol. 7, No. 1, February 1989, pp. 24-30.
Stokar92	Stokar, D., "A Heterogeneous Multiprocessor System for Real Time Image Processing", In Proc. WOTUG 15th Technical meeting, IOS Press, August 1992.
Suetens92	Suetens, P., Fua, P., and Hanson, A.J., "Computational Strategies for Object Recognition", ACM Computing Surveys, Vol. 24, No. 1, March 1992, pp. 5-61.
Tanimoto85	Tanimoto, S.L., "Architectural Issues for Intermediate-Level Vision", Tech. Rept 85-08-11, Dept. of Computer Science, University of Washington, Seattle, August 1985.
Tanimoto86	Tanimoto, S.L., "Paradigms for Pyramid Machine Algorithms", In *Pyramidal Systems for Computer Vision*, Springer-Verlag, Cantoni, V., and Levialdi, S. (Eds), 1986, pp. 173-194.
Tanimoto87	Tanimoto, S.L., Ligocki, T.J., and Ling, R., "A Prototype Pyramid Machine for Hierarchical Cellular Logic", In *Parallel Computer Vision*, Academic Press, Uhr, L. (Ed), 1987, pp. 43-85.
TMC89	Thinking Machines Corporation, *Connection Machine Model CM-2 Technical Summary*, May 1989.
TMC92	Thinking Machines Corporation, *The Connection Machine CM-5 Technical Summary*, January 1992.
Trew91	Trew, A., and Wilson, G., Past, Present, Parallel, Springer-Verlag, 1991.

Turek92	Turek, J., Wolf, J.L., and Yu, P.S., "Approximate Algorithms for Scheduling Parallelizable Tasks", In Proc. Performance Evaluation Review, ACM, Rhode Island, June 1992, pp. 225-236.
Uhr82	Uhr, L., "Comparing Serial Computers, Arrays, and Networks Using Measures of 'Active Resources'", IEEE Trans. on Computers, Vol. 31, No. 10, October 1982, pp. 1022-1025.
Vaudin89	Vaudin, G.J., Nudd, G.R., Atherton, T.J., Clippingdale, S.C., Francis, N.D., Howarth, R.M., Kerbyson, D.J., Packwood, R.A., and Walton, D.W., "A Generalised Parallel Architecture for Image Based Algorithms", In Proc. Eurographics, Hamburg, September 1989.
Vaudin91	Vaudin, J., *A Unified programming system for a multi-paradigm Parallel architecture*, Ph.D. dissertation, Department of Computer Science, University of Warwick, September 1991.
Vega89	Vega-Riveros, J.F., and Jabbour, K., "Review of motion analysis techniques" IEE Proc. Pt. I, Vol. 136, No. 6, December 1989, pp. 397-404.
Walton89	Walton, D.W., "The WPM Count Chip", VLSI Group Memo, Computer Science Dept, University of Warwick, 1989.
Weems89	Weems, C.C., Levitan, S.P., Hanson, A.R., and Riseman, E.M., "The Image Understanding Architecture", International Journal of Computer Vision, Vol. 2, 1989, pp. 251-282.
Weems91	Weems, C.C., Hanson, A.R., and Riseman, E.M., "The Architectural Requirements of Image Understanding with Respect to Parallel Processing", Proc. of the IEEE, Vol. 79, No. 4, April 1991, pp. 537-548.
Williams88	Williams, L.R., and Hanson, A.R., "Depth from Looming Structure", In Image Understanding Workshop, DARPA, April 1988, pp. 1047-1051.
WSTL90	Warwick Strategic Technology Laboratories Ltd., *Image Processing Algorithms from 2D Image Arrays for Fuzes*, Report for RARDE Contract MGW31B/2150, University of Warwick Science Park, England, 1990.

Appendix A

Cluster Bus Ports

The following is a full list of all ports, and their function, that exist on the Cluster bus. The names used for each port are those used by the Cluster assembler (CLASS) and the number in brackets indicate the number of the port, out of the possible 32. Each port is indicate as either being a source port, able to write data onto the Cluster bus, a destination port, able to read data from the Cluster bus, or both. Example use of each of the ports is given using CLASS.

ALU (0)

(src) The internal Y-bus of the ALU appears on the Cluster bus. Note the Y-bus is used in operations such at SORR as well as SORY. A special case exists when the immediate operand is placed on the Cluster bus in the first half cycle and latched at the mid point into the D-latch of the ALU. The Y-bus of the ALU is selected as source for the second half cycle. Note that the D-latch is transparent in all other cases. These two cases are shown in the following CLASS code.

```
/* ALU as source only */
      ; MOVE SORY R00   ;           ; EDGE = ALU
/* immediate ALU cycle where EDGE takes the value of R00+16 */
      ; ADD TODRR R00   ;           ; 0x10 EDGE = ALU
```

IMMED (2)

(src) Places the immediate operand on the Cluster bus.

```
/* source a value of 0x1232 (hex) on the Cluster bus */
      ;                 ;           ; 0x1232
```

SEQ (3)

(src) Places the value of the D outputs from the sequencer onto the Cluster bus.

(dest) The sequencer may read its D inputs at any time except when selected as source. Note, due to timing constraints, when the sequencer is performing an instruction using the D inputs to calculate the next address, only the immediate operand can be used as source, e.g. in a BRA_D instruction. If the value is to come from another source then it should be placed on the sequencers stack and used in the next cycle. However, other instructions, such as FOR_D, can be used with any source.

```
/* push value from ALU onto the stack and use in next cycle */
PUSH_D; MOVE SORY R00   ;           ; SEQ = ALU
BRA_S ;                 ;           ;
```

PEINV (4)

(dest) This Performs a logical AND between the invert bit of the DAP control word and the LSB of the latched data bus. This provides a broadcast facility to the SIMD array. A DAP instruction, such as QT, can be used in conjunction with PEINV. When PEINV is zero, the DAP instruction effectively becomes QF. Note that the PEINV latch has to be written to the cycle before being used, and is automatically set the cycle after.

```
/* PEINV = bit0 of R00 */
      ; MOVE SORY R00   ;           ; PEINV = ALU
/* AND function performed */
      ;                 ; QT        ;
/* PEINV is now set and can be used again */
```

COND (5)

(src) Sets up the condition code that is selected by the sequencer for condition branching operations. Valid condition codes are : C (Carry), N (Negative), V (Overflow), Z (Zero), ULEB (unsigned less than or equal borrow mode), ULE (unsigned less than or equal), SLT (signed less than), SLE (signed less than or equal), ANY (any output from count), SN (synchronise on north), SS (synchronise on south), SE (synchronise on east), SW (synchronise on west).

```
start:
/* set up condition code register on ALU operation */
      ; MOVE SORY R00          ;           ; COND = Z
BRCC_D;                       ;           ; SEQ = $start
```

EDGE (8)

(src) Places the value of the DAP Edge Register onto the Cluster bus. Note there is a one cycle delay when the between the DAP array writing to the Edge Register and it being able to be used on the Cluster bus.

```
/* 2 cycles to read column to edge */
      ;                 ; RXO       ;
      ;                 ; RXO       ;
/* One cycle delay */
      ;                 ;           ;
/* the Edge Register can now be used on the Cluster bus */
      ;                 ;           ; ALU = EDGE
```

(dest) The value on the Cluster bus is written into the Edge Register. Note that it is possible to write to the Edge Register from the Cluster bus and the DAP array simultaneously. This actually results in the Edge Register taking the value from the Cluster bus, but also sets an error bit - see PERR.

```
      ;                 ;           ; EDGE = ALU
```

PEADDR (9)

(dest) Sets the memory address for the DAP RAM. The prototype Clusters have 8Kbits of memory per PE starting at address 0x2000. PSIM, the Cluster simulator, has 1Kbits of memory per PE starting at 0. Note that the address written to the PEADDR destination can be used by the DAP instruction in the same cycle. This is due to the one cycle pipelined operation of the DAP PEs.

```
      ;                 ;           ; PEADDR = 0x200
```

RCSEL (10)

(dest) This selects which column/row (0-15) is to be read in a DAP associative response operation (DAP group 2 instruction). It must be written to one cycle before the DAP instruction is specified.

```
/* select row/column 4 */
        ;                 ;            ; RCSEL = 4
        ;                 ; RXO        ;
        ;                 ; RXO        ;
        ;                 ;            ;
/* the Edge register now contains the associative response */
```

SHFTCTL (11)

(dest) Specifies what happens when data is shifted on the array. It is used to zero inputs on the edges of a Cluster, or on the edge of the whole SIMD array. It uses the lowest six bits from the Cluster bus, each with a specific use :

bit 0 - Cluster mode.

0 = Cluster mode, 1 = array mode. In Cluster mode the edges of the 16x16 DAP array within a Cluster are wrapped around to the opposite edge. In array mode, the edges are connected to adjacent Clusters.

bit 1 - cyclic mode

0 = edges zeroed on shifting, 1 = data wrapped around. Has no effect in array mode.

bit 2 - 0 = south edge zeroed on a shift north,	1 = passes data through
bit 3 - 0 = west edge zeroed on a shift east,	1 = passes data through
bit 4 - 0 = north edge zeroed on a shift south,	1 = passes data through
bit 5 - 0 = east edge zeroed on a shift west,	1 = passes data through

DCTRL (12)

(dest) The D-plane control. When Bit 0 of this latch is set, the D-plane will either be loaded from, or in the DAP PEs, depending upon DAP instruction. Note, this bit must be reset immediately after use. The D-plane has not been used within the prototype Cluster.

```
/* write Q ANDed with D to memory */
        ;                 ; SQ         ; DCTRL = 1
/* reset D-plane control bit */
        ;                 ;            ; DCTRL = 0
```

PERR (13)

(src) Indicates if an error has occurred, since reset on the SIMD array. Two error conditions exist, these are :

1) if the edge register has been updated from both the Cluster bus and the DAP array in the same cycle

2) if shifting on the array took place while in array mode and the adjacent Cluster was not shifting in the same direction. Note, the shift direction that caused the error is also stored.

A total of 6-bits are used to store the error bits :

bit 0-	if set indicates shift error occurred when shifting north
bit 1 -	if set indicates shift error occurred when shifting south
bit 2 -	if set indicates shift error occurred when shifting east
bit 3 -	if set indicates shift error occurred when shifting west
bit 4 -	if zero indicates an error when shifting in array mode
bit 5 -	if zero indicates an error with the Edge Register has occurred

(dest) Clears the error flags when written to.

COUNT (16)

(src) This port reads the number of bits set from the SIMD array. It is active three cycles after the data was active on the SIMD memory lines.

```
/* make DAP memory line active */
      ;                  ;            ; PEADDR = 0x2102
      ;                  ;            ;
      ;                  ;            ;
      ;                  ;            ;
/* Can now read the value from the COUNT */
      ;                  ;            ; ALU = COUNT
```

RAM (24)

(src) Places the value of the shared memory RAM latch onto the Cluster bus. Note, reading the shared memory takes three cycles.

```
/* set up address to be read by writing to ADDR */
      ;                  ;            ; ADDR = 0
/* data read from RAM into RAM latch */
```

```
        ;                  ;            ;
/* data read from the RAM latch across the Cluster bus */
        ;                  ;            ; EDGE = RAM
```

(dest) Used in conjunction with ADDR to write a value into the shared memory. This is a three cycle operation, but appears as two cycles in CLASS and can be overlapped. The two instructions have to be performed immediately following each other.

```
/* put value to be written into the RAM latch */
        ;                  ;            ; RAM = 10
/* write shared memory address for the data to ADDR */
        ;                  ;            ; ADDR = 200
/* data is now written into the shared memory */
        ;                  ;            ;
```

COMMS (25)

(src) Places the value from the Transputer latch onto the Cluster bus. Used for debugging only.

(dest) Places the value of the Cluster bus into a latch which can be read by the Transputer. Used for debugging only.

ADDR (26)

(dest) This is the shared memory address port. For an explanation of its use see RAM above.

LEDS (30)

(dest) Displays a value from the Cluster bus on a set of sixteen leds, one for each bit of the Cluster bus. Note that this is implemented in the hardware only and not the simulator, PSIM.

Appendix B

The Cluster Assembler

An overview of the Cluster Assembler, CLASS, with example routines is given here. Each CLASS instruction contains four fields for the sequencer, the ALU, the DAP, and the Cluster bus control. Each field is separated by a semi-colon, thus a single instruction has the form :

```
Sequencer; ALU          ; DAP        ; Cluster bus control
```

The assembler makes explicit the memory space used for program variables and code. Three memory spaces exist within a Cluster, those of the instruction memory, the controller's shared memory and the PE memory. Three labels are used to denote the different memory spaces, `CSEG`, `DSEG` and `PSEG` respectively. Further, labels are used to denote either the address of variables, within the `DSEG` and `PSEG` memory spaces, or code in the `CSEG` memory space. Labels are denoted by a character string followed by a colon.

The assembler also uses the C Pre-Processor, thus enabling C constructs, such as `#define`, to be used for macro definitions. The sequencer, ALU and DAP instruction formats are described below. The Cluster bus ports are detailed in Appendix A.

The Sequencer

Only a subset of the functionality of the sequencer is used within the controller. Functions not used are: its multiway branching, 'C' style case statements which can be achieved by using 'if then else' structures; master/slave operation; stack extension, for adding a deeper stack more than 33 words; and interrupts. Full details on the sequencer can be found in its data sheet [AMD87].

The sequencer has a total of 64 instructions. The following is a list of mnemonics which are used within the build up of each instruction.

1) The data elements

D	- the cluster bus connection
S	- the internal stack
C	- the internal counter

2) Instructions, with abbreviations in brackets

BRA	(B)	- goto
CALL	(C)	- subroutine call, pushing current address+1 onto the stack
EXIT	(XT)	- goto, popping the stack
DJMP	(DJ)	- if C=1 then C=C-1 and goto, else C=C-1
RET		- pop address from the stack
FOR		- initialise a for loop, pushing address+1 on stack and loading counter
POP		- pop the stack
PUSH		- push onto the stack
LOOP		- initialise a loop, pushing address+1 onto the stack
CONT		- no-operation

The abbreviated instructions are used in conditional operations, by using the abbreviations appended by CC, for condition code being true, and NC, for condition code being false. They are also combined with one of the data elements, either D or S. The FOR loop can use only the D value, and POP/PUSH can use D or C values. Thus an instruction such as XTNC_D means 'if condition code is false then exit using the value from the D bus input'.

The ALU

The instruction to the ALU is complex and a careful understanding of the data sheet [AMD86] is required if the programmer wishes to use this part to its full advantage. The main operations, in the form of single and two operand instructions are explained below, some of which are used in subsequent examples.

The opcode for single and two operand instructions is made up of three parts. Firstly the operation is specified :-

Single operand	Two operand
`MOVE` (copy)	`SUBR` (subtract operand 1 from 2)
`COMP` (complement)	`SUBS` (subtract operand 2 from 1)
`INC` (increment)	`ADD` (addition)
`NEG` (logical negation +1)	`SUBRC`, `SUBSC`, `ADDC` (with carry)
	`AND`, `NAND`, `EXOR`, `NOR`, `OR`, `EXNOR` (bitwise logical)

The second part is a composite formed from either SO (single operand), or TO (Two Operand), the source operand(s), and then the destination operand. These operands can be either: D, the input D latch value from the Cluster bus; R, a register from the internal register file; A, the accumulator; or Z, a zero.

If an internal register is used then its number must also be specified as the third part of the ALU opcode. If no register is used, then the third part denotes the destination operand, which is dropped from the second part of the opcode. This is either A, or Y (the internal bus), preceded by 'NR'.

Example include:-

'`INC SOA NRY`' meaning increment the single operand A and place the result on the Y internal bus;

'`ADD TODRR R06`' meaning add the two operands D and R06 putting the result in R06.

N.B. the internal Y bus is used to transfer the result of both these operations to their destinations and can be made to source the Cluster bus with the resultant value.

The DAP PEs

The DAP is programmed using the same the APAL mnemonics as defined in the DAP series Technical overview [AMT88]. All function groups except 6, 14 and 15 have been implemented as discussed in Section 4.4. The mnemonic of a DAP instruction is a composite of: the destination operand(s); the operation to be performed; and the source operand(s), in that order.

The PE one-bit registers are referred to as Q, C, A and S, the Edge Register, across the rows of the array, as R, which is modified with to OR when it broadcasts down the columns of the array. A logic one is indicated by T and zero by F. Operations which may be performed are: addition, indicated by P; and a logical AND, indicated by M. An optional N may be used to invert the source, and I makes the instruction activity controlled by the A register. A suffix to the instruction indicates the direction of shifts, and addition in vector operations, in the form .N, .E, .W, .S, for North, East, West and South communications respectively.

Examples of the DAP instructions include :-

`ASN`	-	A = inverted store plane
`CQPQS`	-	Q = Q + S, C = carry from this operation
`QRO`	-	Q = Edge Register across columns
`AMQ.N`	-	A = A and (Q shifted N)

The DAP instruction set is highly irregular, in that the functions that apply to one flag may not necessarily be performed to the other flags. The activity controlled operations take two cycle and the vector additions, four. For these operations, the instruction is repeated in all required cycles, and if not supplied is treated as an error at compilation.

Example CLASS code

Two example CLASS operations are included below. The first is the calculation of the Sobel filter, which uses macro definitions of functions for addition and copy etc. The second is a code segment for the calculation of the median across the Cluster SIMD array using the SIMD associative count.

Sobel Filter CLASS code

```
/* Macro definitions */
#define r1  R10
#define r2  R11
#define r3  R12
/* abs : srcdest <- (0 - srcdest) if C clear */
#define abs(nbits,srcdest)                                      \
       ;                  ; QCN        ;                        \
       ;                  ; AQ         ;                        \
       ; MOVE SODR r1     ; QT         ; srcdest-1              \
FOR_D  ;                  ; CQ         ; nbits                  \
       ; INC SORR r1      ; CQPCSN     ; PEADDR=ALU             \
       ;                  ; SIQ        ;                        \
DJMP_S;                   ; SIQ        ;

/* add : dest <- (src1 + src2), Leaves carry in C */
#define add(nbits,dest,src1,src2)                               \
       ; MOVE SODR r1     ;            ; dest-1                 \
       ; MOVE SODR r2     ;            ; src1-1                 \
       ; MOVE SODR r3     ;            ; src2-1                 \
FOR_D  ;                  ; CF         ; nbits                  \
       ; INC SORR r2      ; QS         ; PEADDR=ALU             \
       ; INC SORR r3      ; CQPCQS     ; PEADDR=ALU             \
DJMP_S; INC SORR r1       ; SQ         ; PEADDR=ALU

/* addn : dest <- (src1 + src2.dir) */
#define addn(nbits,dest,src1,src2,dir)                          \
FOR_D  ; MOVE SOZR r1     ; CF         ; nbits                  \
       ; ADD TODRY r1     ; QS         ; src2 PEADDR=ALU        \
       ; INC SORR r1      ; QQ.dir     ;                        \
       ; ADD TODRY r1     ; CQPCQS     ; src1-1 PEADDR=ALU      \
DJMP_S; ADD TODRY r1      ; SQ         ; dest-1 PEADDR=ALU

/* copyn : dest <- src.dir */
#define copyn(nbits,dest,src,dir)                               \
FOR_D  ; MOVE SOZR r1     ;            ; nbits                  \
       ; ADD TODRY r1     ; QS         ; src PEADDR=ALU         \
       ; INC SORR r1      ; QQ.dir     ;                        \
DJMP_S; ADD TODRY r1      ; SQ         ; dest-1 PEADDR=ALU

/* storecan - store c in store */
#define storecan(dest)                                          \
       ;                  ; QC         ;                        \
       ; ADD TODRY r1     ; SQ         ; dest PEADDR=ALU
```

```
/* subn : dest <- (src1 - src2.dir) */
#define subn(nbits,dest,src1,src2,dir)                            \
       ;                ; QT         ;                            \
FOR_D  ; MOVE SOZR r1   ; CQ         ; nbits                      \
       ; ADD TODRY r1   ; QSN        ; src2 PEADDR=ALU            \
       ; INC SORR r1    ; QQ.dir     ;                            \
       ; ADD TODRY r1   ; CQPCQS     ; src1-1 PEADDR=ALU          \
DJMP_S; ADD TODRY r1    ; SQ         ; dest-1 PEADDR=ALU

/* Start of main Sobel code */
        PSEG            /* Cluster SIMD memory */
image2: DEFW   0        /* image2 = 2 * image */
image:  DEFS   8
        DEFW   0,0      /* pad image to 10 bits, same size as temp*/
temp:   DEFS   10
tempx:  DEFS   11
result: DEFS   11
        CSEG            /* Cluster instruction memory */
sobel:
      /* Calculate vertical Gradient */
      addn(9,$temp,$image2,$image,E)
      storecan($temp)
      addn(10,$temp,$temp,$image,W) /*temp=2*image+image.E+image.W*/
      copyn(10,$tempx,$temp,S)
      subn(10,$tempx,$tempx,$temp,N)  /* tempx=temp.N-temp.S */
      abs(10,$tempx)                  /* negate if tempx < 0 */

      /* Calculate horizontal gradient */
      addn(9,$temp,$image2,$image,N)
      storecan($temp)
      addn(10,$temp,$temp,$image,S)
      copyn(10,$result,$temp,E)
      subn(10,$result,$result,$temp,W)
      abs(10,$result)
      add(10,$result,$result,$tempx)
      storecan($result)               /* result=|tempx|+|result| */
      HALT;;;
```

Rank Order Filter CLASS code

```
#define nbits      0x8         /* number of bits in data word */
#define RANK       112         /* Rank-1 required */

      PSEG
image:DEFS 8                   /* image data */
mask: DEFS 1                   /* resultant mask */
tmp:  DEFS 1

      CSEG               /* Cluster Code */
      ;                  ; AR         ; EDGE = 0xFFFF
      ;                  ; AMRO       ;
      ;                  ; QA         ;
      ;                  ; SQ         ; PEADDR = $mask

/* main loop */
      ; MOVE SODR R03    ;            ; ALU = RANK
FOR_D ; MOVE SOZR R01    ;            ; SEQ = nbits
      ; INC SORR R01     ; AS         ; PEADDR = $mask
      ; SUBS TODRY R01   ; AMS        ; $image+8  PEADDR = ALU
      ;                  ; QA         ; COND = N
      ;                  ; SQ         ; PEADDR = $tmp
      ; MOVE SODR R02    ;            ; ALU = COUNT
      ;                  ; AS         ; PEADDR = $mask
      ; SUBS TODRY R01   ; AMSN       ; $image+8  PEADDR = ALU
      ; MOVE SOD NRA     ;            ; ALU = COUNT
      ; SUBR TORAY R03   ;            ;
BRCC_D; SUBS TORAA R02   ;            ; SEQ = $zero
DJMP_S;                  ; SQ         ; PEADDR = $mask
BRA_D ;                  ;            ; SEQ = $end
zero:
      ; SUBR TORAY R03   ;            ;
BRCC_D;                  ; QA         ; SEQ = $next
      ;                  ; SQ         ; PEADDR = $mask
next:
DJMP_S;                  ;            ;
end:
      ;                  ; AS         ; PEADDR = $mask
      ;                  ;            ;
      ;                  ;            ;
      ;                  ;            ;
      ; MOVE SODR R04    ;            ; ALU = COUNT
HALT  ;                  ;            ;
```

Appendix C

The Kalman Filter

The Kalman filter [Kalman60] uses a state-space representation for a linear random system being modelled. The evolution of the state from one time point to the next in a discrete system is written as :-

$$\mathbf{x}(n+1) = \mathbf{F}(n)\mathbf{x}(n) + \mathbf{G}(n)\mathbf{u}(n) + \mathbf{v}(n)$$

where

$\mathbf{x}(n)$	is the system state at time n
$\mathbf{u}(n)$	is a known control input
$\mathbf{v}(n)$	is white, zero-mean Gaussian, additive noise with covariance $\mathbf{Q}(n)$
$\mathbf{F}(n)$	is the state transitional matrix dictating how each state evolves from one time period to the next
$\mathbf{G}(n)$	is the control input gain which dictates how much of the control input enters the system.

The white noise $\mathbf{v}(n)$ is used to model the uncertainty of the system, i.e. if the system model is accurate then $\mathbf{v}(n)$ is small, or if it contains uncertainties then $\mathbf{v}(n)$ large. Associated with the system states is its covariance, $\mathbf{P}(n)$, and is a measure of the amount of noise within the system. For simplification it is assumed that the control input $\mathbf{u}(n)$ is zero in the following formulation of the Kalman filter.

The output of the system, sometimes described as the observation or measurement model, is a linear combination of the observable state and the measurement noise. It can be written as :-

$$\mathbf{z}(n) = \mathbf{H}(n)\mathbf{x}(n) + \mathbf{w}(n)$$

where

z(n)	is the measurement vector
w(n)	is white, zero-mean Gaussian, additive noise with covariance **R**(n)
H(n)	is the measurement matrix relating the measurements to the system states

The white noise w(n) is the amount of noise contained within the measurements.

One iteration of the Kalman filter is performed for each new set of measurements, z(n). It has the effect of updating the estimates of the system being modelled along with its covariance P(n). The sequence of operations performed for one iteration of a Kalman filter is :-

- the one step prediction of the system
- the observation prediction
- the update of the predictions with the new measurements

The operations required are listed below.

The one step prediction of the system states, $\hat{\mathbf{x}}(n+1|n)$, and associated error covariance matrix, $\mathbf{P}(n+1|n)$, is given by :-

$$\hat{\mathbf{x}}(n+1|n) = \mathbf{F}(n)\hat{\mathbf{x}}(n|n)$$

$$\mathbf{P}(n+1|n) = \mathbf{F}(n)\mathbf{P}(n|n)\mathbf{F}'(n) + \mathbf{Q}(n)$$

The predicted measurement, $\hat{\mathbf{z}}(n|n-1)$, and associated error covariance, S(n), is given by :-

$$\hat{\mathbf{z}}(n|n-1) = \mathbf{H}(n)\hat{\mathbf{x}}(n|n-1)$$

$$\mathbf{S}(n) = \mathbf{H}(n)\mathbf{P}(n|n-1)\mathbf{H}'(n) + \mathbf{R}(n)$$

The update of the system state estimates and error covariance is given by :-

$$\hat{\mathbf{x}}(n|n) = \hat{\mathbf{x}}(n|n-1) + \mathbf{K}(n)\left[\mathbf{z}(n) - \hat{\mathbf{z}}(n|n-1)\right]$$

$$\mathbf{P}(n|n) = [\mathbf{I} - \mathbf{K}(n)\mathbf{H}(n)]\,\mathbf{P}(n|n-1)$$

where

I is the identity matrix

K(n) is known as the Kalman Gain matrix

The Kalman gain matrix is given by :-

$$\mathbf{K}(n) = \mathbf{P}(n|n-1)\mathbf{H}'(n)\mathbf{S}(n)^{-1}$$

The Kalman gain matrix determines how much of the innovation, the difference between the observation and the predicted observation, $\mathbf{z}(n) - \hat{\mathbf{z}}(n|n-1)$, is to be incorporated into the current estimate of the system.

An initial value for the system states, $\hat{\mathbf{x}}(0)$, is require along with its error covariance, **P**(0), before the Kalman filter can be initiated. The initial value of the states depends heavily upon the system being modelled but in some instances, when there is a one to one mapping between modelled states and observations, the first observation may be taken as the initial state.

The Extended Kalman Filter

The extended Kalman filter (EKF) is used when the system being modelled is non-linear. This filter linearises the non-linear system around the predicted state at each time point. This can lead to a sub-optimal estimation process, but can be implemented if a linearisation function is available for the system and the observation models. The operations involved in the EKF are very similar to that of the linear Kalman filter. The system model and measurement models given by :

$$\mathbf{x}(n+1) = \mathbf{f}(\mathbf{x},\ n) + \mathbf{v}(n)$$

$$\mathbf{z}(n) = \mathbf{h}(\mathbf{x},\ n) + \mathbf{w}(n)$$

where

f( ) is the state linearisation equation

h( ) is the measurement linearisation equation.

x(n), **z**(n), **v**(n), and **w**(n) are the same as in the linear Kalman filter.

The Jacobians of the state linearisation equation, $\frac{\partial f}{\partial x}\Big|_{x=\hat{x}}$, and observation linearisation model, $\frac{\partial h}{\partial x}\Big|_{x=\hat{x}}$, the used in the first order EKF.

The EKF state prediction equations are given by :

$$\hat{x}(n+1|n) = f(x(n|n), n)$$

$$P(n|n-1) = \frac{\partial f}{\partial x}\Big|_{x=\hat{x}} P(n-1|n-1) \frac{\partial f}{\partial x}\Big|_{x=\hat{x}} + Q(n)$$

The EKF measurement prediction equations are given by :

$$\hat{z}(n|n-1) = h(\hat{x}(n|n-1)$$

$$S(n) = \frac{\partial h}{\partial x}\Big|_{x=\hat{x}'} P(n|n-1) \frac{\partial h}{\partial x}\Big|_{x=\hat{x}'} + R(n)$$

The EKF update equations are the same as those for the linear Kalman filter. The description, for both the Kalman filter and the EKF above, have been kept general and no specific system or measurement models have been used.