Transputer Multi-Processing
and Other Topics
in Machine Vision

A thesis
submitted for the requirements
of the degree
of
Doctor of Philosophy
(Electrical and Electronic Engineering)
in the
University of Canterbury
by
R.F. Browne, B.Sc.(Hons), M.E.

University of Canterbury
1989
A grading table in a kiwifruit packhouse

A kiwifruit grading chart

The lighting provided for an optical size grader
NEW ZEALAND-KIWIFRUIT
EXPORT GRADE STANDARDS

ALLOWANCE FOR SKIN DEFECTS

- Hayward Mark
- Proximity Mark
- Limb Rub
- Flats
- Water Stain
- Greedy Scale
- Sooty Mould
- Healed Botrytis
- Insect Chewing
- Sunburn
- Drop Shoulder

OUT OF GRADE
# Contents

1 Introduction 1

1.1 Introduction ......................... 1

1.2 Kiwifruit Industry .......................... 3

1.2.1 History and Outlook ................. 3

1.2.2 New Zealand Kiwifruit Authority .... 4

1.2.3 Characteristic Fruit Defects .......... 4

1.2.4 Fruit Defects Resulting from the Environment .......... 6

1.2.5 Pests and Diseases .................... 8

1.2.6 Harvesting, Grading, and Packing .......... 10

1.3 Characteristics of Machine Vision ......... 11

1.3.1 Basic Functions of Machine Vision ...... 11

1.3.2 Requirements .......................... 12

1.4 The Organisation of this Thesis ......... 14

2 Multiprocessor Systems 17

2.1 Introduction .......................... 17

2.2 Bus-Based Systems ..................... 18

2.3 Dataflow Systems ........................ 19

2.4 Multiprocessors Based on Dynamic Interprocessor Links .... 21
2.5 Multiprocessors Based On Static Interprocessor Links .......... 21
2.6 Multiprocessor Software ................................... 23
2.7 Synchronization Failure .................................... 24
2.8 Deadlock ................................................... 25
  2.8.1 Deadlock Detection .................................... 28
  2.8.2 Deadlock Prevention ................................... 31
  2.8.3 Deadlock Avoidance ................................... 32
  2.8.4 Deadlock Resolution ................................... 33
2.9 Conclusions ................................................ 33

3 Symmetric degree Four Chordal Ring Networks .................... 35
  3.1 Introduction .............................................. 35
  3.2 Multiprocessors Based On Static Interprocessor Links .......... 36
  3.3 Chordal Ring Networks .................................... 37
  3.4 Analysis of Chordal Ring Networks .......................... 37
  3.5 A Special Class Of Chordal Ring Network .................... 40
  3.6 Extension To Networks Of Any Size .......................... 46
  3.7 Modified Chordal Rings .................................... 47
    3.7.1 Broken Chordal Rings ................................ 47
    3.7.2 Mapping Cellular Arrays Onto Chordal Rings ............ 49
  3.8 Performance .............................................. 50
  3.9 Conclusions .............................................. 53
  3.10 Appendices to This Chapter ............................... 53
    3.10.1 Appendix 3.1 : Proof Of Equation (3.3) ............... 53
    3.10.2 Appendix 3.2 : A Routing Algorithm for a Chordal Ring .. 54
    3.10.3 Appendix 3.3 : A Routing Algorithm for a Broken Chordal Ring . 55
5.6 Memory Reference for Point Operations ........................................... 115
5.7 Memory Reference for Neighbourhood Operations ............................ 117
5.8 Memory Reference for Medium Grain Operations .............................. 118
5.9 Memory Reference for Large Grain Operations ................................. 121
5.10 Memory Reference for Global Operations ....................................... 124
5.11 Discussion ................................................................................. 126
5.12 Conclusions .............................................................................. 127

6 Lighting and the Detection of Projections .................................... 128
   6.1 Introduction ............................................................................. 128
   6.2 Geometry .............................................................................. 130
   6.3 Lighting Model ....................................................................... 132
   6.4 Fitting To General Cylinders .................................................. 133
   6.5 Fitting To Kiwifruit ................................................................ 134
   6.6 Approximations, and Compensation for Dip .............................. 134
   6.7 Geometry Of Shadows From Projections On Kiwifruit ............... 136
   6.8 Combination of Shadows and the Lighting Model ...................... 139
   6.9 Results .................................................................................. 142
      6.9.1 Shadow Profile ................................................................. 142
      6.9.2 Detectability — Theory ..................................................... 143
      6.9.3 Detectability — Experimental ........................................... 144
   6.10 Conclusions .......................................................................... 147

7 Algorithms for Defect Detection in Kiwifruit .................................. 149
   7.1 Introduction ............................................................................. 149
   7.2 Survey of Prior Work .............................................................. 150
   iv
9 Discussion and Conclusions

9.1 Introduction ................................................................................. 201

9.2 Review of This Thesis ................................................................. 201

9.2.1 Overview ................................................................................. 202

9.2.2 Chapter 2: Multiprocessor Networks ....................................... 202

9.2.3 Chapter 3: Chordal Ring Networks .......................................... 203

9.2.4 Chapter 4: Multitranputer Image Processing ......................... 203

9.2.5 Chapter 5: Mapping Image-Processing Operations Onto Multitransputer Networks ......................................................... 204

9.2.6 Chapter 6: Lighting for the Optical Inspection of Kiwifruit .... 205

9.2.7 Chapter 7: Algorithms for Defect Detection in Kiwifruit ........ 205

9.2.8 Chapter 8: Pyramidal Vision Applied to Automatic Inspection .. 206

9.3 Suggestions for Future Research and Development ................ 206

9.3.1 Transputer Networks .............................................................. 206

9.3.2 Pyramidal Vision .................................................................... 211

9.3.3 Prototype Systems ................................................................. 212

9.4 Epilogue ....................................................................................... 213

Bibliography ..................................................................................... 214

A Set Notation .................................................................................. 236

B Graph Theory ................................................................................ 237

B.0.1 Basic Definitions ..................................................................... 237

B.0.2 Isomorphism ............................................................................ 237

B.0.3 Subgraphs ............................................................................... 238

B.0.4 Walks, Paths and Circuits ...................................................... 238
B.0.5 Trees ................................................................. 238
B.0.6 Cut-Sets ................................................................. 238
B.0.7 Directed Graphs ....................................................... 239

C Statistics .................................................................. 240
C.0.8 Test for Linearity of Regression ................................. 240

D Occam and the Transputer ............................................ 242
D.0.9 Folding Editor .......................................................... 242
D.0.10 Occam ................................................................. 243
D.0.11 Transputer ............................................................ 246

E Transputer Image Processing System ............................... 248
E.1 Host Software .............................................................. 249
E.1.1 Program for the Host ................................................ 249
E.1.2 Main program ........................................................ 259
E.1.3 Input Output Utilities .............................................. 260
E.1.4 File Handling .......................................................... 267
E.1.5 Local Procedures .................................................... 268
E.1.6 Images on the Host .................................................. 270
E.1.7 Distribution of Images Over the Network ..................... 279
E.1.8 Miscellaneous Procedures ....................................... 280
E.1.9 Images for the Network ........................................... 288

E.2 Network Software ..................................................... 292
E.2.1 Program for the Network .......................................... 292
E.2.2 Input-Output for the Link Shell .................................. 294
E.2.3 General (Local) Procedures ..................................... 297
<table>
<thead>
<tr>
<th>E.2.4</th>
<th>Fast Image-Processing Procedures</th>
<th>298</th>
</tr>
</thead>
<tbody>
<tr>
<td>E.2.5</td>
<td>Image Processing Procedures</td>
<td>298</td>
</tr>
<tr>
<td>E.2.6</td>
<td>Special Image-Processing Procedures</td>
<td>302</td>
</tr>
<tr>
<td>E.2.7</td>
<td>Internetwork Procedures</td>
<td>304</td>
</tr>
<tr>
<td>E.2.8</td>
<td>Convex Hull Procedures</td>
<td>305</td>
</tr>
<tr>
<td>E.2.9</td>
<td>Procedures for Loading and Sending Images</td>
<td>309</td>
</tr>
<tr>
<td>E.2.10</td>
<td>Procedures for the Link Shell</td>
<td>310</td>
</tr>
</tbody>
</table>
Abstract

This thesis is concerned with machine vision including its application to the task of detecting surface blemishes and shape defects in kiwifruit at a rate of four fruit per second. Existing machine vision technology is subject to the twin constraints of large data volumes and restricted processing time. Two approaches to this problem are explored: the use of large processing power, and the reduction of the data volume.

The provision of large processing power is achieved through the use of networks containing large numbers of micro-processors. The establishment of a Transputer Image Processing System (TIPS) has provided a test facility for the development of algorithms on a multi-transputer system. In particular, distributed versions of the convex hull algorithm and of algorithms for image translation and rotation have been developed. The establishment of TIPS required the development of a shell to provide protection against deadlock and to provide a satisfactory environment for software development. The network topology is a significant factor in the system performance, and a particular network, the degree four chordal ring network, is proposed as a suitable network for transputer-based systems. The manner in which image processing operations map onto a multi-processor is also investigated.

The alternative approach to a practical machine vision system is to decrease the volume of image data. This can be achieved using pyramidal vision, and the approach explored in this thesis uses a rank-based technique for the formation of each subsequent layer of the pyramid. In particular, a rank of one or two results in darker blemishes being emphasized relative to their surroundings. As a consequence, the volume of image data required to preserve blemish information is very much reduced.

Another aspect of machine vision is lighting, and the problem of determining the optimum form of lighting for blemish detection on kiwifruit is explored. A machine vision system based on a combination of pyramidal vision and multi-transputer networks is proposed.
Preface

A number of papers and conference presentations have arisen as a result of the research outlined in this thesis. My supervisor, Professor R M Hodgson, was co-author for a number of these publications.


- *A Transputer Network Applied To Image Processing*, Proceedings of the 25th N.Z. National Electronics Conference, Christchurch, August 1988. This paper was also presented to


In this thesis bibliographical references are denoted by a number in square brackets (such as [253]). However, some of the Occam code provided in this thesis uses the same notation for array dimensions, and a different type face is used for this purpose (such as \([100]\)).
Chapter 1

Introduction

1.1 Introduction

This thesis is concerned with aspects of the application of machine vision to the inspection and grading of kiwifruit. To place this work in its broader context consideration will be given to the relationship between machine and human vision, recent developments in the automation of the industrial work-place, and the emergence of multiprocessor technologies.

The first aspect of this investigation is vision. Machine vision is related to human vision, although the characteristics of each are very divergent. However, the heuristic nature of the development of image processing in modern industry means that the effectiveness of such systems is judged in anthropocentric terms. Vision occupies a large part of the human central nervous system. The development of vision in animals relates to the need to detect movement, whether prey or predators, possibly at a considerable distance, and the need to be able to concentrate visual attention on specific areas in the field of view while ignoring the great volume of irrelevant visual information. In higher animals such as *homo sapiens* the ability to handle a variety of visual tasks is almost certainly a feature of their ability to survive.

The second aspect is industrialization. The introduction of industrialization in Europe and North America in the eighteenth and nineteenth centuries, commonly referred to as the industrial revolution, was characterised by the augmenting of human labour with mechanical energy. What has sometimes been referred to as the second industrial revolution involves the augmenting of human intelligence with the processing power of computers. A major development following the industrial revolution was the introduction of the production line, symbolised by the conveyor belt. Prior to this development work was largely self-paced, or at least followed a flexible schedule dictated by people rather than machinery. The inexorable flow of items along a production line imposes constraints on workers that are quite unlike those met prior to the industrial revolution.
The third aspect of interest is quality control. Free market economies such as prevail throughout most of the western world are buyer's markets [280] in the sense that the preference of buyers will determine the success of a product. This is in contrast to seller's markets such as the New Zealand new car market of the 1950's and 60's, or some of the centrally planned economies of eastern Europe. Although price probably remains the dominant determinant in buyers' markets, quality is playing an increasing role as purchasers become increasingly discriminating. There have been three important aspects of this. Firstly there has been the quest for the highest possible quality, with customers willing to pay premium prices. Secondly, the trend to product liability, particularly in the United States, means that the occasional faulty product that 'slipped through' can lead to financial disaster, necessitating the detection of any defects to a very high degree of confidence. Thirdly, the development of satellites, medical life-support systems, and air transport has resulted in the need to assure very high standards of reliability. Human visual systems have traditionally been the basis of quality control. Although some simple operations such as the sorting of peas from their shells have long been automated, operations based on visual information are still overwhelmingly the province of human beings. However, human beings are not well suited to watching identical or nearly-identical objects on a conveyor belt for eight or more hours per day.

The final aspect of this study is provided by the development of computers. The first electromechanical computers, developed in the late 1930's and early 1940's, were comparatively slow, with a peak speed of perhaps ten additions per second [155]. The first electronic digital computer was ENIAC, developed between 1942 and 1945 at the University of Pennsylvania. Used primarily for the production of ballistics tables, it consisted of 18,000 vacuum tubes and performed 5,000 additions per second [155]. The subsequent development of integrated circuit technology led to the production of microprocessors [202], with the first commercially successful microprocessor being Intel's eight-bit 8080 introduced in 1974. Expressing processing power in MIPS\(^1\), the 8080 microprocessor worked at approximately 0.05MIPS. Further improvements in VLSI\(^2\) technology has seen the production of sixteen and thirty-two bit microprocessors that rival and in some cases exceed the power of mainframe computers. Examples of 32 bit microprocessors include Intel's 80386, Motorola's 68030, and INMOS's transputer. The processing power of these microprocessors is in the range of two to ten MIPS. The quest for ever-greater processing power has lead to the development of computers employing more than one processor. These multiprocessors are currently at an early stage of development, with a few types being commercially available and with a severe shortage of appropriate software.

These four fields form the basis of machine vision, which is a particular aspect of image processing. It has been claimed [25] that digital image processing originated with work related to the Ranger lunar probes of the 1960s. These were launched by NASA to provide information on the lunar surface in support of the Apollo program. The video cameras on board the space craft relayed analog images that were subsequently converted

---

\(^1\)Millions of Instructions Per Second.

\(^2\)Very Large Scale Integration.
to digital form. Digital processing was used to remove various camera geometric and response distortions [25]. Since the time of the Ranger spacecraft, space probes have sent images back from most of the planets in the solar system, plus those planets’ moons. Earth resources, weather, and military surveillance satellites make extensive use of remote sensing technology in which digital image processing is used to enhance the images and to extract information. These operations have the advantage that they do not need to keep pace with the rate at which images are produced: thus computers can spend hours or even longer processing the images. However, machine vision imposes a major restriction: the vision task must keep pace with the flow of objects, typically the speed of the conveyor. This is commonly called operating in 'real-time', although the same term is often used to mean ‘at the video frame rate’.

Machine vision provides a means of achieving quality control without relying on the vagaries of human vision. It has developed as a consequence of industrialization, of the development of computer technology, and in part to the discrimination of modern consumers. Quality control has been central to the past success of the New Zealand kiwifruit industry. The possibility of harnessing machine vision to quality control has attracted the attention of the kiwifruit industry. The remainder of this Chapter is organized as follows. The kiwifruit industry, which forms the principal target of the research presented in this thesis, is described in Section 1.2. Secondly, some characteristics of machine vision are reviewed in Section 1.3. Finally, a brief outline of the remainder of the thesis is presented in Section 1.4.

1.2 Kiwifruit Industry

The kiwifruit, or Actinidia deliciosa, is indigenous to South-East Asia, occurring as a vigorous fruiting vine on forest margins of localities such as the Yangtze Valley [247]. The fruit, previously called Chinese gooseberry and renamed kiwifruit, has a brown hairy skin and is roughly ellipsoidal in shape. When ripe the flesh is soft and green, and in cross-section has a distinctive pattern of seeds symmetrically arranged about a white centre. In this section the nature of kiwifruit, the types of defects found, and some aspects of harvesting, grading, and packing will be described.

1.2.1 History and Outlook

Seed from China was planted in Wanganui at the beginning of this century, with the first fruit being obtained in 1910 [247]. The first commercial plantings were carried out in the Bay of Plenty in the 1930’s. By 1984 about 14,000 Hectares of kiwifruit plantings were producing approximately 60,000 tonnes of fruit, with an anticipated production exceeding 200,000 tonnes by 1990 [247]. This rapid expansion has produced a problem: labour in the main kiwifruit growing areas such as Te Puke is short for the five or six weeks of the
picking season, and with the crop projected to expand rapidly over the next few years this problem is likely to worsen. Coolstores have too small a capacity to greatly stem the huge flow of fruit that enters packhouses in May and June each year, so the sorting and packing task cannot be spread much beyond the picking season.

1.2.2 New Zealand Kiwifruit Authority

The New Zealand Kiwifruit Authority (NZKA) was established in October 1977 as a statutory body to control, advise, promote, and advance the kiwifruit industry [247]. The promotion of kiwifruit as a quality product, especially in overseas markets, has meant that uniform quality control standards must be maintained throughout the industry. The NZKA has been instrumental in establishing and maintaining these standards. In September 1988 the NZKA was replaced by the interim New Zealand Kiwifruit Marketing Board.

1.2.3 Characteristic Fruit Defects

Selective breeding of kiwifruit has resulted in a number of varieties, with the Hayward variety accounting for more than 98% of commercial plantings because of its good flavour and keeping properties [247]. However the Hayward variety is subject to certain shape deformities, especially the Hayward mark, flats, fans, and dropped shoulder. The NZKA has established criteria for the maximum allowable extent of these defects, as follows [211].

**Hayward Mark** Referring to Figure 1.1, the Hayward mark is a longitudinal mark which may deepen into a groove. Occasionally, however, a hook or protrusion develops, as shown in the right and bottom of Figure 1.1, which is liable to break off during handling, offering a site for fruit deterioration. They result from a stamen sticking to the side of the young fruit at the time of setting, the filament of the stamen causing the groove, and the anther causing the hook [247]. Hayward marks without a hook are acceptable in export fruit, but the presence of a hook means that the fruit must be rejected.

**Flats and Fans** Referring to Figure 1.2, the width of a fruit must not exceed the length of the longitudinal axis; nor may the fruit be too out of round when viewed along the axis. Fruit that are excessively fan-shaped are also rejected. Two reasons may be advanced for regulations concerning flats and fans. Firstly, such fruit will often develop a very hard core. Secondly, the artistic value of kiwifruit as a component of food presentation is part of its vigorous promotion on overseas markets, and the symmetry of circularly symmetric slices is seen as giving such fruit a commercial advantage. Flats and fans arise through distorted flowers, and are more prevalent near the base of the fruiting shoot [247].
Dropped Shoulder  Another shape defect is dropped shoulder, as illustrated in Figure 1.3. The cause of this defect is not known [247]. The official NZKA requirement is
that angles of slope less than $15^\circ$ are acceptable, otherwise the fruit must be rejected. Obviously the literal application of this regulation provides serious difficulties for human sorters.

1.2.4 Fruit Defects Resulting from the Environment

Environmental conditions such as frost, sun, wind, hail, and water stain may result in defective fruit. Large areas of dirt on a fruit, such as illustrated in Figure 1.4 are also unacceptable. The NZKA specifies the restrictions on the extent of the various defects.
Frost Damage Frosted fruit usually has collapsed cells on the shoulder, giving a pinched appearance at the stalk end.

Sun Damage Exposure to the sun can result in two types of damage — weathering and sunburn. Excessive exposure to the sun can produce a crazed pattern of 'skinlines' as illustrated in Figure 1.5. The presence of this crazing makes the fruit unacceptable. Alternatively the fruit may have sunburn, resulting in local discoloration which shows up as dark against the lighter colour of the unburnt skin. This contrast forms the basis of the detection of sunburn, which if noticeable makes the fruit unacceptable.

Wind Rub Fruit damaged by the repeated movement against other objects develop surface blemishes. Fruit damaged early in the season form scar tissue, but later in the season this may result in a dark, sunken area referred to as a 'proximity mark' [247]. The NZKA regulations on skin rub specify that any defect must be light and not more than one square centimetre in area, while the regulations on proximity marks specify that they must be light in colour, less than one square centimetre in area, and not more than two in number. The defect in Figure 1.6 makes the fruit unacceptable.
Hail Damage  Hail damage results in surface scars. The relevant regulations for these scars are as for wind rub.

Water Stain  Water may stain the kiwifruit skin, resulting in dark streaks. Such stains are only acceptable if less than one square centimetre in area.

1.2.5 Pests and Diseases

A large and growing number of pests and diseases affect kiwifruit [213]. A number of these cause specific fruit defects.

Leaf Roller Caterpillar  Although the leaf roller caterpillars feed principally on the leaves they can scar the fruit surface, leaving irregularly shaped blemishes. Any visible insect chewings are unacceptable.
Greedy Scale  This is a very small sap-sucking insect whose presence could result in a complete shipment of kiwifruit being rejected at overseas ports. For this reason it is a matter of serious concern, and fruit showing any evidence of greedy scale must be rejected. The visual evidence is in the form of a very small (perhaps one square millimetre) light-coloured spot on the fruit, generally at the stem end and often under one of the sepals.

Passionvine Hopper  The nymph of the passionvine hopper secretes honeydew which becomes colonised by the sooty mould fungus, creating a black deposit. Any noticeable areas of such a surface deposit make the fruit unacceptable.

Sclerotinia  *Sclerotinia* is an infection that will often result in the loss of the fruit. However in less severe cases the infection heals to leave a prominent scar [247]. Such scars are unacceptable if more that one square centimetre in area, if they are not fully healed, or if the scar is cracked.
1.2.6 Harvesting, Grading, and Packing

Harvesting occurs between the beginning of May and the beginning of July. At harvest time the fruit are hard, but as any damage can lead to ethylene production and premature ripening they must be handled with care. The actual date at which harvesting can commence is based on the amount of soluble solids in the kiwifruit juice, and this must be at least 6.2%. Fruit may be placed directly in a cool store, or may be sorted and graded. Currently this is a very labour-intensive operation, with rollers tumbling the fruit along a sorting table to provide an all-round view. Defective fruit are manually lifted from the table and placed on a rejects conveyor. Defect-free fruit are considered to be of export quality. Good lighting is crucial to this visual inspection, but it is to be observed that the diffuse lighting that is the most suitable for seeing surface blemishes is seldom the optimum for detecting Hayward hooks. The significance of good lighting has been recognised by the NZKA’s Quality Standards Committee, who have suggested that lighting standards may be set for packhouse sorting tables [212]. Currently kiwifruit are graded by weight, although the moulded packing trays are based on size rather than weight. A single tray contains about 3.5 Kg of fruit.
1.3 Characteristics of Machine Vision

In this section the basic functions of machine vision systems will be listed and the requirements such as data volume, speed, accuracy, reliability, and cost will be reviewed.

1.3.1 Basic Functions of Machine Vision

The basic steps involved in machine vision can be listed [195] as

1. Image formation.
2. Image pre-processing.
3. Image analysis.
4. Image interpretation.

Not all steps need be present.

Image formation In current machine vision systems, image formation is usually based on television-compatible vidicon or CCD\(^3\) area cameras or on linescan cameras. A linescan camera captures a single line through an image, and a complete picture is constructed by moving the object relative to the camera. Thus the linescan camera is ideally suited to use in conjunction with conveyor belts. Most camera sensors work in broadly the same way. Light impinging on a photo-sensitive surface in the sensor causes an electrical charge to be generated. The photo-sensitive surface is electrically scanned in a raster fashion to produce an electrical signal whose voltage varies with the integrated intensity of the illumination at a location. The standard video format [25] is generated by adding two sets of signals to the signal from the photo-sensitive surface: horizontal and vertical synchronization pulses specify the beginning of a new raster line and the beginning of a new picture respectively.

The next step involves converting this electrical signal into a pattern of numbers that can be read by a computer. Suppose a picture is to consist of a mesh of 512 rows, each consisting of 512 columns. These 65536 picture elements are called pixels. An electrical circuit called a phase-lock loop is synchronized with the horizontal synchronization pulses,

\(^3\)Charge Coupled Device.
and the resultant time interval is divided by 512 (the number of pixels in each row). For each one of these time interval divisions the video signal is sampled by an analog-to-digital converter whose output is in digital form, usually in the range of 0 to 255. These numbers are stored in a high-speed computer memory called a frame store. The equipment which digitizes the video picture is called a frame grabber. The image in the frame store is in a form suitable for reading and processing by a computer, and consists of an array of numbers in which the size of the number represents the intensity of the light falling on the corresponding point of the photo-sensitive surface in the camera.

**Image pre-processing**  *Noise* in image processing may refer to any undesirable aspects of the image. Incorrect pixel values can result from defects in the camera, such as faulty elements in a CCD camera. Electrical 'noise' in the form of radio-frequency pulses or of spikes on the power supply can result in a random occurrence of pixels having values widely at variance with those of their immediate neighbours. Such pixels can be corrected by filtering. This typically involves substituting a value more representative of the immediate neighbourhood. One of the most effective filters for this purpose is the median filter which involves ranking the pixel values in the neighbourhood of each pixel and choosing the median value to replace the given pixel. Noise removal is the principal form of image pre-processing [25].

**Image analysis**  Since the image is held in a computer memory in the form of an array of numbers, those numbers can be manipulated to provide information on the image. For instance, the number of pixels exceeding some given threshold could be counted to yield a measure of the area of an object. Such a measure is called an *image attribute*. The image is subjected to a sequence of image-processing operations, whose objective is usually to produce a number of image attributes. A sequence of image-processing operations will be referred to as an *algorithm*.

**Image interpretation**  The attributes produced by the image analysis are used as the basis for an appropriate decision. For instance, suppose the image analysis algorithm compared an image of a manufactured item with a stored template of what that item should look like. The attribute passed for image interpretation could be the number of pixels over which the image is different to the template. This attribute could then be used to accept or reject the item.

1.3.2  Requirements

Machine vision is subject to a number of constraints and limitations.
Data volume Commercial frame grabbers typically operate on images of size 256 rows by 256 columns (or \(256^2\) for short), or of size \(512^2\) pixels, with each pixel being represented by one byte. The resolution of the analog-to-digital converter on the framegrabber board may, however, be less than 8 bits. Thus the typical data volume per image on commercial systems is 64KBytes and 256KBytes respectively. An alternative approach to determining data volumes is to consider the end requirement. For instance, mensuration may involve accuracies of 1 in 1000 or 1 in 2000. If measurements are made in a single dimension then, in the simplest case, a linescan camera could be used to capture a single line of an image in the required direction, so that quite modest data volumes would be involved. In the general case of two-dimensional measurements, however, pictures would need to be \(1000^2\) or \(2000^2\) to achieve the required accuracy, thus involving data volumes of possibly 4MBytes. In a machine vision application reported by Cormack et al [75] photographic plates from a Schmidt astronomical camera are scanned by a linescan camera which digitizes the plate as 35,000\(^2\) pixels at fourteen bits per pixel, giving a total data volume of approximately \(2 \times 10^9\) bytes.

Considering the detection of greedy scale on kiwifruit, it will be shown in Chapter 7 that reliable detection of such defects will require that the defect must occupy at least five pixels to avoid being filtered out. Thus one millimetre must correspond to at least three pixels. For a field of view of 10cm by 10cm the data volume is approximately 90KBytes.

Speed Human and animal vision involves object recognition within a few hundred milliseconds. Objects move along production lines at rates of typically one to ten per second. In a system for the quality control of pizza bases, described by Hudson [147], three pizzas are inspected every second. A system for the inspection of bottle caps [147] works at a rate in excess of fifteen per second. According to Murray [203] systems engaged in the inspection of horticultural produce should aim at 10 to 15 tonnes per hour, and in the case of potatoes this implies 40 inspections per second. The target rate for kiwifruit inspection used in this thesis and based on the way in which packhouses currently operate, is four fruit per second.

Accuracy The accuracy or success rate of a machine vision system can be defined as the number of correct decisions as a proportion of all decisions made by the system [187]. In many circumstances the application of machine vision will be designed to err on the side of rejection of any questionable items. One set of such applications is those in which a manufacturer’s reputation is at stake. A second set of such applications involves a manufactured object which, although perhaps small in itself, is a critical component of a larger system, and failure of that one component would endanger the viability of the whole. A third set of applications which is gaining in importance, especially in the United States, is the necessity of ensuring freedom from product liability suits.

The checking of assembled products such as the assembly of camera components [187] will normally involve high accuracies – 99.9% or more. The initial objective in the case
of kiwifruit inspection is a success rate of 75%, with a tendency to err on the side of acceptance of faulty fruit. This is because in the early stages a machine vision system is seen as part of the overall inspection system, with the automatic component resulting in the fruit with gross defects being rejected, and the much smaller volume of defective fruit that remain being manually inspected. Since the return to growers for export quality fruit is much higher than for reject fruit, there is considerable interest in not rejecting acceptable fruit.

**Reliability**  If a production line depends on the reliability of a machine vision system, then a high mean time between failures and a small amount of unplanned downtime will be required. A suitable target for a machine vision system in a kiwifruit packhouse would probably be a downtime (excluding time for maintenance, etc.) not exceeding, say, ten minutes per day. Reliability involves issues of both hardware and software.

**Cost**  The ultimate acceptability of any commercial application of machine vision depends on cost. Hudson [147] cites the cost of typical turnkey systems in the U.S. at $US50,000\(^4\) to $US125,000. A machine vision system for the kiwifruit industry should be economic at up to $50,000\(^5\) per sorting line at a processing rate of four fruit per second, but above $100,000 per line the economics are likely to be marginal at the best.

### 1.4 The Organisation of this Thesis

The broad objective of this thesis may be stated as follows:

To consider a system which will detect surface blemishes and shape defects in kiwifruit with a success rate of at least 75% and at a throughput of four fruit per second.

In this thesis emphasis is placed on the characteristics of systems capable of meeting the high data throughput implied by the fruit inspection rate of four fruit per second. In other words, a large amount of processing must be done in a very small amount of time. There are two approaches to this requirement: use large processing power, or decrease the data volume.

The general structure of the thesis is as shown in Figure 1.7. The twin approaches of large processing power and of data compression are pursued in this thesis. Large processing power is achieved by the use of more than one processor, and this is called *multi-processing*. The work on multi-processing reported in this thesis involves three theoretical studies and

---

\(^4\)In 1984 dollars.

\(^5\)In 1988 NZ dollars.
one Chapter on practical work in this field. Multi-processors communicate with each other over interconnection networks, and the selection of a suitable network is important for the development of an efficient system. A theoretical study of multi-processor networks in general is presented in Chapter 2, and of a particular type of network known as the chordal ring in Chapter 3. Breaking problems up into parts that can be distributed over multi-processor systems is a crucial step in the use of such systems, and a study of this topic is presented in Chapter 5. The practical experience with multi-processing networks consists of the development of a Transputer Image Processing System, or TIPS for short, and this is described in Chapter 4. The alternative approach to achieving the main objective of this thesis, based on data compression, uses a method called pyramidal vision and is described in Chapter 8.
A very important consideration is the design of defect detection algorithms for high-speed systems, and this is pursued in Chapter 7. The provision of suitable lighting is of importance for the successful application of machine vision, and this is investigated in Chapter 6.

The reports and papers that have been prepared as part of this programme are detailed in the preface. A considerable amount of material has been relegated to the appendices. This includes background material such as graph theory, and selected software listings. The thesis is complete without these appendices, and they have been included purely to augment the main body of the text.
Chapter 2

Multiprocessor Systems

2.1 Introduction

A feature of digital image processing is the comparatively large volume of data involved. Taking the automatic inspection of kiwifruit as an example, a blemish must occupy at least five pixels to remain after filtering. Thus for a blemish of size one millimetre in a field ten centimetres in extent, the field must correspond to approximately 300 pixels. A common size of frame grabber is 256 by 256 pixels, and if the analog-to-digital converter has an accuracy of eight bits, the resultant data volume is 64Kbytes (where 1K = 1024). If the image-processing algorithm for blemish detection involves, say, 100 computer instructions per pixel, if each fruit inspected requires three views, and if five fruit are to be processed per second, then the computer processing power required is approximately one hundred million instructions per second (abbreviated as 100 MIPS). Future developments in image processing may involve pictures of 2048 by 2048 pixels (roughly equivalent to a medium quality photograph). If these are processed at video rates (25 frames per second in New Zealand) and if each pixel is operated on by an average of, say, 1000 computer instructions, the required processing power is 100,000 MIPS. In comparison the Digital Equipment Corporation VAX 11-750 minicomputer operates at about one MIPS and the CRAY-1 supercomputer operates at about 400 MIPS. Supercomputers are very expensive (typically in excess of ten million dollars) and are thus inappropriate to machine vision systems.

Many image processing algorithms involve operations that are local (that is, they do not need to refer to pixels outside a small window of typically 3x3 pixels centred on the given pixel). Such algorithms can be performed by segmenting the image and apportioning each segment to a processor. For purely local operations the use of \( n \) processors will divide the total execution time by \( n \) compared to a single processor. The case of non-local algorithms such as the Fourier transform or the convex hull is more complex and speed up factors are likely to be less than \( n \) in such cases. Image processing has long been recognised as a promising application of multi-processing, in which a number of processors operate in
parallel (hence the alternative term 'parallel processing').

An important concept that will be used in this chapter is that of granularity, especially in reference to image processing. A 'grain' is the smallest independent unit in a computation.

The remainder of this chapter is arranged as follows. In section 2.2 bus-based systems are reviewed. One view of a program is as a flow of data, and this forms the basis of dataflow systems reviewed in section 2.3. In section 2.4 multiprocessors based on dynamic or switching networks are reviewed, while section 2.5 reviews multiprocessors based on static networks. These static networks will be further developed in subsequent chapters. The problem of synchronization failure is briefly reviewed in section 2.7. Major software issues such as the problem of deadlock, will be considered in sections 2.6 and 2.8. The mapping problem, which is central to the effective use of multiprocessors, will be explored in Chapter 5, using the multiprocessor system described in Chapter 4.

2.2 Bus-Based Systems

A number of approaches have been explored for multi-processing systems. The first, based on traditional single-processor computers, is based on shared memory accessed through common data and address buses. A number of such systems have been constructed. An example of such a system is described by MacKenzie [180] who concludes that the performance of five processors on a single bus is equivalent to three independent processors. The performance of a multi-processor system based on a single bus is thus very restricted. A major consideration in shared memory systems is the use of common or global variables. If one processor wishes to access a variable while another processor is in the course of changing that variable an incorrect value may be accessed. Thus some form of locking is necessary to protect variables that are being updated. Another consideration involves the sharing of a bus. If, as is likely to be the case in any practical system, the number of buses is less than the number of processors, at some stages more than one processor will wish to use a single bus at the same time. If the number of buses is significantly less than the number of processors this bus contention can seriously degrade performance.

Another bus-based multiprocessor system is described by Finkel and Solomon [103]. Their interconnection strategy is based on a system of buses, each bus having a small number of processors. Lang et al [170] discuss the general case of multiple buses, with a performance intermediate between that of a single bus and that of a full crossbar in which every processor can be directly connected to any memory module. Das and Bhuyan [79] derive expressions for the bandwidth of a multiple bus multiprocessor system, and generalize their results to cover partial bus architectures. Marson et al [182] have modelled bus contention in a multiprocessor system and has compared the model with measurements on an actual prototype. Mudge and Al-Sadoun [199] have modelled bandwidth and other performance measures as functions of the number of buses in a multiple-bus system. Retthberg and Thomas [236] conclude that memory contention can be controlled by balancing
a shared-memory multiprocessing system through hardware or software. They suggest the use of fine-grain memory interleaving for common-bus architectures. However their results are only applicable to a limited class of problems. Towsley [281] has presented two models of multiple bus multiprocessing systems, one involving ‘flow equations’ and the other involving ‘surrogate delays’.

Memory access contention in bus-based architectures has been discussed by Davidson [80] who produces a model of memory access conflict, and by Yew et al [302] who state that even a small percentage of ‘hot spots’ involving memory access contention can seriously degrade performance. Mudge et al [200] have analysed multiple bus systems which they see as cheaper than full crossbars but possessing some fault-tolerance as compared to single buses. Fault tolerance in bus-based architectures has also been examined by Pradhan [224].

Bus-based multi-processing systems have the advantage of being an extension of established uniprocessors. Limitations such as restricted bus-bandwidth and memory contention will count against it, especially in image-processing where high data-rates are needed and distributed processing can be easily implemented.

2.3 Dataflow Systems

One approach that promises to avoid the problems encountered with bus-based systems is that of dataflow. The dataflow model of computation is data driven: each instruction is enabled when each of the required operands is supplied (see Figure 2.1). In the dataflow model a dataflow program is viewed as a graph made of actors interconnected by arcs which carry tokens. An actor is enabled when all its input arcs carry tokens [113]. Dataflow systems have been explored by many groups, and a number of dataflow computers have been constructed, including MIT, Manchester, DFSP, Sigma-1, ImPP, Hughes, and EM-3. Dataflow hardware is described by Agerwala and Arvind [3], Dennis [83,84], Dennis and Musunas [85], Gaudiot [112], Gurd [124], Hartimo et al [128], Hiraki et al [133], Patnaik et al [218], Requa [237], Srin [265], Temma et al [279], Vedder et al [286], Yamanguchi et al [300], and Watson and Gurd [291]. A microprocessor, the \( \mu \)PD7281, that employs dataflow principles has been developed by NEC [158].

Dataflow programming is described by Ackerman [1], Agerwala and Arvind [3], Arvind and Gostelow [16], Bic [35], Carlson and Hwang [62], Davis and Keller [81], Faustini and Lewis [98], Hasegawa and Amanniya [129], Jayaraman and Keller [157], Kubo et al [165], and Reed and Patrick [230].

Unfortunately the early promise of the dataflow model of computation has not been met. A major problem is in the handling of structures such as arrays. Gaudiot [112] concludes that there is no easy solution to the problem of dealing with data structures on dataflow machines. Efficiency may be improved by using variable resolution actors
as proposed by Gaudiot and Ercegovac [113]. When applied to image processing, if an
$n$ by $m$ image is considered as a unit and is sent as a sequence of $n \times m$ bytes it is
possible to devise a dataflow mechanism to perform all point operations and, with some
complications, all local-neighbourhood operations. The latter procedure is analogous to
the use of systolic arrays for performing, for instance, rank filtering on an image [208].
Global image operations, however, do not fit so conveniently into the fine-grain, operation-
level concurrency of dataflow machines. For general array handling each token must
include not only the value of the element but also its location in the array. Furthermore, if
iterative operations are performed on an array, the tokens must also include the iterations
number. Each step towards making dataflow computers more general in their application
has resulted in added complexity, so that a single byte pixel value may result in a token
that is, for instance, 96 bits wide [291]. Gajski et al [110] have reviewed some of the

Figure 2.1: Dataflow Model of Computation.
2.4 Multiprocessors Based on Dynamic Interprocessor Links

The performance of multiprocessors based on shared buses deteriorates markedly with increasing numbers of processors, while dataflow computers require a large overhead to be able to handle structured data. An alternative scheme involves dedicated inter-processor communications links. Such multiprocessors might share memory via common buses but are most efficiently implemented using local memory. Ideally every processor would be directly connected to every other processor, but the number of interprocessor links per processor increases as the number of processors in the network. Alternatively a full crossbar could be implemented, based on one link per processor and the switches controlled either from the individual processors or from a supervising processor. The number of switches increases with the square of the number of processors and the system rapidly becomes unacceptably complex.

Research into switching (dynamic) networks has concentrated on schemes that offer a compromise between the power of a full crossbar switch network and more restrictive networks that are not as complex. An important concept in reconfigurable networks is that of blocking. A non-blocking network can satisfy any new, valid interconnection request in the sense of providing a path for that request without disturbing any existing interconnections [183]. A good review of switching networks is given by Masson, Gingher, and Nakamura [183]. Networks that have been studied in relation to multiprocessors include the Benes network [27], the Omega network [171], the Clos network [28,72], the Cantor network [60], the perfect shuffle network [267], and the Banyan network [119]. These networks have been analysed in regard to fault tolerance [4,100] and appropriate control strategies have been developed [11,175]. Dynamic links involve a significant overhead for control, and this has been analysed by Oruc and Oruc [215].

2.5 Multiprocessors Based On Static Interprocessor Links

A static interconnection network could be thought of as a dynamic network which is fixed at compile time instead of dynamically. Ideally the actual network topology can be chosen appropriate to the application and, once chosen, involves neither switching tokens nor a supervisor. Such a system can involve processors operating asynchronously, with synchronisation of processes occurring through the interprocessor links. The problem of achieving synchronization of asynchronous processes has received considerable attention and is central to this model of parallel computation.

In designing a parallel processing system, the choice of an appropriate interconnection topology will involve decisions on the mode of operation, the control strategy, the switch-
ing methodology (if appropriate), and the network topology [101]. Static networks have
the advantage of being simpler than dynamic networks through dispensing with switches,
but some flexibility is lost as a consequence. Amongst static networks that have been
studied are the ring [229], the binary tree [127], the chordal ring [14,186], the hyper­
cube [32,87,130,245,272], the incomplete hypercube [162], the cube connected cycles [226],
the hypertree [121], and cellular arrays [227,232,242,270]. These networks have been anal­
ysed with respect to message-passing efficiency [6,15,77,103,177,295] and to the problems
involved in mapping algorithms onto the processor network [2,5,12,82,116,148,167,173,178,
196,259].

Fault-tolerance has received considerable attention in regard to the communications as­
pcts of static interconnection networks [33,174,223]. A typical requirement is that in the
presence of faults the performance of the system should degrade gracefully. If a single
fault arises it should not disable the system. A significant factor is the minimum number
of network breaks which would isolate part of the network from the remainder of the
network.

The acceptability of multiprocessing systems as computing resources will depend to some
extent on the ease with which algorithms can be implemented. In the case of image
processing this requirement means that the locations of processing nodes in the network
should be simply related to the locations of the corresponding segments in the image.
A simple mapping such as shown in Figure 2.2 is not necessarily the most efficient. For

![Figure 2.2: A Simple Mapping for a Multi-Processor Array.](image)

instance, Miller and Stout [196] use snake-like ordering on a mesh-connected computer.
Hudak [148] presents a programming methodology which isolates the programmer from
the computer architecture through the use of mapped expressions. However the overheads
implicit in this approach are likely to make it unattractive in areas such as real-time
image processing. Agrawal and Jain [5] partition each serial algorithm into several non-
interactive independent subtasks so that parallelism can be used within each subtask. In the architecture which they describe, the avoidance of interprocessor communication is a major issue. Deminet [82] shows the importance of algorithm structure by giving as an example the implementation of two versions of the FFT\textsuperscript{1} algorithm. The version in which the data are ordered so as to avoid the need for tight interprocess synchronization runs three to four times as fast as an alternative version. Lint and Agerwala [178] emphasize the importance of interprocessor communications in algorithm design.

A detailed analysis of a particular static interconnection network, the chordal ring network, is presented in Chapter 3.

2.6 Multiprocessor Software

The problem of establishing a language to make use of the parallelism inherent in a multiprocessing network has been examined by a number of researchers. Dijkstra [89] has established a procedure for using 'guarded' commands to handle non-deterministic program components for which at least the activity evoked, but possibly even the final state, is not necessarily determined by the initial state. A guarded command is a statement list prefixed by a Boolean expression; only when the boolean is true is the statement list able to be executed. Hoare [136], with his work on communicating sequential processes, has extended Dijkstra's guarded commands to include guarded input commands as part of a solution to the problem of synchronization in multiprocessor systems. Hoare introduced a 'PARALLEL' construct for concurrent processes, and proposed an input/output protocol for communication between concurrent processes in which communication takes place when one process names another as destination and the second names the first as source. He proposed that Dijkstra's guarded commands be the sole means of introducing and controlling the non-determinism that is a natural feature of an asynchronous multiprocessor.

The language that Hoare developed, called CSP (Communicating Sequential Processes), allowed input commands to appear in guards. Subsequently, Silberschatz [259] has studied the results of allowing output commands to appear in guards. Hoare's system is also the basis of Occam which is used with the INMOS transputer (see Chapter 4).

Hoare [135] has established a procedure based on the use of sets of axioms and rules of inference which can be used in proofs of the properties of computer programs. Owicki and Gries [217] have used Hoare's deductive system as the basis of a language for parallel programming. Their work is based on that of Cadiou and Levy [57] who apply Scott's mathematical semantics. Owicki and Gries define a statement as being blocked if it has not terminated but no progress in its execution is possible because it is delayed until some condition is true. This leads to the definition of deadlock which is the consequence if a complete program is blocked (see section 2.8). Owicki and Gries's deductive system is used to prove 'partial correctness' (the correctness of an algorithm subject to various

\textsuperscript{1}Fast Fourier Transform
assumptions), freedom from deadlock, and termination.

2.7 Synchronization Failure

Mead and Conway [189] have pointed out that cross-coupled circuits such as flip-flops can be in a state of unstable equilibrium which they call a metastable condition. If the logic threshold voltage of the transistors in a circuit such as that shown in Figure 2.3 is $V_{inv}$, then metastability can be defined as an output voltage from a circuit in a range around $V_{inv}$ that cannot be interpreted as either logic high or logic low. The significance of this in distributed processing is that although individual processors are synchronized, many schemes, including that based on the transputer and described in Chapter 4, involve processors operating asynchronously with respect to each other. Some scheme is needed for synchronization, which will occur when messages are passed. At the hardware level this will probably involve the clocking of an asynchronous signal from the inter-processor link. As shown in [189] the time required for clocking a storage element to get out of a metastable state is unbounded. Mead and Conway state that they suspect that synchronization of an input signal to a free-running clock cannot be accomplished with perfect reliability with finite circuits.

A model for metastability has been developed. It is based on a Poisson distribution and has been verified experimentally. Synchronization failure for a 1MHz signal using NMOS technology may involve a MTBF (Mean Time Between Failures) in the vicinity of $10^6$ seconds (approximately one day). One possible solution is to cascade the signal synchronizers which, in NMOS will increase the MTBF by about $10^7$ for each stage.
The significance for the designer and user of a distributed processing system is that fault-tolerance must be designed into the system if it is to possess any acceptable degree of robustness. This would be necessary even in a research establishment, and would be vital in an industrial application.

2.8 Deadlock

Deadlock occurs when processes become permanently stopped because appropriate resources are not available. Deadlock has been studied with respect to telecommunications networks and loosely coupled computer networks such as database systems in addition to the tightly-coupled systems that are the subject of this chapter. Various surveys of deadlock have been made, including a very readable overview by Isloor and Marsland [154], a general bibliography by Zobel [307], and a survey of distributed deadlock detection algorithms by Elmargarmid [96].

Figure 2.4 is a model of a processor $P$ having a memory of capacity $M$ and input, output channels with capacities $I$ and $O$ respectively. This model will be used to illustrate some types of deadlock. The simplest deadlock might be referred to as a 'logical deadlock' and takes the form

$A$ waiting for $B$

$B$ waiting for $A$

where $A$ and $B$ are processes. This might occur through the incorrect formulation of a problem.
Günther [122] analysed ways in which deadlock could be prevented. His classification of deadlock included direct store-and-forward deadlock in which two adjacent nodes are full and are waiting for transmission to each other. A similar form of deadlock is indirect store-and-forward deadlock which could occur, for example, in a circular network in which each node is full and is waiting to send to the next node. Reassembly deadlock can occur when large messages are broken into smaller packets for transmission and the receiver is unable to reassemble any one complete message. Direct store-and-forward deadlock is illustrated in Figure 2.5, where $P_0$ and $P_1$ are two processes, and it is supposed that $P_0$ cannot remove data from the full input buffer until some of $M_0$'s contents have been output, and similarly for $P_1$. Indirect store-and-forward deadlock is illustrated in Figure 2.6, where $P_2$ is waiting for $P_1$ before sending a message to $P_0$, and so on. Reassembly deadlock is illustrated in Figure 2.7, where deadlock will occur if $M_0 > (O_0 + I_1 + M_1)$.

A classification scheme, adopted from schemes used by Isloor and Marsland [154], and by Elmagarmid [96], divides the topic into detection, prevention, avoidance, and resolution. These topics will be considered with reference to distributed control. Distributed deadlock detection can take place with a central controller or can be fully distributed. Detection would be used in conjunction with some form of resolution. For deadlock prevention all required resources for a process must be declared at the system design stage or at compile time. This makes deadlock prevention a very conservative approach, with an overall undercommitment of resources. Deadlock avoidance is a course of action midway between the other two. Whereas detection allows deadlocks to occur and prevention ensures that deadlocks can never occur, avoidance allows processes to proceed only if the required resources are available.

The next four sections will review the literature on detection, prevention, avoidance, and
Figure 2.6: Example of Indirect Store-and-Forward Deadlock.

Figure 2.7: Example of Reassembly Deadlock.

resolution respectively.
2.8.1 Deadlock Detection

Distributed deadlock detection algorithms normally establish a global view, but some approaches are based on purely local views [96]. The commonest global scheme is based on the construction of a wait-for-graph or WFG for short. Menasce and Munzt [191] introduced the WFG for the detection of deadlock in distributed databases. In the WFG the nodes are processes. A directed edge from node A to node B indicates that process A is blocked and is waiting for B to release a resource. A cycle in the WFG indicates deadlock. As an example of a WFG consider Figure 2.8, which is adapted from [191].

Let the $P_i$ be processes running on two processors $S_1$ and $S_2$. Processes $P_3$ and $P_4$ in the example are divided between $S_1$ and $S_2$ so inter-processor communication is involved. The Wait-For-Graph is a directed graph such that:

1. An edge from $(P_i, S)$ to $(P_j, S)$ means that an invocation of $P_i$ at $S$ is blocked and is waiting for the invocation of $P_j$ to release a resource needed by $P_i$. $(P_i, S)$ is in a state of resource wait for $(P_j, S)$.

2. An edge from $(P, S_i)$ to $(P, S_j)$ means that an invocation of $P$ at $S_i$ is blocked and is waiting for a message from the invocation of $P$ at $S_j$. $(P, S_i)$ is in a state of message wait for $(P, S_j)$. In Figure 2.8 the cycle in $S_2$ is a local deadlock while the $(P_3/P_4)$ cycle is a global deadlock.

In general, deadlock detection consists of building and maintaining the WFG and searching for cycles in it. The WFG thus represents a 'snapshot' of the system. However Ho and

---

$^2$See Appendix B for a review of graph-theoretic terminology.
Ramamoorthy [134] have pointed out that because of the inherent communication delays it is not easy to construct a consistent graph. The WFG is established by transferring messages throughout the network.

Obermack [214] makes the assumption that all messages transmitted between processors in the network are received, and uses an agreed-upon interval between deadlock detection iterations. The algorithm involves the potential for false detection. Because of this problem of false detection and the difficulty of maintaining a consistent WFG, Ho and Ramamoorthy [134] propose more complex detection protocols. In particular, in a two-phase protocol a single processor is periodically chosen as network controller. This site takes control of the network and sends messages to all processors requesting their status and uses this information to create the WFG. If the graph has a directed cycle a second message is sent to all sites requesting their status which, by this time, may have changed. If the WFG constructed from this new information has a cycle then the network is deadlocked. This protocol incurs a heavy overhead, with $4N$ messages per iteration for a network of $N$ processors. Dispensing with any of these messages may result in the detection of false or ‘phantom’ deadlocks [96]. Ahuja [7,8] has developed algorithms for the analysis of network snapshots for classes of store-and-forward networks. Chandy and Lamport [67] have used distributed snapshots to determine the global state of a distributed system. Their scheme does not require synchronised clocks, and is based on a node ‘colouring’ scheme to delineate the instant of the snapshot. There are two phases to this scheme. In the first phase the local status is compiled at each process in the system. In the second phase each process’s status is disseminated in the network and is collected into a snapshot by the process that initiated the snapshot. Spezialetti and Kearns [262] have modified the algorithm of Chandy and Lamport to make more efficient use of the information gathered in the first phase. Their algorithm would appear to be more efficient in cases in which the snapshot was initiated by several processes.

Sinha and Natarajan [260] propose a distributed deadlock detection algorithm based on priorities of transactions, and examine the cost of their algorithm in terms of communication cost, delay, and storage cost. Cidon et al [70] propose a global distributed deadlock detection algorithm for store-and-forward networks. The algorithm is based on a fixed number of buffers per node and a fixed number of buffers per adjacent link. Although the algorithm is simpler than many others it requires network-wide coordination. The algorithm is based on the assumption that there is a spanning-tree rooted at a ‘leader’ in the network. A spanning tree of a graph $G$ is a subgraph that contains all the nodes of $G$ and possesses no circuits. Distributed algorithms exist for determining leaders and spanning trees [70]. Each node holds and updates several status variables, and these are included in messages sent whenever the algorithm is triggered.

According to Awerbuch and Micali [18] most approaches to the problem of deadlock detection had been based primarily on static analysis. They state that
the dynamic nature of the problem makes it so awkward to conceive, that some quite intuitive approaches lead to incorrect algorithms.

For instance a circuit in a directed graph may not correspond to deadlock in the dynamic sense. The deadlock detection protocol algorithm adopted by Awerbuch and Micali is as follows. Firstly, the dynamic problem is reduced to a static partial-input problem. This is then reduced to a static full-input problem. Finally this is resolved by aborting a minimal set of actions. The problem of minimising the number of aborts is NP-complete.

Cidon et al [71] start by developing an algorithm to detect all deadlocked nodes in a static network. A deadlock exists if all transit buffers in a set of nodes \( \{T\} \) are full and there are no directed links out of \( \{T\} \). Cidon et al then develop an algorithm for a dynamic environment. This algorithm detects all nodes that caused the deadlock, and is triggered only when there is a potential for deadlock. Only those nodes which are potentially deadlocked perform the algorithm, and the algorithm does not affect other processes.

Bracha and Toueg [42] present an algorithm for distributed deadlock detection in systems in which processes request \( N \) resources from a pool of size \( M \). This is a generalization of the AND-OR request model in which an OR request is a request that is satisfied by any one of the processes it is waiting for, and an AND request is a request that is satisfied only by all of the processes it is waiting for. According to Bracha and Toueg, some of the existing deadlock detection algorithms have been shown to be incorrect or to be too complicated to be worth proving correct. Their algorithm is developed first for a static system with instantaneous message transmission. Then a static system with transmission delays is modelled with a 'coloured' WFG. Finally the algorithm is extended to a dynamic system whose state can change during the execution of the algorithm. An algorithm for the \( N \)-out-of-\( M \) problem is also proposed by Ryu et al [244]. The method involves maintaining identical WFGs at sites of replicated resources.

Spirakis [264] has published a theoretical technique for examining the parallel complexity of deadlock detection. Its relevance to large-scale systems is likely to be very limited. In his review of 1986, Elmagarmid [96] stated:

More research in the specification, verification, and performance evaluation of deadlock detection algorithms is needed. The lack of unified means by which researchers may specify their algorithms resulted in most of these algorithms being shown incorrect.

In considering deadlock detection algorithms the following must be considered.

- Does the algorithm detect all possible deadlocks?
- Does the algorithm detect any 'phantom' deadlocks?
- What is the time cost or overhead of the algorithm?
• What is the storage cost of the algorithm?

• If a deadlock is discovered will the system recover as gracefully as possible [154]?

• Is it acceptable for certain resources to be held idle by blocked processes for long periods of time until an actual deadlock occurs and is resolved [154]?

• Since many algorithms are network- or application-specific, is the algorithm appropriate in a given circumstance?

2.8.2 Deadlock Prevention

Gelernter [114] has developed an algorithm based on directed acyclic graphs. Data may be lost in this scheme, and it would be up to the receiver to provide acknowledgement and the source to resend if an acknowledgement was not received. Zedan [304] has provided the necessary and sufficient conditions for proof of absence of deadlock in communicating processes. Zedan uses Occam\(^3\) to describe the message communication mechanism. In the method developed by Dally and Seitz [78] any cycles in a network’s WFG are removed by splitting physical channels into groups of virtual channels. Figure 2.9 illustrates an example of the application of virtual channels, and is adapted from an example presented by Dally and Seitz. An interconnection graph, designated by \((I)\), is a directed graph in which nodes represent processors and edges represent channels. A routing function is a function which maps the current channel to the next channel. A channel dependency graph is designated by \((D)\) and is obtained from the interconnection graph. It is a directed graph in which nodes represent the channels of \((I)\) and the edges represent pairs of channels connected by the routing function. A cycle in \((D)\) implies a deadlock. Figure 2.9(b) is the channel dependency graph derived from Figure 2.9(a). The cycle \(C_0/C_1/C_2\) implies that the system will deadlock. The technique of Dally and Seitz is to split each channel in a cycle into two virtual channels:

\[ C_i \rightarrow C_{1i} \text{ and } C_{0i} \]

Order the channels on the basis of the subscripts so that

\[ C_{12} > C_{11} > C_{10} > C_{02} > C_{01} > C_{00} \]

and so on. The mapping rule is that \(C_a \rightarrow C_b\) in \(D\) only if \(a > b\). Then no cycles are possible in \(D\) if the lowest channel, namely \(C_{00}\), is not used. In Figure 2.9(d) the channel dependency graph is shown for the virtual channel scheme illustrated in Figure 2.9(c). The virtual channels share the same physical channels but have their own buffers. Dally and Seitz illustrate their method for three different network topologies.

A multi-processing system to be described in Chapter 4 uses deadlock prevention by allocating buffers that are chosen to be sufficiently large that within the operating range of

---

\(^3\)See Appendix D.
the software deadlock never occurs. The restrictions include such factors as the maximum image-packet size and the number of such packets that can be sourced at any one time. The buffer sizes are initially made very large and are subsequently steadily reduced until deadlock occurs, thus establishing the minimum buffer size.

2.8.3 Deadlock Avoidance

Various versions of the 'banker's algorithm' are commonly incorporated in deadlock avoidance schemes [154]. The basic banker's algorithm requires processes to declare their total resource requirements before they commence. Any process whose remaining needs exceed the available resources cannot proceed. If a demand for a resource can never be met then that process is permanently stopped from running, but presumably this would be
indicative of a poorly designed system.

Merlin and Schweitzer [193,194] use buffer graphs to avoid store-and-forward deadlock as well as a number of other potential deadlock types. A buffer graph is a directed graph whose nodes are a subset of the network buffers, and whose directed edges, connecting some pairs of nodes, indicate the permitted message flow. In buffer graphs there is at least one directed path corresponding to each route in the network. Furthermore buffer graphs contain no directed loops. This approach restricts the full use of the network. Various schemes for constructing buffer graphs are described by Merlin and Schweitzer. Bossomaier and Heath [41] describe a processing shell that provides buffering to decrease the incidence of deadlock. If deadlock should occur the shell provides a facility for flushing blocked buffers.

2.8.4 Deadlock Resolution

Deadlock detection algorithms will run repeatedly with some suitable time between iterations, and following deadlock detection various schemes exist for deadlock resolution. In a scheme published by Sinha and Natarajan [260] a central controller, upon detecting a deadlock, sends an abort signal to the lowest priority process. The aborted process releases all resources, and the central controller reallocates the resources that have been requested by other processes. Deadlocks are detected by circulating a message called a probe, and each processor maintains a queue of probes. In the scheme developed by Cidon et al [70] whenever a node discovers that it is deadlocked it generates a deadlock message that is forwarded through the tree to the leader.

2.9 Conclusions

The processing requirements for real-time image-processing cannot be met by current microprocessors, and a number of alternatives exist. One of these, based on the idea of data flowing through a processing system, is conceptually attractive but has proven difficult to implement. A more promising approach is based on the use of multiple processors. An extension of traditional bus-based computers provides easy access to shared variables but suffers from bus and memory contention problems when the number of processing units on a bus becomes large. The alternative of network-based multi-processors takes two forms — those in which inter-processor links can be switched and those in which the links are fixed. There is thus a tradeoff between shorter access paths, and greater hardware and software complexity.

A major problem in multiprocessor networks is that of deadlock, and a great deal of research has gone into deadlock detection, prevention, and avoidance. The easiest but least efficient way of dealing with deadlock is to allocate sufficient resources in terms of buffers and so on to ensure that within specified constraints deadlock should never
occur. A more efficient approach is to use deadlock avoidance in which processes do not proceed until all resources are available. The most potentially efficient is deadlock detection and resolution, but this problem has resulted in a large number of incorrect algorithms. The multi-transputer described in Chapter 4 uses the simplest approach, namely deadlock prevention. This choice was made because I felt that it would provide an operational image-processing system much sooner than either deadlock avoidance or deadlock detection and resolution.
Chapter 3

Symmetric degree Four Chordal Ring Networks

3.1 Introduction

In the previous chapter static multi-processor interconnection networks were reviewed. Such networks are of particular significance for image processing because images can be conveniently segmented, with the segments being distributed throughout a network of processors. In this chapter techniques for analysing a particular interconnection network, the symmetric chordal ring network of degree four, are presented. Expressions for the network diameter (the maximum distance a message must travel between any pair of processors) and the mean inter-processor distance are derived. The network incorporates the maximum number of processors for a given diameter, and has a communications cost, measured either as network diameter or as the mean internode distance, of $O(\sqrt{N})$. Possible modifications to this network include breaking the ring to provide communications with a host computer, and a non-optimal chordal displacement to allow a cellular array to be mapped onto the chordal ring. Such networks provide a practical means of networking processors such as the transputer, and the results presented in this Chapter have been verified on a network of transputers.

The work described in this Chapter originated from a study of the way in which the INMOS transputer, which will be described in greater detail in Chapter 4 and Appendix D, could be best applied to image processing. A feature of the transputer is the bi-directional inter-processor links that allow a transputer to directly communicate with up to four other transputers, thus permitting easy networking. In designing a parallel processing system, the choice of an appropriate interconnection topology will involve decisions on the mode of operation, the control strategy, the switching methodology, and the network topology. Static networks have the advantage of being simpler than dynamic networks through dispensing with switches, but some flexibility is lost as a consequence.
3.2 Multiprocessors Based On Static Interprocessor Links

Static interconnection networks have been reviewed in Chapter 2, in which static interconnection networks were presented as dynamic networks which are fixed at compile time instead of dynamically. Such systems can involve processors operating asynchronously, with synchronisation of processes occurring through the interprocessor links. A number of network topologies have been analysed in the literature.

The transputer is well-suited to multi-processing systems designed for image processing. For the present discussion it is sufficient to state that in addition to the customary bus interface the transputer possesses four bidirectional interprocessor links. Thus a row of transputers based on nearest-neighbour connections will have a number of unused links. The ends of the row can be joined to form a ring, and remaining links can be interconnected to produce a variety of topologies. Of particular interest are those interconnection patterns that are symmetric, as it should be easier to map algorithms onto such networks, and be easier to perform routing within them.

Local image-processing operators can be catered for by providing links between adjacent processors, forming a row. By connecting the processors at each end a ring network is formed. Define the inter-processor distance \( d(i, j) \) as the minimum number of links which must be traversed to interconnect processors \( i \) and \( j \). Since global operators involve passing data with a frequency which is constant over the whole sequence of local memories, the total time will be minimized if the mean distance is minimized. If each processor can proceed asynchronously the mean distance will be a sufficient criterion, but if the algorithm must be synchronized at intermediate points (for instance, for the passing of computed values between processors for subsequent processing) the diameter (the maximum inter-processor distance) may be of significance.

If an image processing system is being used in a real-time application (for instance, in an automated inspection system) the total processing time available per picture is dictated by the application. If through external changes the rate at which pictures are presented to the system is increased, or if changes in the algorithms involve an increased amount of processing, the system architecture may need to be extended to incorporate added processors. If this can be done incrementally and without a major redesign the architecture will be more acceptable as the basis for practical systems.

For a system which is to handle both local and global image processing operators the criteria for selecting an optimum interconnection strategy may be summarised as follows:

1. Minimize the network diameter.
2. Minimize the mean interprocessor distance.
3. The system should be incrementally extensible.
4. Any single faults should not lead to a complete failure of the system.
5. Algorithms should be able to be implemented as simply as possible.

The remainder of this Chapter is organised as follows. The next section reviews general chordal ring networks. In section 3.4 the characteristics of a symmetric chordal ring network, including the network diameter, the mean inter-processor distance, and the number of alternative paths of minimal length, are derived. A special class of network is introduced in section 3.5, and some of its properties are derived. In section 3.6 the results from the study of the special class of network are applied to networks of any size. In section 3.7 two major modifications to the 'ideal' chordal ring networks are investigated, and in section 3.8 the performance of an actual network is compared with that predicted by the theoretical investigations.

3.3 Chordal Ring Networks

Chordal ring networks of degree three were first proposed by Arden and Lee [14]. Doty [91] has presented a generalization of the chordal ring network of Arden and Lee. However, for larger systems the resultant networks lack the simplicity of those of Arden and Lee. As Akers and Krishnamurthy [9] point out, solutions to the \((d, k)\) graph problem \(^1\) often ignore factors such as symmetry, ease of routing, and the structure of the graph.

McKeown [186] has investigated chordal ring networks of fixed chord length (independent of the number of processors). The resultant network is efficient in regard to local communication and has useful fault-tolerance, but a large system based on this topology would be very inefficient if processing involved global references as the network diameter is of \(O(n)\).

Arden and Lee [15] have investigated the properties of multtree structured (MTS) graphs. When drawn in circular form these graphs are seen to be related to chordal rings. Arden and Lee concentrated on MTS graphs of degree three, establishing bounds on the diameter. The mapping from an image space onto the MTS graph nodes is unlikely to be simple.

The remainder of this Chapter will deal with chordal ring networks of degree four in which each node has two circumferential and two chordal links.

3.4 Analysis of Chordal Ring Networks

Consider a ring of \(N\) nodes (processors). These nodes will be referred to as nodes 0, 1, ..., \((N - 1)\), and for convenience these will be assumed to be in ascending order when the ring is traversed in a clockwise direction. An equivalent numbering system exists obtained by traversing the ring in the reverse direction, namely -1, -2, ... so that -1 is the same as \((N - 1)\). Each node is connected to its two nearest neighbours; thus node \(i\) is connected

\(^1\)Given a maximum degree \(d\) and a maximum distance \(k\), find the graph with the maximum number of nodes [190].
to nodes \((i - 1)\) and \((i + 1)\), or more briefly nodes \((i \pm 1)\). In particular node 0 is connected to nodes 1 and \((N - 1)\).

In addition to the ring connections each node will have two auxiliary (or chordal) connections to other nodes. If a bi-directional link connects nodes \(i\) and \(j\), then the existence of the link is completely described if \(i\) is said to be connected to \(j\) or if \(j\) is said to be connected to \(i\). For convenience, each link is associated with just one node. Thus in the case in which each node has two auxiliary links (that is, a total of four links, two being devoted to the ring), only one destination needs to be specified. Because of symmetry all nodes are equivalent and the network can be analysed in terms of the communication between node 0 and the other nodes in the network.

Let the chordal links of the network have a constant chord length (or displacement) = \(d\). Then \(i = (p_id + q_i) \mod N\) \((p_i\text{ and } q_i\text{ integers})\) generates all the nodes \(i \in [0, N - 1]\).

The number of chordal transfers is \(p_i\) and the number of circumferential transfers is \(q_i\).

The number of full rotations involved in going from 0 to \(i = (p_id + q_i) \mod N\) is

\[
r = (p_id + q_i) \div N
\]

where \(\div\) represents integer division.

The allowed transitions are such that either \(p_i\) or \(q_i\) (but not both) can change by one, corresponding to the inter-processor links. For example, \(3d + 4\) can go to any one of \(3d + 3\), \(3d + 5\), \(2d + 4\), or \(4d + 4\).

Definition: The length \(L_j\) of an internode link or connection from node 0 to node \(j\) is the number of links needed. Thus \(L_j = p_j + q_j\).

The method to be adopted in this Chapter is through the formation of a list of chordal transfers. This technique will be used in section 3.5 for the analysis of optimal networks. By listing the lengths of forward or clockwise and reverse or anti-clockwise chordal transfers the values of all the intervening distances can be deduced by the principle that the difference in distance between adjacent nodes cannot exceed one. For instance, given a system of \(N = 13\) nodes and a chord length (displacement) of \(d = 5\), the following table of distances from node 0 is derived from the result of two clockwise (to nodes 5 and 10) and two anti-clockwise (to nodes 13-5 = 8 and 13-10 = 3) chordal transfers:

<table>
<thead>
<tr>
<th>Node:</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distance:</td>
<td>0</td>
<td>-</td>
<td>2</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

and the list of chordal transfers will consist of

\[
\{(C_j, L_j)\} = \{(0,0),(3,2),(5,1),(8,1),(10,2),(0,0)\}
\]

where the \(C_j\) refer to nodes and the \(L_j\) refer to distances from node 0. The values \((0,0)\) at the ends of the list are the null transfers.

Since the distances from node 0 to any two adjacent nodes differ by at most one, the remainder of the table can be filled in. For instance, node 9 can be reached by a single circumferential link transfer from either node 8 or node 10, giving total distances from

38
node 0 of 2 and 3 respectively. The minimum of these, namely 2, is the minimum distance between nodes 0 and 9. The remainder of the table is similarly derived, yielding:

<table>
<thead>
<tr>
<th>Node:</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distance:</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

and the network's diameter is seen to be two. The next pair of chordal transfers involves a distance of three which is too great to affect the final table.

In the general case the list is formed by sequences of clockwise and anti-clockwise chordal displacements until these displacements become too long to affect the internode distances. Let \( c_a, c_{a+1} \), \( (c_{a+1} > c_a) \) be any two nodes that are contiguous in the list of chordal transfers, with corresponding lengths (that is, minimum transfers from node 0) of \( L_a, L_{a+1} \). Then by interpolation the maximum possible length for any node \( c_k \) such that \( c_a < c_k < c_{a+1} \) is given by

\[
\max\{L_k\} = (c_{a+1} - c_a + L_{a+1} + L_a) \text{ div } 2 \pmod{N} \tag{3.1}
\]

The maximum minimal path for the full network, that is the diameter, is given by the maximum of the set of values defined by (3.1).

The number of alternate minimal internode paths between any pair of nodes is of importance in regard to fault tolerance and to the distribution of communications loads. Let \( (p, q) \) be a minimal path where

\[
(pd + q) \mod N = x, \quad p \in [-N/2d, N/2d], \quad q \in [-d/2, d/2].
\]

If \( p > 0 \) and \( q > 0 \), then \( (p, q) \) represents more than one possible path. For example, \( (2,1) \) takes the following possible routes:

- \( 0 \rightarrow d \rightarrow 2d \rightarrow 2d + 1 \)
- \( 0 \rightarrow d \rightarrow d + 1 \rightarrow 2d + 1 \)
- \( 0 \rightarrow 1 \rightarrow d + 1 \rightarrow 2d + 1 \)

That is, there are three possible routes. Referring to Figure 3.1, the options can be represented by a binary tree. As is evident from Figure 3.1, if the total choices of the minimal routes to \( pd + q \) is designated by \( T(p, q) \), then

\[
T(p, q) = T(p, q - 1) + T(p - 1, q) \tag{3.2}
\]

In particular,

\[
T(0, q) = T(p, 0) = 1.
\]

Solving this recurrence relation

\[
T(p, q) = \frac{(p + q)!}{p!q!} \tag{3.3}
\]

Equation (3.3) can be readily verified by substituting into equation (3.2), and a proof is provided in Appendix 3.1 at the end of this Chapter. If the node can also be expressed in terms of a set of optimal paths with \( p < 0 \) then (3.3) also applies to these alternative paths. Thus \( T(p, q) \) supplies a lower bound on the number of optimal paths.
3.5 A Special Class Of Chordal Ring Network

In this section a special class of network is derived from the condition that the network diameter should be a minimum. As was shown above, this is equivalent to minimising the mean internode distance. A consideration of the way in which the list of chordal transfers is constructed provides a technique for structuring a network to minimise the diameter.

Designate the middle node by $c_m$, so that

$$c_m = N \div 2$$

The maximum length between points $(c_j, L_j)$ and $(c_{j+1}, L_{j+1})$ in the list of chordal transfers is given by

$$k_j = (c_{j+1} - c_j + L_j + L_{j+1}) \div 2 \quad (3.4)$$

occurring at node

$$c_{xj} = (c_{j+1} + c_j + L_{j+1} - L_j) \div 2 \quad (3.5)$$

The network diameter is $k = \max\{k_j\}$. The list of chordal transfers is represented diagrammatically in Figure 3.2. Two possibilities exist for the distribution of minima.
<table>
<thead>
<tr>
<th></th>
<th>Node</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clockwise</td>
<td>$N \ (mod \ N)$</td>
<td>0</td>
</tr>
<tr>
<td>Clockwise</td>
<td>$2p_m d$</td>
<td>$2p_m$</td>
</tr>
<tr>
<td>Clockwise</td>
<td>$N - d$</td>
<td>1</td>
</tr>
<tr>
<td>Clockwise</td>
<td>$(p_m + 1)d$</td>
<td>$p_m + 1$</td>
</tr>
<tr>
<td>Anti-clockwise</td>
<td>$N - p_m d$</td>
<td>$p_m$</td>
</tr>
<tr>
<td>Anti-clockwise</td>
<td>$c_m$</td>
<td></td>
</tr>
<tr>
<td>Clockwise</td>
<td>$p_m d$</td>
<td>$p_m$</td>
</tr>
<tr>
<td>Anti-clockwise</td>
<td>$N - (p_m + 1)d$</td>
<td>$p_m + 1$</td>
</tr>
<tr>
<td>Clockwise</td>
<td>$d$</td>
<td>1</td>
</tr>
<tr>
<td>Anti-clockwise</td>
<td>$N - 2p_m d$</td>
<td>$2p_m$</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 3.2: List of Chordal Transfers.

(Clockwise and anti-clockwise chords, starting from node 0)

$p_m$ stands for the middle value of $p.$

about the central node. Figure 3.2 shows the order $f$ (forward or clockwise), $c_m$ (centre), $r$ (reverse or anti-clockwise) where clockwise and anti-clockwise refer to the direction
of traversal of the network via the chords. This order will be used for the subsequent calculations. The alternative system involves $r$, $c_m$, $f$, and the corresponding results will be stated but not derived. Figure 3.3 illustrates the chordal transfers for the 'clockwise'

<table>
<thead>
<tr>
<th>Node</th>
<th>Length</th>
<th>Local Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clockwise</td>
<td>$(j + 1)d$</td>
<td>$j + 1$</td>
</tr>
<tr>
<td>Anti-clockwise</td>
<td>$N - 2p_m d + jd$</td>
<td>$2p_m - j$</td>
</tr>
<tr>
<td>Clockwise</td>
<td>$jd$</td>
<td>$j$</td>
</tr>
</tbody>
</table>

Figure 3.3: Chordal Transfers for Clockwise Nodes.

nodes $j$, $j + 1$ bracketting an 'anti-clockwise' node $N - 2p_m d + jd$. Let the local maxima between pairs of nodes be represented by $k_{j1}$, $k_{j2}$ as shown in Figure 3.3. From (3.4)

\[
\begin{align*}
    k_{j1} &= \left[2p_m(1 - d) + N\right] \text{ div } 2 \\
    k_{j2} &= \left[(2p_m + 1)(1 + d) - N\right] \text{ div } 2 \\
\end{align*}
\]

Equation (3.6) gives the maximum internode distance within the two intervals illustrated in Figure 3.3. The objective is to minimize $k_j$ where

\[k_j = \max\{k_{j1}, k_{j2}\}\]

This will occur where $k_{j1} = k_{j2}$. Then, making use of the fact that

\[x \text{ div } 2 = y \text{ div } 2 \Rightarrow x = y + \delta, \delta \in \{-1, 0, 1\}\]

it follows that

\[2N = (4p_m + 1)d + (1 + \delta)\]
Substituting the value of \( d \) derived from (3.7) into (3.6), and noting that \( k_j = k_{j1} \),

\[
k_j = [N + 2p_m(4p_m + 2 - 2N + \delta)] \div (4p_m + 1)] \div 2
\]

(3.8)

Since the right hand side of (3.8) is independent of \( j \), the network diameter \( k \) is given by 
\( k = k_j \) for all \( j \) in the list of chordal transfers. \( k \) is a function of \( p_m \) and the objective is to minimize \( k \). From (3.8),

\[
k' = \frac{16p_m^2 + 8p_m - 2N + \delta + 2}{(4p_m + 1)^2}
\]

is the derivative with respect to \( p_m \). Using \( k' = 0 \) to determine the conditions for \( k \) to be a minimum, and requiring that the solution be integral, it follows that

\[
N = 8p_m^2 + 4p_m + 1
\]

Substituting in (3.8),

\[
k = 2p_m
\]

and from (3.7),

\[
d = 4p_m + 1
\]

Expressing \( n \) and \( d \) in terms of \( k \):

\[
N = 2k^2 + 2k + 1
d = 2k + 1
\]

(3.9)

and the network diameter is \( O(\sqrt{N}) \). Furthermore,

\[
k'' = \frac{8(2N - \delta - 1)}{(4p_m + 1)^3} > 0
\]

so that the corresponding values of \( k \) are minima, as required.

As mentioned above, an alternative scheme exists for the distribution of chordal transfers about the central node. In terms of the parameter \( p_m \) this scheme gives

\[
N = 8p_m^2 - 4p_m + 1
\]

\[
k = 2p_m - 1
\]

\[
d = 4p_m - 1
\]

Expressing \( N \) and \( d \) in terms of \( k \) the result is the same as given in (3.9). For both schemes \( N \) and \( d \) are odd; even values of \( k \) generate the first set while odd values generate the second. Thus (3.9) generates a set of network parameters from the set of integral values of \( k \). These networks will be referred to as optimal chordal ring networks.

The procedure followed above can be repeated for the alternative assumptions of \( N \) even and/or \( d \) even. In none of these cases are there any optimal solutions.
The list of chordal transfers can be used to calculate the mean internode distance. The distances $D = |p| + |q|$ from node 0 to nodes 1, 2, ..., $c_m$ (that is, for the first half of each network), together with the corresponding values of $p$ and $q$, for the first three networks in the sequence are as follows:

To node: 1 2 3 4 5 6 7 8 9 10 11 12
for k=1: p: 0 -1
q: 1 0
D: 1 1
for k=2: p: 0 0 -2 1 1 1
q: 1 2 0 -1 0 1
D: 1 2 2 2 1 2
for k=3: p: 0 0 0 -3 1 1 1 1 -2 -2 -2
q: 1 2 3 0 -2 -1 0 1 2 -1 0 1
D: 1 2 3 3 2 1 2 3 3 2 3

The sums of these sequences can be expressed as follows:

$$S_1 = 2 \times 1$$
$$S_2 = 2 \times 1 + 4 \times 2$$
$$S_3 = 2 \times 1 + 4 \times 2 + 6 \times 3$$

Extending this pattern,

$$S_k = 2 \times 1 + 4 \times 2 + \ldots + 2k^2$$
$$= 2 \sum_{j=1}^{k} j^2$$
$$= \frac{k(k+1)(2k+1)}{3}$$

This sum must be doubled to give the sum for the full set of nodes. Dividing by $N$ the mean internode distance is

$$\bar{D} = \frac{2k(k+1)(2k+1)}{3(2k^2 + 2k + 1)} \quad (3.10)$$

Note that mean internode distance is $O(\sqrt{N})$. Parameters for some of the optimal networks are given in Table 3.1.

Referring to the pattern of distances developed above, since the number of inter-processor references requiring $i$ links is $4i$, the probability that $i$ links will be required is

$$P(i) = \frac{4i}{N-1} = \frac{2i}{k(k+1)}, \quad i \leq k \quad (3.11)$$

The sequences given above for $k = 1, 2, 3$ can also be used to derive a useful result concerning the distribution of traffic on the internode connections. In each case $\sum |p| =
Table 3.1: Some Optimal Networks.

The most noteworthy feature is that the diameter is increased by one. This follows independently from the fact that the original network was optimal. Summarising the pattern established in Table 3.2, the total additional distance from node \( a \) is \( \delta = 2k - 1 \). Since only nodes \( a \) and \( b \) are affected the total additional distance is \( 2\delta \) and the increase in mean is \( 2\delta / N^2 \).

Defining degradation as

\[
degradation = \frac{\text{increase in mean}}{\text{mean}} \times 100\%
\]

the following formula results:

\[
degradation = \frac{300(2k-1)}{k(k+1)(2k+1)(2k^2 + 2k + 1)}
\]

(3.12)
(Change of distances from node 0 to the listed node resulting from a break between nodes 0 and 1)

<table>
<thead>
<tr>
<th>k</th>
<th>Node:</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>Total additional distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>original:</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>new:</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>original:</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>new:</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>original:</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>new:</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.2: Fault Tolerance.

For $k \geq 3$ the degradation is less than 1 percent. The minimum number of links which must be severed to isolate part of the network is four.

3.6 Extension To Networks Of Any Size

It might be supposed that the diameter of an optimal network is a monotonically increasing function of the number of nodes. Although this proves to be the case for most values of $N$ there are sufficient exceptions to rule out any exact predictions for diameter. Since the special class of networks described in section 3.5 represents an optimal case, the diameter $k$ for a network with $N$ nodes must be such that

$$k \geq \frac{\sqrt{2N-1}-1}{2}$$

(from (3.9)). The optimum displacement is multivalued and does not follow any simple rule. Thus the analysis presented in section 3.5 can only be used as a guide in the case of networks that do not belong to the optimal set.

Rearranging (3.10) the mean internode distance $\bar{D}$ can be expressed in terms of the number of nodes $N$:

$$\bar{D} = \frac{(N-1)\sqrt{2N-1}}{3N}$$

This represents a lower bound on the mean internode distance for non-optimal networks.

In comparing a 16 node hypercube (degree and diameter = 4) with a symmetric chordal ring network of degree four, it is to be noted that the diameter of an optimal chordal ring network with 16 nodes is three, and the maximum number of nodes for a chordal ring of diameter four is 41. Thus the performance of the hypercube is inferior to that of a chordal ring network. A simple routing algorithm for a chordal ring network is provided in Appendix 3.2 at the end of this Chapter.
3.7 Modified Chordal Rings

A transputer-based image-processing system should meet two requirements in particular:

1. A link must be provided so that a master or host can communicate with the network,
2. Segmented images should map readily onto the network.

Requirement (1) means that the chordal ring must be broken. Since a pure chordal ring is symmetric the break can be made at any point, and for convenience will be made in the edge connecting nodes $N - 1$ and 0. Requirement (2) depends on the manner in which images are segmented. Since this is commonly carried out in a cellular fashion, chordal rings that are isomorphic to cellular arrays (with appropriate edge connections) are of particular interest.

In the remainder of this section the effects of breaking a chordal ring will be investigated and the conditions for a cellular array to map onto a chordal ring will be studied.

### 3.7.1 Broken Chordal Rings

A break between nodes $N - 1$ and 0 to provide communication with a host means that some routes that would optimally go between $N - 1$ and 0 are lengthened. The consequent loss of communications efficiency is of some significance, and this will be calculated for a broken special chordal ring network.

Consider routes starting from node 0. Routes proceeding circumferentially via node $N - 1$ are those to nodes $N - g$ where $g \leq d/2$. Since $d = 2k + 1$, $d \text{ div } 2 = k$. Firstly, consider node $N - k$, which is reachable in $k$ steps in an unbroken network. With the break let node $N - k$ be optimally reachable via $pd + q$. Then

$$pd + q = N - k = 2k^2 + k + 1 = k(2k + 1) + 1 = kd + 1$$

Thus $p = k$ and $q = 1$. The distance is thus $k + 1$, which is 1 more than the previous route. Now consider node $N - k + i$, $i \geq 1$. The unbroken route is of length $k - i$. If the new route is $pd + q$ then

$$pd + q = N - k + i = kd + (1 + i)$$

Thus $p = k$ and $q = 1 + i$, giving an extra distance of $1 + 2i \geq 3$. However any node $N - k + i$ is reachable by 2 extra steps by means of

$0 \rightarrow d \rightarrow d - 1 \rightarrow N - 1 \rightarrow \cdots \rightarrow N - k + i$

where the first three steps replace $0 \rightarrow N - 1$. Thus at most two extra steps are required for any node $N - k + i$. Proceeding in this manner a table of additional distances can be constructed as shown in Table 3.3.
To node: 

<table>
<thead>
<tr>
<th>From node</th>
<th>( N - k )</th>
<th>( N - k + 1 )</th>
<th>( N - k + 2 )</th>
<th>( \ldots )</th>
<th>( N - 1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>( \ldots )</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>( \ldots )</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>( \ldots )</td>
<td>2</td>
</tr>
<tr>
<td>( \ldots )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( k - 1 )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>( \ldots )</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 3.3: Additional Distances for a Broken Chordal Ring.

The sum of the \( i^{th} \) row is \( 2(k - i - 1) + 1 \), so the total extra distance is given by

\[
S = \sum_{i=0}^{k-1} (2(k - i - 1) + 1) = k^2
\]

Since additional distance is also involved for routes in the opposite direction the expression (3.10) for mean becomes

\[
\bar{D} = \frac{2k[(k + 1)(2k + 1)(2k^2 + 2k + 1) + 3k]}{3(2k^2 + 2k + 1)^2}
\]  

(3.15)

and efficiency is

\[
\frac{\bar{D}_{\text{old}}}{\bar{D}_{\text{new}}} = \frac{(k + 1)(2k + 1)(2k^2 + 2k + 1)}{(k + 1)(2k + 1)(2k^2 + 2k + 1) + 3k}
\]  

(3.16)

The efficiencies of some broken chordal rings are given in Table 3.4.

<table>
<thead>
<tr>
<th>( k )</th>
<th>( N )</th>
<th>Actual Diameter</th>
<th>Efficiency (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>13</td>
<td>3</td>
<td>97.01</td>
</tr>
<tr>
<td>3</td>
<td>25</td>
<td>4</td>
<td>98.73</td>
</tr>
<tr>
<td>4</td>
<td>41</td>
<td>5</td>
<td>99.35</td>
</tr>
<tr>
<td>5</td>
<td>61</td>
<td>6</td>
<td>99.63</td>
</tr>
<tr>
<td>6</td>
<td>85</td>
<td>7</td>
<td>99.77</td>
</tr>
<tr>
<td>7</td>
<td>113</td>
<td>8</td>
<td>99.85</td>
</tr>
<tr>
<td>8</td>
<td>145</td>
<td>9</td>
<td>99.89</td>
</tr>
<tr>
<td>9</td>
<td>181</td>
<td>10</td>
<td>99.92</td>
</tr>
<tr>
<td>10</td>
<td>221</td>
<td>11</td>
<td>99.94</td>
</tr>
</tbody>
</table>

Table 3.4: Efficiency.

These figures can be taken as an indication of the performance of general broken chordal rings. The analysis in the general case is much more involved and a simple expression has not been obtained. A routing algorithm for a broken chordal ring is given in Appendix 3.3 at the end of this Chapter.
3.7.2 Mapping Cellular Arrays Onto Chordal Rings

The cellular array is of particular interest in image processing since rectangular images can be readily segmented in a cellular fashion. Thus the mapping of a segmented image onto a cellular array is conceptually simple, although by no means necessary since a suitable operating system will make all inter-processor communications transparent to the user. The objective of this section is to find out how to map cellular arrays onto chordal rings. For this purpose it is necessary to provide formal definitions of the two networks.

Define a chordal ring as a graph consisting of a set of nodes \( P = \{0, \cdots, N - 1\} \) and a set of edges \( E = \{r_0, \cdots, r_{N-1}, c_0, \cdots, c_{N-1}\} \) where \( r_i \) (called a 'ring' edge) is the unordered pair \((p_i, p_{i+1})\) and \( c_i \) (called a 'chordal' edge) is the unordered pair \((p_i, p_{i+d})\) where \( d \) is the chordal displacement, and the subscript addition is modulo \( N \).

Define a cellular array as a graph consisting of a set of nodes \( P = \{(i,j), i \in [0, N_r - 1], j \in [0, N_c - 1]\} \) and a set of edges \( E = \{r_{i,j}, i \in [0, N_r - 1], j \in [0, N_c - 2]; c_{i,j}, i \in [0, N_r - 2], j \in [0, N_c - 1]\} \) where the 'row' \( r_{i,j} \) is the unordered pair \((p_{i,j}; p_{i,j+1})\) and \( c_{i,j} \) is the unordered pair \((p_{i,j}; p_{i+1,j})\). Note that row edges corresponding to \( j = N_c - 1 \) and column edges corresponding to \( i = N_r - 1 \) are not defined. The procedure will involve seeking chordal rings isomorphic to cellular arrays allowing the undefined edges to match those of the chordal ring.

Let \((i, j)\) be a node in the cellular array. Consider the following transformation:

\[
(i, j) \rightarrow (N_c i + j) \tag{3.17}
\]

Set \( p = N_c i + j \). Disregarding edges corresponding to \( j = N_c - 1 \) and \( i = N_r - 1 \), the row edges are \\{(\(i, j\), \(i, j+1\))\} and the column edges are \\{(\(i, j\), \(i+1, j\))\}. Applying the transformation (3.17) and interpreting addition as modulo \( N \) where \( N = N_c N_r \), the row edges become

\[
\{(N_c i + j, N_c i + j + 1)\} = \{(p, p + 1)\}
\]

and the column edges become

\[
\{(N_c i + j, N_c (i + 1)j)\} = \{(p, p + N_c)\}
\]

which is a chordal ring with \( N \) nodes and a displacement of \( N_c \).

For instance, a network of twenty transputers could be segmented as four rows of five columns, using a chordal displacement of five (see Figure 3.4), or as five rows of four columns, using a chordal displacement of four. Allowing for the break between nodes 34 and 00, the diameter in both cases is four, while the mean distance is 2.230 and 2.120.
respectively. The corresponding mean distance for the optimum displacement is 2.080. The requirement for cellular mapping thus involves a net efficiency of 93.3% and 98.1% respectively.

Other broken chordal rings have been simulated. For instance, given a network of 120 transputers, the optimum displacements of 14 and 16 both give a diameter of 8 and a mean path length of 5.140. The network can be factored in numerous ways for cellular mappings, and the results are summarised in Table 3.5 where efficiency is relative to that obtained with a displacement of 14 or 16. From the table it can be seen that for segment aspect ratios not exceeding 2:1, that is, those between 8:15 and 15:8, the loss of efficiency does not exceed approximately 10%. However, the penalty for larger segment aspect ratios increases rapidly.

### 3.8 Performance

Table 3.6 summarises a series of performance measurements made on a broken chordal ring network consisting of twenty transputers. The test involved passing data packets between nodes so that each node sourced one packet for each other node (that is, 19 packets were
Table 3.5: Mapping a Chordal Ring of 120 Nodes onto a Cellular Array.

sent from every processor). Measurements were made for four chordal displacements, and for two data packet sizes — 1000 words (at four bytes per word) and 4000 words. These data are plotted in Figure 3.5.

<table>
<thead>
<tr>
<th>Rows</th>
<th>Columns</th>
<th>Diameter</th>
<th>Mean Path Length</th>
<th>Efficiency (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>60</td>
<td>31</td>
<td>15.617</td>
<td>32.9</td>
</tr>
<tr>
<td>3</td>
<td>40</td>
<td>21</td>
<td>10.714</td>
<td>48.0</td>
</tr>
<tr>
<td>4</td>
<td>30</td>
<td>16</td>
<td>8.515</td>
<td>60.4</td>
</tr>
<tr>
<td>5</td>
<td>24</td>
<td>14</td>
<td>7.203</td>
<td>71.4</td>
</tr>
<tr>
<td>6</td>
<td>20</td>
<td>12</td>
<td>6.489</td>
<td>79.2</td>
</tr>
<tr>
<td>8</td>
<td>15</td>
<td>11</td>
<td>5.741</td>
<td>89.5</td>
</tr>
<tr>
<td>10</td>
<td>12</td>
<td>10</td>
<td>5.463</td>
<td>94.1</td>
</tr>
<tr>
<td>12</td>
<td>10-</td>
<td>10-</td>
<td>5.453</td>
<td>94.3</td>
</tr>
<tr>
<td>15</td>
<td>8</td>
<td>11</td>
<td>5.677</td>
<td>90.5</td>
</tr>
<tr>
<td>20</td>
<td>6</td>
<td>12</td>
<td>6.418</td>
<td>80.1</td>
</tr>
<tr>
<td>24</td>
<td>5</td>
<td>14</td>
<td>7.201</td>
<td>71.4</td>
</tr>
<tr>
<td>30</td>
<td>4</td>
<td>16</td>
<td>8.376</td>
<td>61.4</td>
</tr>
<tr>
<td>40</td>
<td>3</td>
<td>21</td>
<td>10.667</td>
<td>48.2</td>
</tr>
<tr>
<td>60</td>
<td>2</td>
<td>30</td>
<td>15.250</td>
<td>33.7</td>
</tr>
</tbody>
</table>

Table 3.6: Performance of a Broken Chordal Ring.

In Table 3.6, $x$ is the mean link distance for the network. The times ($T_{1000}$ and $T_{4000}$) are mean values for each group. The difference $\Delta T$ is used in order to eliminate factors which are not a function of the communications aspect of the network. Regression analysis yields

$$\Delta T_{est} = 4.616x + 13.9$$

and this is used to produce the final column in the table. Variations in performance measurements, with standard deviations varying between 0.2% and 2% of the observed values, arise through the fact that the network used for the test was constructed of processors operating asynchronously.

The performance timings of the network fall into four groups, corresponding to the four displacements. There are two variables in the regression line: $x$ and $\Delta T$. It is necessary to determine whether the observed variations in the regression line are most likely to
have occurred through variations in the data. More precisely, it is necessary to test the hypothesis that

$$H_0 : \Delta T_i = a + bx_i, \ i \in [1..4]$$

The number of points used was 13, divided into four groups. Using the \( F \) distribution [294], \( F = 1.126 \). From tables of the percentage points of the \( F \) distribution [294], \( F_{0.25}(2, 13) = 1.55 \). Thus the hypothesis (3.18) cannot be rejected at confidence levels of 75% and above. Therefore the mean link distance \( x \) fully describes the observed values of \( \Delta T = T_{4000} - T_{1000} \). The linear correlation coefficient is \( r = 0.9986 \). Thus it has been shown that there is a linear relationship between link distance and communications time.
3.9 Conclusions

Symmetric chordal networks of degree four offer a practical means of interconnecting processors having four inter-processor links. The symmetry of the networks makes the mapping of algorithms onto the network a comparatively simple task. In particular, for image processing applications, an image will map very simply and conveniently onto the network. In this Chapter techniques have been given for determining the diameter and mean interprocessor distance for a given network. It has been shown that if a network is chosen to have a minimum diameter (by appropriate choice of the number of nodes and/or the chord length) then the mean internode distance will also be a minimum. A particular subset of these networks is optimal in the sense of having the greatest possible number of processors for a given diameter. In optimal networks the diameter and the mean are both of $O(\sqrt{N})$.

Although optimal networks are not incrementally extensible, any number of processors can be added to a general chordal ring network. If the number of additional processors is large the optimum chord length is likely to change. Single faults result in a comparatively small degradation in performance and a minimum of three faults can be tolerated in the network before any nodes become isolated. Although a simple algorithm exists for routing in an optimal chordal ring network, routing in non-optimal and broken networks might best be achieved by means of look-up tables. The theoretical performance of chordal rings has been verified by measurements on an actual chordal ring network.

3.10 Appendices to This Chapter

3.10.1 Appendix 3.1 : Proof Of Equation (3.3)

The most convenient proof of (3.3) is to derive the recurrence relation from it.

\[
T(p, q) = \frac{(p + q)!}{p!q!}
\]

\[
= \frac{(p + q)(p + q - 1)!}{p!q!}
\]

\[
= \frac{p(p + q - 1)!}{p!q!} + \frac{q(p + q - 1)!}{p!q!}
\]

\[
= \frac{(p + q - 1)!}{(p - 1)!q!} + \frac{(p + q - 1)!}{p!(q - 1)!}
\]

\[
= T(p - 1, q) + T(p, q - 1)
\]

The proof is completed by noting that $T(p, 0) = T(0, q) = 1$.
3.10.2 Appendix 3.2: A Routing Algorithm for a Chordal Ring

This algorithm generates a link number, where the links are labelled as shown in Figure 3.6.

```
{ Let source = originating node
  destination = destination node
  n = number of nodes in the system
  k = diameter. }

VAR
  difference, sign, index1, index2, link : INTEGER;
  { index1 and index2 are derived from the difference between
    the two node numbers and are used to determine the
    appropriate link number }
BEGIN
  IF destination <> source THEN
    BEGIN
      difference := destination - source;
      { Form difference in positive direction }
```
IF difference < 0 THEN difference := n + difference;
IF difference > (n DIV 2) THEN

{ It is a mirror image of the other case }
BEGIN
   difference := n - difference;
   sign := -1
   END ELSE sign := +1;

{ The simplest case - use ring moves }
IF difference <= k THEN
   IF sign < 0 THEN link := 0 ELSE link := 1 { -1 or +1 }
   ELSE { Make use of the symmetry properties of the network }
   BEGIN
      index1 := 2*((difference - k - 1) DIV (2*k)) + 1;
      index2 := difference - index1*k;
      IF index2 > index1 THEN
         IF sign < 0 THEN link := 3 ELSE link := 2 { +d or -d }
         ELSE { index2 <= index1 }
            IF sign < 0 THEN link := 2 ELSE link := 3 { +d or -d }
         END
      END
   END
   END
END;

3.10.3 Appendix 3.3: A Routing Algorithm for a Broken Chordal Ring

{ Let source = originating node
   destination = destination node
   n = number of nodes in the system
   displacement = chordal displacement }

VAR
   chordmoves, ringmoves : ARRAY [0..3] OF INTEGER;
   difference, n_minus_difference, index, link,
   minimum, sum, bestmove, result, compare : INTEGER;
   negative : BOOLEAN;
BEGIN
   minimum := very_large_number; { so that actual minimum will be smaller }
   difference := destination - source;
negative := (difference < 0);
IF negative THEN difference := -difference;
n_minus_difference := n - difference;

{ Set up options – there are four possibilities: } chormov[O] := difference DIV displacement;
ringmov[O] := difference MOD displacement;
chormov[1] := chormov[0] + 1;
ringmov[1] := displacement - ringmov[0];

{ Adjust for break between nodes 0 and n-1 } IF chormov[2] = 0 THEN chormov[2] := 2;

{ Find the minimum choice from then four possibilities } FOR index := 0 TO 3 DO
BEGIN
sum := chormov[index] + ringmov[index];
IF sum < minimum THEN { this is a better move }
BEGIN
minimum := sum;
bestmove := index
END
END;

{ Adjust sign } IF bestmove >= 2 THEN negative := NOT negative;

{ Make the choice } IF chormov[bestmove] <> 0 THEN
BEGIN
IF (ringmov[bestmove] > 0) AND (chormov[bestmove] = 1) THEN
BEGIN

{ Form temporary result } IF negative THEN result := source - displacement
ELSE result := source + displacement;

{ Adjust for overflow or underflow } IF result < 0 THEN result := n + result
ELSE IF result >= n THEN result := result - n;
{ Obtain absolute difference }
   compare := destination - result;
   IF compare < 0 THEN compare := -compare;
   IF compare >= displacement THEN

{ A ring move }
   BEGIN
      IF (bestmove = 1) OR (bestmove = 3) THEN
         { since ring motion is reverse of chordal motion }
         negative := NOT negative;
         IF negative THEN link := 0 ELSE link := 1
      END ELSE

{ A chordal move }
   IF negative THEN link := 3 ELSE link := 2
   END ELSE

{ Chordal move }
   IF negative THEN link := 3 ELSE link := 2
   END ELSE

{ Ring move }
   IF negative THEN link := 0 ELSE link := 1
   END;
Chapter 4

Multitransputer Image Processing

4.1 Introduction

The introduction to this thesis outlined the major objective of this research, namely the development of techniques for the real-time inspection and sorting of kiwifruit. Of the two major approaches, involving either large processing power or data reduction, this chapter deals with the former. The required processing power is in the region of hundreds of MIPS\textsuperscript{1}. Although this processing power may become available in single processors within the next decade, especially with the development of gallium arsenide microprocessors\textsuperscript{2}, currently, multiprocessors are required in order to obtain the required power.

A number of researchers have achieved practical implementations of transputers in networks – for instance, Bowler et al\textsuperscript{40} describe a reconfigurable network used to solve problems in physics, while Mills and O'Neill\textsuperscript{197} describe a system in which a fixed-configuration network is applied to electrostatic powder flow measurement. However, these seem to have been developed on an ad hoc basis, and what has been lacking is a 'harness' or 'shell' to overcome some of the problems that are specific to multi-transputer networks.

In this chapter the practical implementation of a multiprocessor network based on transputers is described. A processing shell was developed in response to problems such as deadlock, and this shell was used as the basis of the image processing system running on the network. Various possible network configurations are described, with the chordal ring\textsuperscript{2} being chosen because of its good global characteristics. However, the toroid and the ternary tree are both interesting alternatives. The image processing system runs on an IBM PC, and facilities to control the text and graphics screens are described. The

\textsuperscript{1} Millions of Instructions Per Second

\textsuperscript{2} See Chapter 3.

58
shell imposes strict message protocols, and these are used as the basis of the relationship between the host transputer and a network consisting of twenty transputers. Some measurements have been made to assess the performance of the system and are reported.

4.2 A Processing Shell for the Transputer Network

In Chapter 2 the problem of deadlock in a processing network was considered, and various approaches to the problem were reviewed. Some of the difficulties associated with deadlock in a multi-transputer system have been referred to by Morrow et al [198]. Distributed deadlock detection can take place with a central controller or can be fully distributed. Detection would be used in conjunction with some form of resolution. For deadlock prevention all required resources for a process must be declared at the system design stage or at compile time, which makes deadlock prevention a very conservative approach, with an overall undercommitment of resources. Deadlock avoidance is a course of action midway between the other two. Whereas detection allows deadlocks to occur and prevention ensures that deadlocks can never occur, avoidance allows processes to proceed only if the required resources are available.

The two principal types of deadlock of relevance to the system that I have developed are:

- Provision of insufficient resources in the form of buffers so that transactions cannot proceed.
- A cycle in the wait-for-graph (WFG)\(^3\).

Since the development of efficient algorithms for deadlock detection is still at an early stage, and published algorithms involved processes which may not have been easy to implement on a transputer network, or may not have functioned correctly when implemented, I decided not to use deadlock detection or deadlock avoidance. Instead, the approach adopted was to use deadlock prevention for store-and-forward deadlock.

In regard to cycle detection in the WFG, consider Figure 4.1. Suppose that Figure 4.1(a) illustrates the WFG of a two-processor system at some time instant — that is, it is a system 'snapshot'. Ensuring that there are no cycles in the graph at this instant and at later instants is a significant problem, even in those cases in which the WFG can be determined at the time of designing and programming the system. The same graph is repeated in Figure 4.1(b) with the addition of single input and output processes on each processor. With these in place the problem of detecting cycles is significantly reduced, even in the case of two processors. When applied to the Department's network of twenty transputers this approach results in a major simplification.

\(^3\)See Chapter 2.
Figures 4.1(b) and 4.2 illustrate the operation of part of a 'link shell' which was designed as a programming aid, following early difficulties in achieving deadlock-free code. Other features of the shell include a message logger that resides in each node and can be interrogated by the host, and a message format that, while not restricting the system in any way, imposes consistency checks. Cormack et al. [75] note that problems and errors can create such complex symptoms that special test harnesses may be necessary to provide incremental test information. An unstruc-
Norman and Fisher [210] have described a shell (or 'communications harness') which performs some of the functions of the shell described in this section.

The remainder of this section is organised as follows. Section 4.2.1 describes the format adopted for all messages. Section 4.2.2 provides an overview of the link shell. Section 4.2.3 describes the way in which the channel buffers that are used at the input, merged stream, and output stages of the shell operate. Section 4.2.4 examines the issues determining the optimum sizes of the buffers and the message-passing efficiency as a function of buffer size. Section 4.2.5 describes the facility that allows 'post-mortems' to be held on communications within the network.

4.2.1 Messages

A major aspect of the link shell is its use of a specific message format which, if adhered to, provides significant checking while imposing comparatively little overhead. The message
format is as follows:

\[
\text{destination; number; tag; [data values;...]} \text{ end.of.message}
\]

where destination is the unique identifier of the target transputer (several link transfers may be needed to reach it), the number in the message is the count, excluding \textit{end.of.message}, of the values that follow, the tag is an integer that identifies a particular process (and usually a specific number of data items), and the \textit{end.of.message} marker is a unique integer. It is up to the writer of the applications software to specify a particular message format for each tag value.

### 4.2.2 Link Communications

An overview of the link shell has been presented in Figure 4.2 and a simplified version of the folded code is given below\(^4\). A detailed explanation of Occam is given in Appendix D. However, a brief description will be given here to assist in the interpretation of the code. The Occam runs under INMOS's Transputer Development System (TDS) which includes a folding editor. Analogous to the closing of a book, a 'fold' of code can be closed, leaving a 'crease' whose presence is indicated by three dots. In this chapter such folds will be opened where relevant. Scope in Occam is established by indentation, and processes are of three types: sequential (SEQ), parallel (PAR), and alternative (ALT). Channels, declared using CHAN, are abstract objects that can exist purely as data items ('soft' channels) or as inter-transputer links ('hard' channels). In the code anything following two hyphens is a comment. The link shell code is as follows:

\begin{verbatim}
PROC frame ([4]CHAN in, out, VAL INT processor)
   -- This is an idealised version of the network transputer shell
   -- and is not fully supported by the Transputer Development System
   -- Some details have been omitted for clarity.
   [4] [buffer.size+1]CHAN OF INT transfer: -- channel buffer
   -- the optimum buffer size depends on the application
   ... other declarations

PROC Link.Response (VAL INT link.num, VAL INT value)
   -- to send value to link L: transfer[L][0] ! value
   SEQ
      ... user coding goes in here
   : -- Link.Response

... PROC channel.buffer
   -- This procedure consists of a pipeline of channels to provide
\end{verbatim}

\(^4\)A more extensive software listing is provided in Appendix E.
-- a buffer between output from the user code in Link.Response
-- and the actual transputer links.

-- Main code starts here
SEQ
 processing := TRUE
 PAR
 WHILE processing
   INT x :
   ALT link = 0 FOR 4
      in[link] ? x
      Link.Response(link,x)
   WHILE processing
   PAR link = 0 FOR 4
      channel.buffer(link,out[link])
      -- 'soft' channels to
      -- output links
 : -- frame

Data appearing on an input link are read and placed in the appropriate row of the input buffer. Data passes along the rows of the input buffer and on exiting are merged into a single stream by means of an ALT statement. When the data values are read from the input buffer the first three values, namely destination, number, and tag, are used as the basis of a consistency check. The tag is associated with a specific message, and generally such messages will have a length that can be specified at system-design time. The expected lengths are included in the software and a check is made to ensure that the number in the message header is the same as the number expected for that specific tag. The remainder of the message is then read in and placed in the merged buffer, which is a single buffer. However, when a difference between the expected and actual numbers is detected the remainder of the message up to end.of.message is read in and discarded, and an error message can be sent to the host. Variable length messages can be accommodated — a reserved integer (specifically -1, since this should never occur as an actual count) is provided as the count associated with that tag, and no test is made.

The output from the merged buffer is passed to the applications software. This is, of course, not part of the shell. It is assumed, however, that in the course of the data-processing various output messages will be generated. These are placed in one of four output buffers, corresponding to the four links. There are two reserved integers in regard to the output buffers: end.buf and table.buf. The first is used to flush the output buffers and is not placed on an output link. The second indicates that a table is to be referenced, the next two integers of the message providing the table identifier and the number of values to be passed. A table is any data structure that can be defined as an array in Occam, and the appropriate table must have been correctly initialised by the applications software.
In particular, in the application of the transputer network to image-processing the table number is interpreted as the image number. This ensures that large data structures such as image segments do not impose an unduly high overhead by requiring very large channel buffers.

The overall structure of the software has been designed so that all network processors use the same basic process. Since channel allocations are dependent on their locations in the network, this information is passed in to the process via the process parameters which are then used to assign the internal channel references. This structure can be seen in the listing in Appendix E.

4.2.3 Channel Buffers

The channel buffers are adapted from a technique described by Pountain [222], and their operation will be explained for the particular case of the output buffers. They are illustrated diagrammatically in Figure 4.3 and are declared as

\[
\text{[4][out.buf.size+1]CHAN OF INT out.buffer:}
\]

where the values in square brackets are the array dimensions. The first of these dimensions specifies the four rows corresponding to the four output links and the actual value of out.buf.size is chosen to be large at the beginning of the system development and is steadily reduced until deadlock occurs.

The outer PAR process of the shell includes the code shown:

\[
\begin{align*}
\text{PAR} \\
\text{WHILE TRUE}
\end{align*}
\]
output.buffer (0, host.side.out)
WHILE TRUE
  output.buffer (1, network.side.out)
WHILE TRUE
  output.buffer (2, forward.chord.out)
WHILE TRUE
  output.buffer (3, reverse.chord.out)

The WHILE TRUE construct ensures that the process that follows (in this case an invocation of output.buffer) will be repeated indefinitely. The links are designated host.side.out and so on. The PROC output.buffer has the general structure shown below:

PROC output.buffer (VAL INT output.buf.num, CHAN output)
  ...
  PROC transfer.along
  ...
  PROC transfer.out
  PAR
    transfer.along ()
    transfer.out ()
  : -- output.buffer

The purpose of transfer.along is to pass data values from one channel element to the next. The code for this is

PROC transfer.along ()
  PAR buffer.element = 0 FOR out.buf.size
  SEQ
    input IS out.buffer[output.buf.num][buffer.element] :
    output IS out.buffer[output.buf.num][buffer.element+1] :
    INT next.value :
    WHILE TRUE
      SEQ
        input ? next.value
        output ! next.value -- pass it on
    : -- transfer.along

The effect of transfer.along is to transfer values through to the end of the buffer, and it is the function of transfer.out to place the values on the output links. The code is shown below, with output identifying the particular output link.

PROC transfer.out ()
  PROC handle.table ()
    -- The tables are application-specific (e.g. image segments)
INT table.id, number :
SEQ
  out.buffer[output.buf.num][output.buf.size] ! table.id; number
  IF (table.id >= 0) AND (table.id < num.of.tables)
    SEQ -- This is a valid table
      SEQ table.pointer = 0 FOR number
      output ! tables[table.id][table.pointer]
      -- Transfer the contents of the table
      free.tables[table.id] := TRUE
      -- it is no longer needed (this instruction optional)
    otherwise
      handle.errors () -- An application-specific handler.
  : -- handle.table
INT next.value :
WHILE TRUE
SEQ
  out.buffer[output.buf.num][output.buf.size] ? next.value
  -- Obtain the next data value from this row
  -- of the output buffer.
  IF
    next.value = end.buf  -- One of the reserved values
      SKIP -- Flushing the buffer so nothing to transfer
    next.value = table.buf  -- Another of the reserved values
      handle.table ()
    otherwise  -- Not a reserved value - send as is.
      output ! next.value
  : -- transfer.out

4.2.4 Sizes of Buffers

A test of the network was formulated in which each of the first $N$ processors in the network sends a data packet to each of the other $N - 1$ processors. The actual message-passing protocol included a system of queues in which a processor establishes and maintains a queue containing the identities of those data packets above a specified size (approximately fifteen integer-sized words in the existing implementation, and referred to as tables). The recipient of a table sends an acceptance tag, and on receipt of this tag the sender of the table then sends the next one from the queue. The buffer characteristics were investigated in the context of a chain of processors and of a chordal ring. In this section the case of transfer in a single direction is analysed, this is extended to bi-directional transfers, and performance measurements for packet transfers as a function of buffer sizes

\[\text{See Chapter 3.}\]
are reported. These performance measurements include the minimum buffer sizes such that deadlock is prevented.

Let a chain of \( N \) processors be involved in uni-directional transfers such that each processor sends packets to all processors above it in the chain. Thus processor 1 sends packets to processors 2, 3, \( \cdots \), \( N \), giving a total of \( N - 1 \) packets. Label data packets by the ordered pair \((\text{source}, \text{destination})\). Then the packets sent by processor 1 are \((1,2), (1,3), \cdots, (1,N)\). Processor 2 sends packets to processors 3, 4, \( \cdots \), \( N \), a total of \( N - 2 \). In addition processor 2 passes on packets \((1,3), (1,4), \cdots, (1,N)\) from processor 1, giving an extra \( N - 2 \) packets. Thus processor 2 sources or transfers a total of \( 2(N - 2) \) packets. Similarly processor 3 sources \( N - 3 \) packets, and passes on \( N - 3 \) packets from processor 1 and \( N - 3 \) packets from processor 2, a total of \( 3(N - 3) \) packets. In general the \( i^{th} \) processor sources the packets with labels \((i,i + 1), (i,i + 2), \cdots, (i,N)\), a total of \( N - i \). Furthermore this processor transfers \( N - i \) packets from each of the preceding \( i - 1 \) processors, a total of \((i-1)(N-i)\) packets. The overall total for the \( i^{th} \) processor is thus \( i(N - i) \).

The total number of packet transfers for a uni-directional transfer is thus

\[
T_{\text{uni}} = \sum_{i=1}^{N-1} i(N - i) = N \sum_{i=1}^{N-1} i - \sum_{i=1}^{N-1} i^2
\]

Use

\[
\sum_{i=1}^{p} i^2 = \frac{1}{6} p(p + 1)(2p + 1)
\]

and

\[
\sum_{i=1}^{p} i = \frac{1}{2} p(p + 1)
\]

Then

\[
T_{\text{uni}} = \frac{1}{6} (N - 1)N(N + 1)
\]

The total number of packets involved in a bi-directional transfer will be just twice this. That is

\[
T_{\text{bi}} = \frac{1}{3} (N - 1)N(N + 1) \tag{4.1}
\]

It would be anticipated that the total time taken for these transfers would be proportional to \( T_{\text{bi}} \). Equation (4.1) will be compared with measurements of time for a sequence of values of \( N \).

In Table 4.1 the times were obtained by subtracting the execution time for a single processor from the total execution time in order to eliminate the fixed time. It is hypothesised that the resultant times conform to the following expression:

\[
Time = \alpha T_{\text{bi}} + \beta N
\]

where the first term is the transfer time and the second term allows for processes that are internal to each processor. The value of \( T_{\text{bi}} \) is given in (4.1). The root-mean-square fit of the data in Table 4.1 to this expression gives

\[
Time = 0.00126T_{\text{bi}} + 0.041N \tag{4.2}
\]

67
Table 4.1: Measured Packet Transfer Times (in seconds) as a Function of Number of Processors.

(N is number of processors, \( T_{bi} \) is number of packets)

<table>
<thead>
<tr>
<th>( N )</th>
<th>( Time )</th>
<th>( T_{bi} )</th>
<th>( N )</th>
<th>( Time )</th>
<th>( T_{bi} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.000</td>
<td>0</td>
<td>11</td>
<td>1.063</td>
<td>440</td>
</tr>
<tr>
<td>2</td>
<td>0.008</td>
<td>2</td>
<td>12</td>
<td>1.279</td>
<td>572</td>
</tr>
<tr>
<td>3</td>
<td>0.016</td>
<td>8</td>
<td>13</td>
<td>1.542</td>
<td>728</td>
</tr>
<tr>
<td>4</td>
<td>0.037</td>
<td>20</td>
<td>14</td>
<td>1.820</td>
<td>910</td>
</tr>
<tr>
<td>5</td>
<td>0.100</td>
<td>40</td>
<td>15</td>
<td>2.135</td>
<td>1120</td>
</tr>
<tr>
<td>6</td>
<td>0.200</td>
<td>70</td>
<td>16</td>
<td>2.470</td>
<td>1360</td>
</tr>
<tr>
<td>7</td>
<td>0.312</td>
<td>112</td>
<td>17</td>
<td>2.836</td>
<td>1632</td>
</tr>
<tr>
<td>8</td>
<td>0.475</td>
<td>168</td>
<td>18</td>
<td>3.194</td>
<td>1938</td>
</tr>
<tr>
<td>9</td>
<td>0.644</td>
<td>240</td>
<td>19</td>
<td>3.591</td>
<td>2280</td>
</tr>
<tr>
<td>10</td>
<td>0.839</td>
<td>330</td>
<td>20</td>
<td>4.028</td>
<td>2660</td>
</tr>
</tbody>
</table>

with an RMS error of 0.1. Combining the regression equation with (4.1), the resultant data are plotted in Figure 4.4.

![Figure 4.4: Execution Time for Packet Transfer.](image)

It is desired to characterise the performance of the shell in terms of the buffer sizes. Two measures are of significance: the minimum set of values that will prevent deadlock, and the execution time for some standard data transfer as a function of buffer size. Since three sets of buffers are involved, this treatment will look at each in turn, and will then
combine the three to estimate performance as a function of buffer sizes for a chordal ring network.

The execution time $T$ of a bi-directional data transfer of 4KByte packets between $N$ processors is given in Table 4.2. The execution times in seconds were determined by subtracting the execution time of a single-word packet from that for the full 4KByte packet, thus eliminating packet-independent times. Both a linear and a quadratic expression were fitted to these data values. The quadratic fit was not significantly better than the linear fit, which was

$$Time = (0.0556N - 0.238)buf_{in} + (0.459N - 1.95)$$

(4.3)

A similar procedure was followed in the case of the merged buffer size. The execution time $T$ of a bi-directional data transfer of 4KByte packets between $N$ processors is given in Table 4.3. The linear fit was

$$Time = (0.0494N - 0.185)buf_{in} + (0.513N - 2.37)$$

(4.4)

Finally the performance of the output buffer was measured, and the results are shown in Table 4.4 for 16KByte packets. The most important conclusion to be drawn from Table 4.4 is that the packet transfer time is approximately independent of the output buffer size.

Given that execution time increases as both $buf_{in}$ and $buf_{merged}$ increase, deadlock prevention was effected by making $buf_{in}$ and $buf_{merged}$ small, and allowing $buf_{out}$ to increase.
<table>
<thead>
<tr>
<th>Output Buffer Size</th>
<th>Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5</td>
</tr>
<tr>
<td>40</td>
<td>3.173</td>
</tr>
<tr>
<td>60</td>
<td>3.354</td>
</tr>
<tr>
<td>80</td>
<td>3.048</td>
</tr>
<tr>
<td>100</td>
<td>3.082</td>
</tr>
<tr>
<td>120</td>
<td>3.121</td>
</tr>
<tr>
<td>160</td>
<td>3.193</td>
</tr>
</tbody>
</table>

Table 4.4: Data Transfer Times as a Function of Output Buffer Size.

<table>
<thead>
<tr>
<th>Output Buffer Size</th>
<th>Maximum Processors, N, for Reliable Transmission of 16KByte Packets</th>
<th>Elapsed Time (seconds)</th>
<th>Data Rate μs/Byte per proc</th>
<th>Link Utilisation (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>5</td>
<td>3.403</td>
<td>26.6</td>
<td>1.2</td>
</tr>
<tr>
<td>15</td>
<td>6</td>
<td>5.984</td>
<td>32.0</td>
<td>1.0</td>
</tr>
<tr>
<td>18</td>
<td>8</td>
<td>11.002</td>
<td>32.7</td>
<td>1.0</td>
</tr>
<tr>
<td>19</td>
<td>9</td>
<td>15.885</td>
<td>37.3</td>
<td>0.9</td>
</tr>
<tr>
<td>22</td>
<td>11</td>
<td>23.753</td>
<td>37.1</td>
<td>0.9</td>
</tr>
<tr>
<td>24</td>
<td>13</td>
<td>35.697</td>
<td>39.8</td>
<td>0.8</td>
</tr>
<tr>
<td>27</td>
<td>14</td>
<td>41.228</td>
<td>39.6</td>
<td>0.8</td>
</tr>
<tr>
<td>29</td>
<td>17</td>
<td>61.722</td>
<td>40.2</td>
<td>0.8</td>
</tr>
<tr>
<td>32</td>
<td>19</td>
<td>80.970</td>
<td>42.2</td>
<td>0.8</td>
</tr>
<tr>
<td>34</td>
<td>19</td>
<td>78.966</td>
<td>41.1</td>
<td>0.8</td>
</tr>
<tr>
<td>40</td>
<td>20</td>
<td>89.557</td>
<td>42.1</td>
<td>0.8</td>
</tr>
</tbody>
</table>

Table 4.5: Output Buffer Size for Prevention of Deadlock.

until the system did not deadlock. The results of this are shown in Table 4.5. The value of the data rate is determined by calculating the total number of packets involved from (4.1), multiplying this by 16384 for the number of bytes in one packet, dividing by the number of processors N, and dividing the result into the elapsed time. The link utilisation is determined as a percentage of the theoretical maximum data transfer over the intertransputer links. In the actual transputer network the links are set at 10 Mbit per second. From [153] the bi-directional data transfer rate is 800KByte per second, giving a theoretical maximum of 3.2MByte per second, and this figure was used to determine the final column in Table 4.5. The comparatively low values for link utilisation suggest that the rate at which data can be transferred through the network is not significantly restricted by the speeds of the inter-transputer links. From Table 4.5 there is a slight decrease in utilisation with increasing numbers of transputers in the network, and it is not clear to what this could be attributed since time is not a function of the output buffer size, and the sizes of the other buffers were kept constant for these tests. The results in
the table are subject to some variation as, in the vicinity of the boundary, the system would sometimes deadlock and sometimes not deadlock. The size of both the input and merged buffers was two for these measurements. Table 4.5 gives an indication of the buffer sizes for any application of a transputer network based on deadlock prevention. In specific cases the minimum size of the output buffers may be smaller, but this is best determined by steadily reducing them until deadlock occurs. Regression analysis applied to Table 4.5 gives

\[ \text{buf}_{\text{out}} = 1.60N + 4.03 \]  

(4.5)

with an RMS error of 1.8, and

\[ \text{Time} = 3.5\text{buf}_{\text{out}} - 44.8 \]  

(4.6)

with an RMS error of 7.9.

A combination of (4.3) and (4.4) yields

\[ \text{Time} = (0.056N - 0.24)\text{buf}_{\text{in}} + (0.049N - 0.19)\text{buf}_{\text{merged}} + (0.39N - 1.79) \]  

(4.7)

In (4.7) the coefficients of \( \text{buf}_{\text{in}} \) and \( \text{buf}_{\text{merged}} \) are sufficiently similar that they could be taken as being the same, yielding

\[ \text{Time} = (0.053N - 0.22)(\text{buf}_{\text{in}} + \text{buf}_{\text{merged}}) + (0.39N - 1.79) \]  

(4.8)

There are two points to be noted from (4.8). Firstly, the size of the input and merged buffers are of equal importance in determining the performance of the network. There are, however, four input buffers, so from the point of view of memory utilisation minimisation of \( \text{buf}_{\text{in}} \) is more important than minimisation of \( \text{buf}_{\text{merged}} \). Secondly, since the optimum performance of the link shell can be determined by setting the buffer sizes to zero in (4.8), the optimum performance for 16KByte data packet transfers is given by

\[ \text{Time} = 0.39N - 1.79 \]  

(4.9)

4.2.5 Logging of Link Communications

The final facility of the link shell to be described is the communications logger. This places the link identifier, the value passed, and the time into a data store which can be interrogated by means of a specified tag sent by the host. For checks of synchronization between transputers it is necessary for the clocks throughout the network to be approximately synchronized. This must be handled by the applications software. In the Transputer Image Processing System an initialization message from the host is broadcast to all transputers which initialize their clocks on receipt of the signal. There will thus be a slight time delay between one transputer and the next in the network. This time delay has been estimated to be less than one millisecond, and has not proven to be a problem.

It is worth noting that the problem of providing synchronized clocks is analogous to that examined by Einstein in his *Special Theory of Relativity*, and a similar solution applies
to the transputer network. Each network processor in turn sends a message to the host which returns the host's clock value. The network processor notes the elapsed time and uses half of this amount as the quantity to be added to the host's clock value to give the current time.

However, if a race condition exists at any network processor between the receipt of messages from two or more other processors, it is not certain that accurately synchronized clocks would help resolve the issue. The only reliable technique appears to be to design a system in which race conditions cannot occur.

4.3 A Transputer Image Processing System (TIPS)

Amongst a number of transputer products commercially available are IBM-PC compatible boards containing one or four transputers each. Figure 4.5 illustrates the system developed for this thesis. An IBM PC-AT (compatible) acts as a fileserver, providing keyboard entry, disk input/output, and screen output. An INMOS B004 single transputer board running the INMOS Transputer Development System (TDS) interfaces to the PC bus, converting appropriately-addressed data-bus bytes to a serial stream for passage on one of the transputer links. The transputer has access to 2 Mbyte of dynamic RAM via the local address and data buses. This transputer is the 'host' for the network.
One of the three remaining serial links is connected to one of five boards that are produced by CSA (Computer System Architects). Each of these boards contains four T414 transputers, each transputer having its own \( \frac{1}{4} \) MByte of dynamic RAM. Access is available to all four serial links for each transputer, so the number of possible ways in which the transputers can be interconnected is very large. The network, configured as a chordal ring, is illustrated in Figure 4.6.

![Figure 4.6: Chordal Ring Configuration for the Multi-Transputer System.](image)

The initial aim was to develop a basic image processing system able to handle images from the Department’s VIPS (VAX Image Processing System). However, the transputer development system on the IBM PC forced various alterations to the ideal of complete compatibility and images including header records must be converted to ASCII hex files before being accessed since non-printing ASCII characters have special interpretations for the TDS. The image is read into the host RAM before distribution over the network.

The remainder of this section is organised as follows. In section 4.3.1 the significance of the network configuration is examined. In section 4.3.2 the manner in which the software running on the host transputer controls the graphics and character attributes of the IBM-PC is described. The general objectives of the TIPS image processing system are described in Section 4.3.3, with the message protocols being described in Section 4.3.4. Some features of the image-processing software are described in Section 4.3.5. A partial listing of this software can be found in Appendix E. In section 4.4 the performance of the software is analysed and its application to automatic inspection is reviewed.
4.3.1 Network Configuration

A commonly used network configuration for transputer-based image processing systems is the toroidal mesh as illustrated in Figure 4.7. For instance, Cok [73] states that

No other network combines the simplicity and expandability of the toroidal structure.

Richards [239] uses a toroidal mesh of transputers for the study of neural networks, while Wylie [297] uses a four by ten torus for hydrodynamic modelling. Accordingly the message-passing efficiency of the toroid will be examined.

Let the nodes of a toroid be labelled as shown in Figure 4.8, where a three by three mesh is labelled using a ternary code. From this a table of distances can be formed, as shown in Table 4.6. A similar table could be established for any $n \times n$ toroid. The resultant pattern depends on whether $n$ is odd (as in Table 4.6) or is even. Let $m = n/2$ where the division is integer division. Then for $n$ even, row 0 of the table of distances is

$$0 \ 1 \ 2 \ 3 \ \cdots \ (m-1) \ m \ (m-1) \ \cdots \ 3 \ 2 \ 1$$
Table 4.6: Internode Distances for a Three by Three Toroid.

and row $i$, $i \leq n/2$, is

\[ i \ (i+1) \ (i+2) \ \cdots \ (m+i-1) \ (m+i) \ (m+i-1) \ (i+2) \ (i+1) \]

The sum of the $i^{th}$ row is given by

\[ S_i = \sum_{j=0}^{n/2} (i+j) + \sum_{j=1}^{n/2-1} (i+j) \]

75
\[ S_i = \frac{n^2}{4} + ni \]  

Now row \( j \) is the same as row \( (n - j) \), so

\[ S_i = \frac{n^2}{4} + n(n - i), \quad i > n/2 \]

Thus the sum for the network is given by

\[ S = \sum_{i=0}^{n/2} \left( \frac{n^2}{4} + ni \right) + \sum_{i=n/2+1}^{n-1} \left[ \frac{n^2}{4} + n(n - i) \right] \]

Using a change of summation variable for the second part by setting \( j = i - n/2 \), the sum becomes

\[ S = \sum_{i=0}^{n-1} \frac{n^2}{4} + \sum_{i=1}^{n/2} i + \sum_{j=1}^{n/2-1} [n(n - i)] \]

Using

\[ \sum_{i=0}^{p} i = \frac{p(p + 1)}{2} \]

the sum for the toroidal network becomes

\[ S = \frac{n^3}{2} \]

Since the total number of entries in the table is \( n^2 \), the mean internode distance is \( n/2 \).

A similar analysis can be performed for odd \( n \), and the results can be summarised as

\[ Mean = \begin{cases} \frac{n}{2}, & n \text{ even} \\ \frac{n^2 - 1}{2n}, & n \text{ odd} \end{cases} \]

The toroidal network is compared with the chordal ring network in Table 4.7. In the table, the chordal mesh refers to the chordal ring onto which a two-dimensional mesh can be mapped and with the mean internode distance minimised. This table reveals that there is a small but significant advantage in using chordal ring networks, especially if the optimal form of the chordal ring is used.

An alternative network has been described by Downing and Bennett [93]. They make use of a ternary tree, illustrated in Figure 4.9 as the basis of a multi-transputer network for image processing. If the number of levels in the tree is \( v \), then the number of nodes is \((3^v - 1)/2\). Thus the number of nodes increases rapidly with the number of levels. Furthermore, in the pure implementation of the tree each transputer in the final row only uses one of its four links, thus substantially increasing the mean internode distance for the network. However the performance would be substantially improved by joining transputers in the final row as illustrated by the dashed lines in Figure 4.9, formed from the additional links

\[
\begin{align*}
c &\rightarrow c + 4, \ c \mod 3 = 0 \\
c &\rightarrow c + 1, \ c \mod 3 \neq 2 \\
c &\rightarrow c + 3^{v-1} + 3, \ c \mod 3 = 2
\end{align*}
\]
<table>
<thead>
<tr>
<th>n</th>
<th>Number of nodes</th>
<th>Toroid</th>
<th>Chordal ring</th>
<th>Chordal mesh</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>9</td>
<td>1.3333</td>
<td>1.3333</td>
<td>1.3333</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>2.0000</td>
<td>1.8125</td>
<td>1.8750</td>
</tr>
<tr>
<td>5</td>
<td>25</td>
<td>2.4000</td>
<td>2.2400</td>
<td>2.4000</td>
</tr>
<tr>
<td>6</td>
<td>36</td>
<td>3.0000</td>
<td>2.7778</td>
<td>2.9167</td>
</tr>
<tr>
<td>8</td>
<td>64</td>
<td>4.0000</td>
<td>3.7188</td>
<td>3.9375</td>
</tr>
<tr>
<td>10</td>
<td>100</td>
<td>5.0000</td>
<td>4.6900</td>
<td>4.9500</td>
</tr>
<tr>
<td>12</td>
<td>144</td>
<td>6.0000</td>
<td>5.6458</td>
<td>5.9583</td>
</tr>
<tr>
<td>14</td>
<td>196</td>
<td>7.0000</td>
<td>6.5816</td>
<td>6.9643</td>
</tr>
<tr>
<td>16</td>
<td>256</td>
<td>8.0000</td>
<td>7.5195</td>
<td>7.9688</td>
</tr>
<tr>
<td>18</td>
<td>324</td>
<td>9.0000</td>
<td>8.4660</td>
<td>8.9722</td>
</tr>
<tr>
<td>20</td>
<td>400</td>
<td>10.0000</td>
<td>9.4200</td>
<td>9.9750</td>
</tr>
</tbody>
</table>

Table 4.7: Comparison of Mean Internode Distances.

Figure 4.9: Example of a Ternary Tree.

for the final row, where c is the column number. The performance of a few of these networks is summarised in Table 4.8. Downing and Bennett use the ternary tree for feature extraction for object recognition, and for this application the hierarchical nature of the network is useful. The ternary tree may also have advantages when applied to
### Table 4.8: Characteristics of Ternary Trees.

<table>
<thead>
<tr>
<th>Levels</th>
<th>Number of Transputers</th>
<th>Mean Internode Distance Pure Ternary Tree</th>
<th>Improved Ternary Tree</th>
<th>Chordal Ring</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4</td>
<td>2.00</td>
<td>1.33</td>
<td>0.75</td>
</tr>
<tr>
<td>3</td>
<td>13</td>
<td>3.00</td>
<td>2.79</td>
<td>1.54</td>
</tr>
<tr>
<td>4</td>
<td>27</td>
<td>4.47</td>
<td>4.41</td>
<td>2.98</td>
</tr>
<tr>
<td>5</td>
<td>121</td>
<td>6.21</td>
<td>6.11</td>
<td>5.16</td>
</tr>
</tbody>
</table>

In order to achieve a good performance for global operations the TIPS network has been configured as a chordal ring. There remains, however, the question of an appropriate chordal displacement. As explained in Chapter 3, a two-dimensional mesh will map onto a chordal ring if the displacement is a factor of the number of nodes in the ring. This will, in general, involve a small compromise in performance since the resultant choice of displacement is in general not optimal. In this section the question of whether the two-dimensional mesh is the best network for machine vision tasks such as kiwifruit sorting will be examined.

The acceptance of meshes is probably based on the fact that the relationship between adjacent image segments for global operations is made conceptually easier by mapping segments onto processors in such a way that adjacent segments lie on adjacent processors. The simplest such mapping is the two-dimensional mesh. A non-mesh configuration will mean that data transfer between adjacent image segments will not necessarily be between adjacent processors. The question is: does the added efficiency achieved by the adjacency of processors in a mesh configuration balance the loss of efficiency resulting from the larger mean internode distance? This can only receive a definitive answer with respect to particular algorithms. However it is to be noted that an application involving only point and global operations will perform better on an optimised chordal ring, while an application involving shape recognition is likely to be better suited to the mesh configuration.

Let the time per unit transfer over a link be \( t \). Let the total number of link transactions from a given processor be \( N \), of which a fraction \( \beta \) are nearest-neighbour transactions, assumed to be evenly divided between row and column, while the remaining \( 1 - \beta \) are global transactions. Let the network be configured so that nearest neighbours within a row map to adjacent processors in the network, and let nearest neighbours within a column map to processors \( c \) links apart, where \( c = 1 \) for the mesh network and \( c > 1 \) for non-mesh networks. Finally let \( g \) be the mean internode distance for the network. Then the total time for these transactions is

\[
T = N t \beta \left( \frac{1 + c}{2} \right) + (1 - \beta) g
\]

\[^{6}\text{See Chapter 8.}\]
where the term \((1 + c)/2\) is the mean number of link transfers for rows and columns combined. Compare two networks, the first having a mesh configuration with \(g = g_1\) and \(c = 1\), while the second is non-mesh and has \(g = g_2\). If \(T_1, T_2\) are the two transaction times then the requirement that the mesh network is faster than the non-mesh network, that is, \(T_1 < T_2\), means that

\[
\beta > 1/(1 + \frac{c - 1}{2(g_1 - g_2)})
\]  

(4.13)

Values of \(c\) for some chordal rings are given in Table 4.9.

<table>
<thead>
<tr>
<th>Rows</th>
<th>Columns</th>
<th>Nodes</th>
<th>Displacement</th>
<th>Inter-column distance, (c)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
<td>12</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>12</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>16</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>20</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>20</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>24</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>24</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>30</td>
<td>12</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>30</td>
<td>12</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>36</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>48</td>
<td>20</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>6</td>
<td>48</td>
<td>20</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>64</td>
<td>14</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>10</td>
<td>80</td>
<td>22</td>
<td>6</td>
</tr>
<tr>
<td>10</td>
<td>8</td>
<td>80</td>
<td>22</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>100</td>
<td>44</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 4.9: Some Values of Inter-Column Distance, \(c\).

Applying (4.13) to the TIPS network, the best configuration has a displacement of 8 (which is not a factor of 20), a mean internode distance of \(g = 2.08\), and a column distance for nearest-neighbour segments of \(c = 2\). The best mesh configuration has a displacement of 4 and a mean internode distance of \(g = 2.12\). This yields \(\beta > 0.074\), so that if more than 7.4% of all internode transactions are nearest-neighbour transactions the mesh network is likely to yield the best performance. Since the mean internode distances for the two networks are so similar the loss in performance in the case of all transactions being global would be very small. Accordingly the TIPS network has been configured as a chordal ring with a displacement of four.
4.3.2 Transputer Control of the IBM-PC Text and Graphics Screens

The use of the graphics and character attributes available on the IBM-PC considerably enhance the usefulness of the PC as the basis of a transputer system. In this section a facility is described that overcomes the limitations imposed by the standard INMOS products.

The TDS allows input from the keyboard, character output to the screen, and the handling of text files. In its standard form the system does not allow control of the attributes of displayed characters, access to other screen pages, or access to screen graphics modes. If an attempt is made to place the screen in graphics mode the TDS will reset it to text mode. Likewise access to text pages other than page 0 is blocked. A memory-resident program has been developed that gives full access to these features.

In the remainder of this section the features provided by this utility are described, the method by which they are implemented is explained, and the means for invoking these facilities from Occam are outlined. It is not necessary for the user of the system to know how these features are implemented as they are totally transparent.

Features Provided

The features provided in this utility are summarised in Table 4.10. The low and high resolution modes are the CGA (320 columns by 200 rows) and EGA (640 columns by 200 rows) standards respectively. In the pure graphics modes each point is addressed via its screen coordinates using the protocol

\[
\text{row.low; row.high; column.low; column.high; [colour]}
\]

The low and high parts of the row are the values of row \textit{mod} 256 and row \textit{div} 256 respectively, expressed as bytes, and sent to the screen as characters. The column value is treated similarly. The fifth term is present only for low-resolution graphics and sets the screen colour. To use graphics mode \textit{to.low.res} or \textit{to.high.res} is sent to the screen, followed by a stream of coordinates as given in the protocol. The screen will remain in graphics mode until an out-of-range coordinate is given, at which stage the system is returned to text mode.

The graphics facility described above is suited to producing line drawings, but the overheads make it very slow if a full picture is to be displayed. The alternative 'painting' facilities are implemented by means of one of the sequences

\[
\text{to.low.paint; row.low; row.high; column.low; column.high; count.low; count.high; four-points; four-points; four-points; ...}
\]
<table>
<thead>
<tr>
<th>Feature</th>
<th>Code</th>
<th>Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set low-resolution graphics mode</td>
<td>#FE (254)</td>
<td>to.low.res</td>
</tr>
<tr>
<td>Set high-resolution graphics mode</td>
<td>#FC (252)</td>
<td>to.high.res</td>
</tr>
<tr>
<td>Set low-resolution painting mode</td>
<td>#FB (251)</td>
<td>to.low.paint</td>
</tr>
<tr>
<td>Set high-resolution painting mode</td>
<td>#FA (250)</td>
<td>to.high.paint</td>
</tr>
<tr>
<td>Set character-attribute mode</td>
<td>#F9 (249)</td>
<td>to.attr</td>
</tr>
<tr>
<td>Set character-attribute mode (with cursor updating)</td>
<td>#F8 (248)</td>
<td>to.attr.up</td>
</tr>
<tr>
<td>Set character-attribute mode (with cursor updating and scrolling)</td>
<td>#F7 (247)</td>
<td>to.attr.sc</td>
</tr>
<tr>
<td>Set mixed low-resolution mode</td>
<td>#F6 (246)</td>
<td>to.low.mixed</td>
</tr>
<tr>
<td>Set mixed high-resolution mode</td>
<td>#F5 (245)</td>
<td>to.high.mixed</td>
</tr>
<tr>
<td>Set active-display-page mode</td>
<td>#F4 (244)</td>
<td>to.active.page</td>
</tr>
<tr>
<td>Set cursor-position mode</td>
<td>#F3 (243)</td>
<td>to.cursor</td>
</tr>
<tr>
<td>Set screen-clear mode</td>
<td>#F2 (242)</td>
<td>to.clear.page</td>
</tr>
<tr>
<td>Set continuous character-attribute mode</td>
<td>#F1 (241)</td>
<td>to.continuous</td>
</tr>
<tr>
<td>Clear continuous character-attribute mode</td>
<td>#F0 (240)</td>
<td>to.discontinuous</td>
</tr>
<tr>
<td>Set blanking mode</td>
<td>#EF (239)</td>
<td>to.blank</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Feature</th>
<th>Code</th>
<th>Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set low-resolution graphics mode</td>
<td>#FE (254)</td>
<td>to.low.res</td>
</tr>
<tr>
<td>Set high-resolution graphics mode</td>
<td>#FC (252)</td>
<td>to.high.res</td>
</tr>
<tr>
<td>Set low-resolution painting mode</td>
<td>#FB (251)</td>
<td>to.low.paint</td>
</tr>
<tr>
<td>Set high-resolution painting mode</td>
<td>#FA (250)</td>
<td>to.high.paint</td>
</tr>
<tr>
<td>Set character-attribute mode</td>
<td>#F9 (249)</td>
<td>to.attr</td>
</tr>
<tr>
<td>Set character-attribute mode (with cursor updating)</td>
<td>#F8 (248)</td>
<td>to.attr.up</td>
</tr>
<tr>
<td>Set character-attribute mode (with cursor updating and scrolling)</td>
<td>#F7 (247)</td>
<td>to.attr.sc</td>
</tr>
<tr>
<td>Set mixed low-resolution mode</td>
<td>#F6 (246)</td>
<td>to.low.mixed</td>
</tr>
<tr>
<td>Set mixed high-resolution mode</td>
<td>#F5 (245)</td>
<td>to.high.mixed</td>
</tr>
<tr>
<td>Set active-display-page mode</td>
<td>#F4 (244)</td>
<td>to.active.page</td>
</tr>
<tr>
<td>Set cursor-position mode</td>
<td>#F3 (243)</td>
<td>to.cursor</td>
</tr>
<tr>
<td>Set screen-clear mode</td>
<td>#F2 (242)</td>
<td>to.clear.page</td>
</tr>
<tr>
<td>Set continuous character-attribute mode</td>
<td>#F1 (241)</td>
<td>to.continuous</td>
</tr>
<tr>
<td>Clear continuous character-attribute mode</td>
<td>#F0 (240)</td>
<td>to.discontinuous</td>
</tr>
<tr>
<td>Set blanking mode</td>
<td>#EF (239)</td>
<td>to.blank</td>
</tr>
</tbody>
</table>

Table 4.10: Features of the Graphics Utility.

or

```
to.high.paint; row.low; row.high; column.low; column.high; count.low; count.high; eight-points; eight-points; eight-points; ...
```

The row and column coordinates refer to the top-left hand coordinates of the area to be covered, and the count (in the form of two bytes, calculated as \( \text{mod} \) and \( \text{div} \ 256 \) respectively) is the number of bytes being written. In the case of low-resolution painting the colour of each point is encoded as two bits (corresponding to four colours), and four of these are packed into each byte. In high-resolution painting there is no colour information and eight points are included in each byte, each bit corresponding to a screen pixel being off or on. The first character to be sent to the screen following the specified number of bytes returns the system to text mode.

The mixed modes provide a combination of painting and direct graphics. The top-left and bottom-right boundaries are specified, to provide a picture anywhere on the screen. Direct graphics can then be sent to draw, for instance, line graphs.

Single character attributes are established using the protocol

```
to.attr.up; character; attribute
```
where the attribute is a standard BIOS value. Some examples of values used in the Transputer Image-Processing System are shown in Table 4.11 where the # indicates that the codes are expressed in hexadecimal. Note that in Table 4.11 the only code with the high-order bit set is attr.warn. This is because this bit causes the character to flash, in the same way that the cursor normally flashes. Only one character-attribute can be established for each use of to.attr.up, so that three characters must be sent to the screen for each displayed character.

The alternative is to use the continuous character-attribute mode in which to.continuous is followed by the attribute. All subsequent characters then appear with this attribute. In producing formatted output using window-style screens, the ability to blank a window is provided by

```
to.blank; rows; columns; attribute
```

in which the screen rectangle of size rows by columns from the current cursor position is overwritten with spaces having the specified attribute.

Most graphics boards provide a number of text screens, and the active screen can be changed by sending to.active.page followed by the page number as a byte (normally 0 to 3). The active page can be cleared and the cursor position set by use of to.clear.page and to.cursor respectively. For the latter command the subsequent two bytes are the text row and column respectively.

### Implementation

The utility, named GR.COM, is a memory-resident program [241] occupying approximately 2K bytes in the PC’s memory space. Starting with the screen in normal mode, every character sent to the screen via a video interrupt is trapped by GR.COM. If the character’s ASCII code is one of those listed in Table 4.10, the appropriate action is initiated, otherwise the character is passed to the normal video service. The action will be illustrated in two cases — that of a character attribute and that of high-resolution graphics.
Character Attributes with Cursor Update

Receipt of to.attr.up results in a switch in GR.COM being set. The next two bytes are trapped in GR.COM. The first is placed in an even-numbered address in the regen (video memory buffer) area and is displayed as a character. The second is placed in an odd-numbered address and is interpreted as a bit-mapped code that establishes the character attribute (eight foreground colours, high or low intensity, eight background colours, and optionally blinking). The switch in GR.COM is cleared once the second byte has been received.

High-Resolution Graphics

Receipt of to.high.res results in any subsequent attempt to change the video mode back to text mode being blocked. A buffer sufficient for four bytes is cleared. The subsequent four characters sent to the screen are trapped and placed in the buffer which, when full, is used to calculate the screen coordinates (row, col). If either row or col is outside the allowable range the system is returned to text mode, otherwise the memory bit addressed by (row, col) is set and the buffer is cleared ready for the next group of four bytes.

Occam Invocation of the Graphics Features

In this section Occam code [150,184,222] is provided to illustrate the way in which GR.COM can be used. In this code use is made of the following procedures:

- PROC send (VAL BYTE char) which sends a single byte to the screen.
- PROC position (VAL INT row, col) which places the cursor at the specified location.
- PROC select.page (VAL INT page.number) which selects the given screen page.
- PROC char.attr (VAL BYTE character, attribute) which places the given character on the screen with the specified attribute and updates the cursor position.
- PROC write.at (VAL []BYTE str, VAL BYTE attr, VAL INT start.row, start.col) which writes an Occam-standard string to the screen starting at (start.col, start.row) and using the specified attribute (the screen is not scrolled and the string should not contain carriage-return).
- PROC plot.point (VAL INT this.row, this.col, VAL BYTE colour) which plots a point in the specified colour at the specified position in low resolution graphics.
An Example: A Line Drawing With Shading  As an example of the use of the 
routines described above consider the problem of joining two points on the screen with a 
straight line that may be obscured by a mask (representing a foreground). The mask is 
held in an array shade. This example uses a procedure

PROC abs (VAL INT a, INT absolute)

which returns the absolute value of a, and a procedure

PROC swap (INT a, b, c, d)

which swaps a and b if b ≥ a, in which case c and d are also swapped.

The first procedure below plots a point with shading, while the second uses this to produce 
a line. The shading arrays are updated.

PROC plot.point.shade (VAL INT this.row, this.col, VAL BYTE colour)
-- Plot a point in low-resolution graphics with shading.
SEQ
   IF
      this.row < shade[this.col]
   SEQ
      plot.point (this.row, this.col, colour)
      shade[this.col] := this.row -- update the shade
   otherwise
      SKIP -- don't plot
   :
-- plot.point.shade

PROC join.points.shade (VAL INT row1, col1, row2, col2,
                             VAL BYTE colour)
-- Join the given points, applying shading
INT r1, c1, r2, c2, r, c, r.diff, c.diff :
SEQ
   r1 := row1  -- Interval arithmetic requires that the
   c1 := col1  -- coordinates be available as variables
   r2 := row2
   c2 := col2
   abs (r2-r1, r.diff) -- Find the absolute row difference
   r.diff := r.diff+1 -- and adjust to include end points
   abs (c2-c1, c.diff) -- Find the absolute column difference
   c.diff := c.diff+1 -- and adjust to include end points
   IF
4.3.3 Objectives of the Image Processing System

The general objective of this work was to examine the ways in which image processing operations could be distributed over a network of processors, and to measure the performance of such a system. Although a number of multi-processor image processing systems have been developed in a number of locations around the world, especially over the last few years with the ready availability of processors such as the transputer, the descriptions in the literature tend not to describe the basic structure of the software. One exception is a system developed on a Meiko M40 Computing surface by Norman and Fisher [210]. This consists of a double buffered communications harness in which link loading is taken into consideration. They state that they provide sufficient buffers to allow the communications to work without deadlock, these buffers being structured in much the same manner as that used in the link shell described above (section 4.2). The optimum network routing is established by a network search algorithm in which every processor sends a message to every other processor, the identities of intermediate processors being appended to these messages as they pass through the network. On completion of this sequence of transfers, each network processor determines the shortest path to every other processor by counting the number of appended processor identities for each possible path. The authors say that the utilisation rate may exceed 50% in some circumstances.

The shell described in section 4.2 was used as the basis of TIPS. Since the software was seen as forming the basis of a development system, a comfortable form of user interface seemed preferable. Accordingly the command-driven approach used in VIPS was replaced with a menu-driven design. Initially a basic set of image-processing operations as shown in Table 4.12 was provided. In order to be able to obtain performance data for actual image-processing operations, timing information had to be provided. The total execution time consists of three parts: the time taken for an instruction to be distributed throughout the
network, the time taken for the instruction to be executed on each of the processors, and finally the time taken for the messages acknowledging the completion of the instruction on each of the processors to be received and tallied by the host. Accordingly it was necessary to be able to find the value of the second of these three times as any likely industrial transputer-based system would have the instructions permanently distributed in the network. This has been achieved by allowing instructions to be iterated any number of times, thus permitting nett instruction times to be derived. An enhancement of the present system would allow a sequence of image-processing operations to be pre-loaded on the network, and this would provide a realistic measure of the performance of a transputer network on a routine task. Since a major objective is performance some code optimisation has been carried out but the link shell remains a major bottleneck.

### 4.3.4 Message Protocols

The operation of the full network (including the host) is based on the use of a standard message protocol using the format:

```
destination.processor; number.of.values; [;values]; end.of.message
```

The value of `destination.processor` can take any of the values

- 0 to 19 for the network processors
- `end.proc` for the final processor
- `host.proc` for the host processor
- `broadcast` to distribute throughout the network

where `end.proc`, `host.proc`, and `broadcast` are reserved integers that are well outside the range of sizes of any actual transputer networks. In the message protocol the `end.of.message`
is also a reserved integer \((= 2^{30} - 1)\). The sequence of \textit{values} must contain at least one member, and the first member must be a tag which determines the interpretation of the remainder of the message. Currently TIPS contains some 65 tags, and seven of these will be used as the basis for characterising the system. The first part of each tag designates the direction: \textit{hn} for host to network, \textit{nn} for network to network, and \textit{nh} for network to host.

### Load an Image

The protocol is

\[
\text{hn.load.image; image.number; rows; cols; [pixels]}
\]

This instruction is sent to each processor in turn, designating the identity of the image and the number of rows and columns including borders for this segment (see below). The border region, which has a width of one pixel in the system as implemented, is sent as part of the image segment, and most distributed operations do not need any information concerning the border. In the protocol, the square brackets indicate that the contents, namely \textit{pixels}, are repeated. In this case the number of pixels is \(\text{rows} \times \text{cols}\), and the total number of values sent with the message is the number of pixels plus four.

### Send an Image to the Host

The protocol is

\[
\text{hn.send.image; image.number}
\]

When the host requires an image for display or saving to disk this message is sent to all processors. In this case the instruction is not specific to a particular processor and a special destination is given, namely \textit{broadcast}. The effect of this is illustrated in Figure 4.10. A processor in the first row of the array sends the message down the column, and to the next processor in the first row, unless this column is the last one. Other processors send the message down the column. This distributes the message through the network in an efficient manner. This instruction requires that the processors know about their neighbours in the network (see below).

### Provide Information on Location in the Network

This instruction is sent to each processor in turn, and the protocol is

\[
\text{hn.neighbours; L.H.S.; R.H.S.; top; bottom; number.of.columns; row.number; column.number; centre(BOOL); single.column(BOOL)}
\]

where \textit{L.H.S.} is the processor number of the transputer immediately to the left of the current one, and so on. A special processor number is reserved for \textit{no.neighbours}, used for
one or more of the neighbours of edge processors. The number of columns in the network, and as currently configured this will be either four (in which case the Boolean single.column will be false), or one (in which case single.column will be true). The location in the network is given by row.number and column.number, and the Boolean centre is true if

\[
\text{column.number} = \text{number.of.columns \ div \ 2}
\]

where div designates integer division. The centre processor for each row plays a special function in forming the distributed convex hull (see below).

A Network Processor has Finished Loading an Image The protocol is

```
nh.finished.load.image; image.number; processor
```

This message is sent by each network processor to the host on receipt of an image segment. It provides confirmation for the host that the network is performing correctly and as currently implemented is used by the host as a signal that the next segment can be sent.
Network Response to Testing  This message arises in response to the host message
hn.test which is sent to a special processor called end.proc. The network configuration
indicates the location of the last transputer in the chordal ring. On receipt of a mes-
sage with a destination of end.proc, a processor will send it on to the next processor
circumferentially in the ring unless the Boolean last.transputer is true. The protocol is

hn.test.response; processor; day; month; year; displacement

The contents of the message indicate the number of processors in the network, indepen-
dently of the way in which the chordal links have been configured or of the way in which
an array has been mapped onto the network. The date is updated every time the net-
work software is changed. The displacement is the chordal displacement of the network
as currently configured. This value is passed as a parameter from the configuration code
to the code resident on each network transputer.

Request an Image Segment  This is used by the move instruction and involves the
interchange of parts of images between network processors. The protocol is

nn.request.image.segment; requesting.processor; source.image;
row.start; column.start; row.step; column.step; border;
destination.image; destination.row.start;
destination.column.start

The destination processor is designated in the message header, and the portion of the
image in that processor is designated by the position of the lower-left corner and by the
size of the segment. The destination information in the final two lines of the protocol
is not required by the processor providing the segment but is returned as part of the
segment (see below) and indicates to the requesting processor what should be done with
the message. This is because the procedure handling the message which returns the image
segment is quite independent of the requesting procedure, and there is no communication
between them. This is a general requirement of the link shell, and is characteristic of a
number of the functions implemented.

Send an Image Segment  This is in response to nn.request.image.segment. The pro-
tocol is

nn.send.image.segment; source.processor; destination.image;
row.start; column.start; row.step; column.step; border;
destination.start.row; destination.start.column;
number.of.pixels; [pixels]
The number of pixels is \( \text{row.step} \times \text{column.step} \), and the total number of values sent in this message is \( \text{number.of.pixels+11} \).

4.3.5 Some Particular Features

A number of features of the transputer image processing system will be described in this section.

**Loading an Image Onto the Host**  The transputer development system (TDS) reserves certain characters for file control, in particular the features of the folding editor\(^7\). This means that binary files cannot be read, and all image files must be converted into hexadecimal files, with two printable characters in place of each binary byte. The TDS provides facilities for manipulating files from within an Occam program, and the sequence involved is as follows:

1. Create an empty fold within the current fold structure. I have provided a dummy fold for this purpose so that the fold containing the executable code for the host transputer cannot be corrupted (if this fold is altered in any way the TDS considers it to be uncompiled, so that the compiler must be invoked before re-running).

2. Name this fold with the name of the image file. The file extension must be .tsr, and this is automatically supplied.

3. This named fold is filed, and the TDS, on finding that the file exists in the current directory, makes a copy of that file on the disk. It is that copy which is referenced by the TDS.

4. The fold is opened. Then the file is opened, read into the host, and closed. Finally the fold is closed.

5. The file is deleted from the disk by deleting the fold, so that the system returns to its original state.

This sequence is long, cumbersome, and inefficient, and it is to be hoped that upgrades to the TDS will overcome some of these problems. The operation for saving an image is similar to the above, with the important difference that the filed fold is not deleted, so that the system is not returned to its original form.

The image format is as follows. The header record consists of

\[
\text{rows} \quad \text{columns} \quad \text{CR/LF}
\]

\(^7\text{See Appendix D.}\)
where \( \textit{rows} \) and \( \textit{columns} \) are each four bytes, and \( CR/LF \) is two bytes. Each remaining row of the image is represented as

\[
\text{pixel1 pixel2 pixel3 ... CR/LF}
\]

where each pixel is two bytes and there is a maximum of 64 pixels in each row of the record. This is because the TDS can handle records with a maximum of 128 bytes, and for images of 100 by 100 pixels each row must be divided into two rows of data.

**Image Segmentation** Low-level image processing operations are conveniently divided into point, neighbourhood, and global operations. The manner of segmentation of an image is of no significance in regard to point operations. The significance of network architecture to non-local operations has been examined in section 4.3.1. The case of neighbourhood operations is worth additional consideration. Consider three by three window operations. Along the boundary of an image segment such an operation will require reference to pixels in adjacent processors (diagonally adjacent in the case of corner pixels). A useful simplification is to segment the original image using overlapping segments as illustrated in Figure 4.11. Operations are carried out on the border elements for all point and neighbourhood operations. Global operations such as convex hull and image rotation must, however, proceed without the border. Morrow et al [198] have used a similar scheme in their transputer-based image processing system. However, they establish the borders once the image has been distributed.
The cost in terms of loss of efficiency in the scheme used by TIPS is

\[
Loss = \frac{2s_r b + 2s_c b + 4b^2}{s_r s_c}
\]

for an image segment of size \( s_r \) rows and \( s_c \) columns, based on a border width of \( b \). For instance, segmenting an image of size \( 100^2 \) pixels over an array of five rows by four columns of transputers, and using a border width of one pixel, the loss of efficiency is 18.8%. In the current implementation of TIPS this loss is justified because of the considerable overheads involved in link communications. In alternative implementations the overhead would need to be reassessed.

**Distribution of Images Over the Network**  The host is connected to the network by a single link, and this forms a significant bottleneck as far as the transmission of images between the host and the network is concerned. Many schemes for image distribution were considered, and the most efficient is only likely to be found by experimentation. The network processors can be given an active role in the distribution process, or can be made into simple receptors. A promising scheme would involve the host sending the image as a continuous stream, with each network processor being responsible for extracting its own segment plus an appropriate border and passing on the image to subsequent processors. Although this scheme is attractive from the point of view of the host, it is not clear whether the network would be able to accept the data values at the rate at which they would become available from the host.

The scheme adopted involves the host sending out an image segment plus border to each processor in turn, and waiting for that processor’s acknowledgment before continuing. One advantage of this scheme is that it minimises the requirement for data buffers in each network processor. Two versions of TIPS have been developed. In one of these the images are stored on the network processors as byte arrays, with one pixel per byte, while in the other images are stored as integer arrays. The latter scheme is more costly in terms of memory, but is faster and more convenient as byte values must be converted to integer values for any arithmetic operations.

**Logical Operations**  A class of point operation on one image or between two images is the set of Boolean or logical operations. In the current implementation of TIPS these consist of AND, OR, XOR, and NOT. The T414 transputer used in TIPS uses a 32-bit word size, and the Occam facility for RETYPING variables has been used to effect a significant speedup in the use of logical operations on byte images. The code used in implementing the AND operation is as follows:

```plaintext
groups.of.4.cols := (cols+3)/4 -- number of groups of four columns
SEQ r = 0 FOR rows
    [image.buffer.cols]BYTE im1 RETYPES image.buffer[image1][r] :
    [image.buffer.cols]BYTE im2 RETYPES image.buffer[image2][r] :
```

92
SEQ c = 0 FOR groups.of.4.cols
VAL boundary IS 4*c:
INT chunk1 RETYPES [im1 FROM boundary FOR 4] :
INT chunk2 RETYPES [im2 FROM boundary FOR 4] :
chunk1 := chunk1\chunk2

The image segments image1 and image2 are of size rows by cols. The original image buffer is referred to by expressions of the form image.buffer[i][r][c] where i is the image number, r is the row number, and c is the column number. The image buffer is thus a triply-dimensioned BYTE array. The variables im1 and im2 are coincident with the original image buffer but are singly-dimensioned. The chunk variables RETYPE groups of four bytes as single integers, and the bit-wise AND in the final line is performed effectively four bytes at a time.

**Move an Image** This operation involves the passage of image segments between neighbouring processors. Referring to Figure 4.12, a general move will involve three image rectangles. With positive row and column shifts as illustrated, rectangle (1) comes from the processor in the same column and the previous row, rectangle (2) comes from the processor in the previous row and previous column, while rectangle (3) comes from the processor in the same row but previous column. For processors on the edge of the processor array null images must be substituted. The size and positions of the rectangles are based on the size of the row and column shifts. When an instruction to move an image

![Figure 4.12: The MOVE Instruction as a Transfer of Three Rectangles.](image-url)
is broadcast through the network, each processor determines the boundary coordinates in both source and destination processors, and the identity of the source processors, for each of the three image rectangles. Three instructions requesting an image segment are broadcast (see above). Since these requests contain the full information for placement of the image rectangles no further action is required on the part of the move instruction. Each processor replies by sending the appropriate rectangle (see above), and these segments contain destination information which enables the receiving processor to place the segments appropriately. The move instruction must take account of the boundary surrounding each full segment, and this is not involved in the move.

Rotate an Image  Referring to Figure 4.13, the operation of rotating an image through $90^\circ$ involves dividing the image into squares such that the boundaries of the squares have row and column values of $x$ where

$$x \mod r_s = 0$$
$$x \mod c_s = 0$$

(4.14)

where $r_s$, $c_s$ are the number of rows and columns in each image segment. Each processor determines the set of solutions to (4.14) that lies within its segment. The resultant squares will form the destinations of image squares from other processors in the array, and requests are transmitted to the appropriate processors. Each square is rotated before being returned. As with the move instruction, these requests contain full destination information which allows the image square to be correctly placed without further reference to the rotate instruction.

Figure 4.13: Rotation of an Image.
Expand an Image  The ability to expand the intensity range of an image to occupy the full range (0 to 255 in the case of 8-bit images) is useful in software development as it makes images easier to view and provides a means of standardisation for images provided under varying lighting conditions. The TIPS implementation of this instruction proceeds in two stages. Firstly the host transmits a request to all processors to return the local maximum and minimum pixel intensities of the specified image. The host then determines a global maximum and minimum, which are broadcast to the network as part of the expansion command. The border region is not included in assessing the extreme values, but is involved in the resultant expansion.

Convex Hull  Downing and Bennett [93] report the development of a convex hull algorithm for a distributed transputer network. However at the time of the publication of their paper they do not appear to have the distributed form of the algorithm operational. The distributed convex hull implemented on TIPS is a row-based procedure which replaces the intensity profile of a complete row with its convex envelope. The addition of column data to provide a full convex hull would add considerably to the complications. According to McNeill [188] the row-based convex hull is adequate for surface blemish detection in kiwifruit. The intensity envelope is pegged at zero intensity levels, as shown in Figure 4.14. Each row of transputers has one transputer designated as the centre (the second one in the case of four transputers per row). Only the centre transputers respond to the hull instruction. For each image row in turn the centre sends requests to each

![Figure 4.14: Distributed Convex Hull.](image)
processor in the same row of processors, resulting in the appropriate image row being returned. The centre processor forms the convex hull of this row and redistributes this within its own row of processors. This sequence of operations is illustrated in Figure 4.15 for a single row of processors.

![Diagram of operations for the Distributed Convex Hull](image)

Figure 4.15: Sequence of Operations for the Distributed Convex Hull.

This sequence of operations is performed in every row of processors in parallel.

### 4.4 Performance of the System

In this section the performance of the system will be examined. There are two aspects to this. Some of the characteristics of Occam as implemented on the transputer are studied in Section 4.4.1. One particular characteristic, that of channel slices, is investigated further in Section 4.4.2. Finally, the performance of the image processing software is reported in Section 4.4.3.
4.4.1 Some Aspects of Occam on the Transputer

Valkenburg [285] has examined the way in which Occam instructions compile for running on the transputer. He reports that the use of internal RAM leads to a considerably improved performance since no prefix instructions are generated, leading to very compact code. Prefixes are used to allow the operand to be expanded to any length, and the number of prefixes depends, amongst other factors, on the distance of memory references from the base address. Thus the most frequently referenced variables should reside in internal memory.

A major problem with establishing timings is that the compiled code produced depends on the actual position of the code in memory. For instance, if the code for the simple loop

```plaintext
SEQ rep = 0 FOR repetitions
   x := 1
```

is repeated in a program, the timings for each invocation can differ by several percent. This is related to the way in which the transputer uses prefixes, especially when establishing large constants in the code (see, for instance, Valkenburg [285]). In order to overcome this problem the relevant code was placed in a variety of locations in a program to obtain a number of timings. It seems likely that with a large program these effects would tend to smooth out, since these code sections are likely to be scattered through the software. Some basic operations that occur frequently in image processing were analysed, and are reported in the following paragraphs.

Channels Let $c$ be a 'soft' channel, that is, one that exists within a single transputer. Compare the two blocks of Occam code:

```plaintext
PAR
   c ! x
   c ? y
   y := x
```

The effect of each is the same. However the former takes nearly five times as long as the latter. The use of channels is an important feature of the concept of communicating sequential processes [136], which forms the basis of Occam. It is extensively promoted in the INMOS literature as a practical technique for passing values between variables, and the link shell used by TIPS makes major use of channel buffers. As can be seen from the result quoted above, channels are computationally expensive. This has also been noted by Winder [293].

Array references The time taken for $a[b] := x$ where $b = 3$ and $x = 5$ is approximately $55\%$ longer than $a[3] := 5$. The sequence
SEQ \( i = 0 \) FOR \( n \)
\[
a[i] := i
\]
takes over three times as long as the equivalent list of assignments (that is, \( a[0] := 0, \)
\( a[1] := 1, \) and so on) for \( n = 2 \), and increases steadily as \( n \) increases, becoming over six
times as long for \( n = 20 \). Thus in speed-critical sections of code replicated assignments
should only be used if either the number of replications is variable or the number of
replications makes the alternative approach impractical.

**Array slices** Given [100]INT \( a, b \), then

\[
[a \text{ FROM 0 FOR 100}] := [b \text{ FROM 0 FOR 100}]
\]
runs approximately eight times as fast as

\[
\text{SEQ } i = 0 \text{ FOR 100} \\
\quad a[i] := b[i]
\]

As noted by Winder [293] a similar comment applies to channel slices. Since channels
provide the basis of the link shell buffers the study of channel slices is pursued in greater
detail in the next section (see Page 101).

**Retyping of arrays** The technique outlined above can be combined with *retyping* [150]
to effect an important speedup in references to multi-dimensional arrays. Given [100]INT \( q \)
\( \text{RETYPES } [p[y] \text{ FROM 0 FOR 100}] \), then \( q[i] := z \) runs typically 30% faster than
\( p[y][i] := z \). This is the technique employed in Section 4.3.5. For a 15MHz T414
transputer the actual time per array reference has been measured as 5.2\( \mu s \), although this
time increases by about 5\% for each additional nibble involved in the retyping. Thus the
dimension of the retyped array should be made as small as possible.

My experience with retyping has lead me to treat this technique with some caution, as
frequently there are no time savings. It would seem that in many cases the compiled
Occam code is fairly well optimised. For instance, the former of the following two blocks
of code takes 12\% longer than the latter:

\[
[100][100]\text{INT } a : \\
\text{SEQ } p = 0 \text{ FOR 100} \\
\quad a[p][3] := 7
\]
\[
[100][100]\text{INT } a : \\
\text{SEQ } p = 0 \text{ FOR 100} \\
\quad a[3][p] := 7
\]
BYTES versus INTegers  There is some cost in defining variables as byte arrays, especially when arithmetic is involved since the array elements must then be converted to integers before any arithmetic operations. If the result of an arithmetic operation is to be stored in a byte array not only must it be converted to a byte value but it should be checked first to ensure that it is in the range [0···255]. Compare the following blocks of Occam code:

\[
\begin{align*}
\text{[1000]BYTE im :} & \quad \text{[1000]INT im :} \\
\text{SEQ i = 0 FOR 1000} & \quad \text{SEQ i = 0 FOR 1000} \\
im[i] := \text{BYTE}((\text{INT } im[i])-10) & \quad \text{im[i]} := \text{im[i]}-10
\end{align*}
\]

Both have the effect of subtracting ten from each element. The first takes approximately 15% longer than the second. If byte arrays are used then a useful saving is obtained by pre-declaring constants. Comparing

\[
im[i] := \text{BYTE 0} \quad \text{VAL BO IS BYTE 0:} \\
im[i] := \text{BO}
\]

the first takes approximately 12% longer than the second. The first of these assignments took 9.1\(\mu\)s on a 15MHz T414.

Logical operations  For byte arrays, a simple way of performing logical operations is illustrated by the second of the following blocks of code, which both operate on \([100]\) BYTE im1, im2.

\[
\begin{align*}
\text{SEQ i = 0 FOR 100} & \quad \text{[25]INT image1 RETYPES im1 :} \\
im2[i] := im1[i] \text{\textbackslash im2[i]} & \quad \text{[25]INT image2 RETYPES im2 :} \\
\text{SEQ i = 0 FOR 25} & \quad \text{SEQ i = 0 FOR 25} \\
\text{image2[i] := image1[i] \textbackslash image2[i]}
\end{align*}
\]

The former takes 4.5 times as long as the latter.

Conditionals versus array references  A technique provided in TIPS for fast thresholding (refer to the program listing in Appendix E) replaces the test based on \textit{IF} with an array reference. It is based on the proposition that an operational machine vision system will have a set value of threshold, so that the array, declared as \([256]\) INT thres in the following listing, will have been initialised before processing commences. Compare the following blocks of code:

\[
\text{IF} \quad y := \text{thres}[x]
\]

99
The latter runs approximately 10% faster than the former.

**Some alternative conditionals**  Consider the following blocks of code, which are equivalent:

```plaintext
IF
  c = 1
  b := TRUE
  c = 2
  b := TRUE
otherwise
  SKIP

IF
  (c=1) OR (c=2)
  b := TRUE
otherwise
  SKIP
```

The former takes 10% longer than the latter if the condition branches occur with equal probability. The following blocks of code are equivalent, but the former takes some 19% longer than the latter:

```plaintext
IF
  x = 0
  b := TRUE
otherwise
  b := FALSE
```

**Other operations**  Various other efficiencies are easily achieved. For instance, \( x/4 \) takes over twice as long as \( x >> 2 \), where \( >> \) means shift right. Similarly, \( y := x*65536 \) takes 34% longer and \( y := x \times 65536 \) takes 10% longer than \( y := x<<16 \).

**Compiler options**  The only compiler option that generates additional code is *range checking*. Consider the following blocks of code:

```plaintext
[100] INT a :
  a[77] := 777

(INT)b := 43
(BYTE)bb := BYTE b
```

With range checking enabled the second block takes 9% longer to run than with range checking disabled, but there is no discernible difference in the case of the first block.
4.4.2 Channel Slices

As noted above, array slices and channel slices run considerably faster than the equivalent discrete reference. Since channel buffers are a critical component in the link shell, a study was made to estimate whether the use of channel slices would provide a useful speedup. The first problem to be noted with the use of slices is that the size of the slices must be a constant that has been determined at compile time. Thus such a system will not accommodate variable length messages. Accordingly two buffer strategies were compared:

1. Variable length messages are passed through channel buffers one integer wide.

2. Fixed length messages are passed through channel buffers of unit length and of width equal to the length of the messages.

Strategy number one is that outlined by Pountain [222] and since it is the easiest to implement it was the one adopted as the basis of the link shell used by TIPS.

Consider strategy number two. If the system is to cope with variable length messages then messages longer than the size of the buffer must be subdivided while shorter messages must be packed with dummy values. All messages can be divided into a header/trailer and a body. In the case of the TIPS format the header consists of destination; number; tag and the trailer consists of end of message. In addition certain components of the message may need to be repeated if the message is subdivided. The header/trailer thus consists of at least four integers. Let this number be represented by $h$.

If message $i$ has a length of $m_i$ then the body is of length $m_i - h$. Let the fixed size of the channel buffer be $f$. Then message $i$ must be divided into

$$ s_i = \lceil \frac{m_i - h}{f - h} \rceil $$

submessages, where $\lceil \cdot \rceil$ indicates that the result of the division must be rounded upwards. Let the time per word for buffer strategies one and two be $t_1$ and $t_2$ respectively. Let message $i$ occur with probability $p_i$. Then the mean time per message using strategy one is

$$ \tau_1 = t_1 \sum_i p_i m_i $$

and for strategy two is

$$ \tau_2 = f t_2 \sum_i p_i s_i $$

where summation is over all messages. Time $t_2$ is a function of the buffer width $f$, and the expression

$$ t_2 = \frac{a}{f} + b, \quad a \approx 16.22\mu s, \quad b \approx 0.6917\mu s $$

101
has been found to be a good approximation, with a correlation coefficient of 0.999993 for \( f \in [10, 100] \) in the case of a 15MHz T414 transputer. Thus

\[
\tau_2 \approx (a + b f) \sum_i p_i s_i
\]  

(4.19)

Measured values of the ratio \( \tau_1/\tau_2 \) for messages whose length is set equal to the buffer size are shown in Table 4.13. As can be seen strategy two offers a considerable speedup.

<table>
<thead>
<tr>
<th>Buffer size</th>
<th>Ratio of times</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>13.2</td>
</tr>
<tr>
<td>40</td>
<td>27.4</td>
</tr>
<tr>
<td>100</td>
<td>35.4</td>
</tr>
<tr>
<td>400</td>
<td>41.0</td>
</tr>
<tr>
<td>1000</td>
<td>42.3</td>
</tr>
</tbody>
</table>

Table 4.13: Time Ratio as a Function of Buffer Size.

In the remainder of this section the optimum buffer size will be investigated.

Referring to equation 4.19, if the probability distribution \( p \) of messages is known, the value of \( f = f_{opt} \) which minimises \( \tau_2 \) can be calculated. Since the calculation of the \( s_i \) involves integer arithmetic, an analytical solution for the buffer length \( f \) cannot be obtained and the easiest way of determining \( f_{opt} \) is to perform a numerical search. For instance, suppose there are two message lengths, one of ten words which occurs with probability \( p_{10} \), and the other of 100 words which occurs with probability \( p_{100} = 1 - p_{10} \). Suppose that the header/trailer consists of five words. Then the value of \( f_{opt} \) as a function of the probability of occurrence of the messages of length ten is as shown in Figure 4.16. If, as is the case with typical image-processing algorithms run on TIPS, shorter messages predominate, so that \( p_{10} \) is close to 1.0, then the optimum buffer size varies rapidly with \( p_{10} \). However, if the dominant message traffic involves passing image segments, so that \( p_{10} \) is relatively small, say \( p_{10} < 0.4 \), then the optimum buffer size is equal to the length of the longest messages, which is 100 in the example above.

Since the probability distribution of messages is, in general, known for machine vision tasks such as kiwifruit inspection, the optimum buffer size can be established by the techniques outlined above.

### 4.4.3 Image Processing Performance

The performance of the system was measured with respect to some of the image-processing operations that are important in the detection of surface blemishes in kiwifruit. The measurements were made by having each network transputer measure the time taken to perform the operation, so that the time required to set up the operation on the network and to collate the network responses could be excluded from the final time. Any machine
vision application would presumably follow a set algorithm, so that the large amount of communication between the host and the network in TIPS would be very much reduced. The results of these time measurements are presented in Table 4.14, the first figure representing the mean time and the second the maximum time. The final row in the table contains the times quoted by McNeill [188] for these operations running on the single-processor HIPS\(^8\). The distributed form of the convex hull cannot be measured by this technique, and the elapsed time as measured by the host is very much longer because of

\(^8\)High-resolution Image Processing System.
Consider two possible surface blemish detection algorithms, the first based on a hardware implementation of the rank filter, the second implementing the rank filter in software.

Algorithm A is:

subtract background, convex hull, subtract images, threshold/area

while algorithm B is:

subtract background, rank filter, convex hull, subtract images, rank filter, threshold/area

From Table 4.14 the expected run time can be estimated for the single-column network configurations. The results of these estimates are given in Table 4.15. Considering the figures in Table 4.15 for a single column of transputers, the times for execution of algorithm A can be approximated by

\[ Time_A = \frac{1400}{N} \text{ milliseconds} \]  

### Table 4.14: Times for Some Image-Processing Operations.

(The figures given are the mean and maximum times in milliseconds, and are for 100 by 100 images).

<table>
<thead>
<tr>
<th>Network</th>
<th>Image-Processing Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows</td>
<td>Columns</td>
</tr>
<tr>
<td>---------</td>
<td>---------</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>(1)</td>
<td>(1)</td>
</tr>
</tbody>
</table>

\(^{(1)}\)
<table>
<thead>
<tr>
<th>Transputer network</th>
<th>Time (milliseconds) for algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows</td>
<td>Columns</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>20</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4.15: Estimated Times for Kiwifruit Surface Blemish Detection Algorithm.

Similarly the execution times of algorithm B can be approximated by

\[ Time_B = \frac{2700}{N} \text{ milliseconds} \]  \hspace{1cm} (4.21)

where \( N \) is the number of transputers. The numbers of transputers required to achieve four fruit per second and twenty-five fruit per second based on the results above are shown in Table 4.16, with any inter-transputer communications overhead and image load time being excluded. Ideally such overheads would not exceed, say, 10%.

<table>
<thead>
<tr>
<th>Fruit Views per Second</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm</td>
</tr>
<tr>
<td>A</td>
</tr>
<tr>
<td>B</td>
</tr>
</tbody>
</table>

Table 4.16: Minimum Numbers of Transputers Required for Kiwifruit Blemish Detection.

4.5 Conclusions

An image-processing facility has been described. This is based on a network of transputers configured as a chordal ring. The software is based on a link shell which makes the details of the network configuration invisible to the applications programmer and to the system user. The main features of this shell are as follows.

- A set of link buffers is provided to allow deadlock-free message passing.
- The imposition of a specific message format provides consistency checks.
- A message log is provided as an aid to software debugging.

A range of image-processing operations have been implemented, providing experience in the writing of distributed software. These make use of a specially-written utility that
allows the PC character attributes and graphics screens to be controlled by the host transputer from within the transputer development system.

The performance of individual sections of Occam code have been analysed and some of these have been used to improve the efficiency of the image processing system. Some measurements of the performance of the image processing system on the network of transputers have been obtained. These measurements have been used to estimate the performance of a transputer network devoted to machine vision, suggesting that the objective of running the kiwifruit surface blemish detection algorithm in less than a quarter of a second could be achieved on a network of ten transputers if the rank filter was implemented in hardware, and on a network of twenty transputers if the rank filter was implemented in software. Issues such as fault tolerance have not been studied. The network of twenty transputers used in this study is fairly susceptible to electrical noise, especially spikes or surges over the power supply lines, and any industrial system would need to include some hardening against noise.
Chapter 5

Mapping Image-Processing Operations onto Transputer Networks

5.1 Introduction

Image processing as considered in this chapter consists of operations on discrete picture elements or pixels. An image commonly consists of a rectangular array of pixels, each pixel consisting of one or more bytes. Industrial applications of image processing, which might typically involve the inspection of an object on a conveyor belt, must be carried out within a time interval dictated by the speed of the industrial process. The processing power needed to achieve these speeds is typically in the hundreds of MIPS (millions of instructions per second), and networks of processors are a means of attaining these speeds. The transputer is a processor that is well-suited to constructing multiprocessors for image processing since each transputer can communicate with its four nearest neighbours. An image divided into regular rectangular segments will thus map directly onto a transputer array.

Image processing operations can be categorized as point, neighbourhood, and global [225]. When a point operator is applied to a picture the resultant value of any given pixel depends only on the value of that pixel. An example of such an operator is image thresholding in which pixels are replaced by ‘black’ or ‘white’ depending upon whether they are below or above some specified threshold. Neighbourhood operators involve a window around each pixel so that the resultant value depends on the values of the pixels in the immediate vicinity. A filter operating on an $n$ by $n$ window is an example of a neighbourhood operator. With global operators such as the Fast Fourier Transform or FFT [74] every output value may depend on the value of every pixel in the picture. The result of applying a global operator may be another image or may be some other data structure.
Fine grain parallelism can be applied to local operators since, given sufficient resources, one transputer could be applied to each pixel in the image, with each transputer working entirely independently. Neighbourhood operators are medium-to-fine grain; for example a 512 by 512 pixel image mapped onto a network consisting of an array of sixteen transputers results in a subimage of 16K pixels on each transputer. Only around the boundaries of the subimages will transputers need to interact, and even this can be avoided by overlapping the subimages. Thus both local and neighbourhood operators map easily onto a network of transputers. The case of global operators is often more complicated. For example, although parts of the FFT algorithm provide the opportunity for fine-grain parallelism, there is no task-to-network mapping that avoids the need for large-scale inter-transputer communications. An algorithm for the parallel computation of one-dimensional FFTs has been published by Kirk and Verly [163].

A stochastic model for evaluating the performance of parallel programs has been proposed by Gelenbe et al [115]. They use simulation to validate their results.

The objective of the work described in this chapter is to formulate techniques by which the performance of transputer networks applied to image processing can be estimated. The maximum difference between estimated and actual performance for the measurements reported in this chapter is approximately 14%. The remainder of this chapter is arranged as follows. In section 5.2 the mapping problem is reviewed. In section 5.3 a general memory model is established. In section 5.4 a multiprocessor model is proposed as a basis for discussion. Some of the characteristics of a particular multitransputer network are outlined in section 5.5 as a basis for calculating memory models. Subsequent sections analyse point (section 5.6), neighbourhood (section 5.7), medium grain (section 5.8), large grain (section 5.9), and global (section 5.10) memory reference models.

5.2 The Mapping Problem

The mapping problem is the problem of optimally assigning the modules of a parallel program to the processors of a multiprocessor system. The criterion for optimality is commonly the minimisation of the run time of a program or the maximisation of the utilisation of the available bandwidth (the message-carrying capacity) of the interprocessor network. The general problem is thought to be NP-complete [39], so researchers usually make simplifying assumptions or address a particular aspect of the problem. This section contains a literature survey.

Early work on the mapping problem was carried out by Bokhari [37], Chu et al [68], Efe [95], Lew [177], and Lint and Agerwala [178]. Bokhari’s work, based on the fixed allocation of software modules, includes a heuristic algorithm which proceeds by pairwise interchange to find the optimum mapping. Bokhari showed that the mapping problem is equivalent to the graph isomorphism problem. Chu, Holloway, Lan, and Efe examined
three approaches: a graph-theoretic approach based on a min-cut algorithm, an integer-programming technique within the domain \( \{0,1\} \), and heuristic techniques which offer compromises between the preceding two approaches. Efe used a two-stage approach: firstly a module clustering algorithm obtains the minimum interprocessor communications cost without constraints, then overloaded or underloaded processors are identified and some modules are shifted to achieve a balance. Lew analysed the mapping problem in terms of the problem of finding the shortest path in a directed graph. Lint and Agerwala gave special consideration to communications in considering the mapping problem.

The problem of load balancing, involving the partitioning of a problem in such a way that the workload across multiple processors is balanced, has been considered by Berger and Bokhari [29], by Ma et al. [179], by Ni and Hwang [207], and by Sadayappan and Ercal [246]. Berger and Bokhari use a binary decomposition of the problem domain to partition it into rectangles requiring equal computational effort, and study the communication costs of mapping this partitioning onto different multiprocessors. Ma et al. use a branch and bound technique to find the minimum-cost function. Ni and Hwang use queueing models for probabilistic load balancing, while Sadayappan and Ercal use a heuristic graph-based scheme in which they perform an initial nearest-neighbour mapping, followed by a boundary refinement to improve the load balancing. Bianchini and Shen [34] minimise a cost function (equivalent to maximising the total network bandwidth). They assume a fixed placement of the algorithm modules on the architecture. Bokhari [39] addresses the problem of the optimum mapping of program modules onto the architecture for a specific architecture (a single-host, multiple-satellite system) and for three types of program including single-tree structured parallel programs. Cvetanovic [77] examines the case of shared-memory architectures but her results are applicable to networks. She shows that where the computation and communication overhead can be fully decomposed amongst all the processors the speedup is a nondecreasing function of the level of granularity. However if communication is non-decomposable (that is, cannot be broken into smaller units) and the bandwidth is restricted the speedup has a maximum as a function of granularity, and then decreases as granularity continues to grow.

Fukunaga et al [109] have developed an iterative algorithm that assigns jobs that communicate with each other to adjacent processors. Following the first solution, small perturbations are used to achieve near-optimality. Huang [146] provides a partitioning algorithm that finds the optimum by evaluating all possibilities and inspecting the resulting performances. Hudak [148] treats the multiprocessor as a single autonomous computer onto which a program is mapped, rather than as a group of independent processors that carry out complex communication and require complex synchronization. Kruatrachue and Lewis [164] consider best grain size, based on a technique of grain packing, with a time cost of \( O(N^4) \). The authors claim that the grain size specified by a programmer using Occam is unlikely to be near the optimal grain size, and the resultant parallelization yields no advantage over automated parallelization of a sequential program. Lee and Aggarwal [173] use a set of objective functions for optimality evaluation of mapping.

---

1See Appendix B to this thesis for a review of relevant features of graph theory.
the problem graph onto the system graph. One of their functions, based on the maximum communications overhead, is suitable for real-time applications, including image processing. They used a hypercube to test their results.

All of the work reviewed above is based on the assumption that the program modules have been established as complete units to be mapped onto the architecture. As part of the study of memory utilisation reported in the next section, the effect of the placement of variables with respect to a program module will be considered.

5.3 Memory Utilisation

Optimum memory allocation has been studied by Gupta and Toong [123] with respect to shared bus systems. They consider "local memory" and "global memory", using a split-transaction bus that enhances bus bandwidth. The problem of "hot-spots" in multi-stage networks accessing shared memory has received considerable investigation (see for instance [166]), and a variety of solutions have been proposed (for example, Yew et al [301] propose a technique for distributing hot-spot accesses over a software tree whose nodes can be dispersed among many memory modules). In this section memory utilisation issues, with particular reference to image processing, will be examined.

Consider a distributed multi-transputer system in which each transputer has its own local memory and there is no global memory. For convenience data values will be referred to as 'variables' in the remainder of this section. Then, relative to a given transputer, variables may be local to that transputer, in an adjacent transputer, or further removed in the network. This chapter explores the question of the efficiency of access to variables, using memory access models, and considers the consequences of the various assumptions. The analysis is carried out with particular reference to image processing operations in which images are segmented in some way, and there is a one-to-one correspondence or mapping between segments and transputers. Conventionally the segmenting is done in a matrix fashion to yield equi-sized rectangular segments, although these constraints need not apply.

Local memory access to read a variable provides the fastest means of providing the value of a variable. Access to neighbouring transputers is assumed to occur via the inter-processor links of the network. The requesting transputer sends a request; the target transputer reads the request, accesses its own memory, and sends the appropriate value; finally the requesting transputer reads the value. These overheads are high for a single value but can be shared by a packet of values. Access time obviously increases for more remote transputers.

In general, let memory references be of \( u \) types, with elapsed time per word of \( t_i, i \in [0, u - 1] \). No differentiation is made between reading and writing. In a given application the frequency of reference is \( f_i, i \in [0, u - 1] \). Two conditions could be made concerning
the optimized use of memory resources:

(1) \( t_i \geq t_j \quad \forall i > j, \quad i, j \in [0, u - 1] \).

(2) \( f_i \leq f_j \quad \forall i > j, \quad i, j \in [0, u - 1] \).

This requires that memory be organized in such a way that the fastest memory is used to store the most frequently-accessed data. These conditions, however, are not assumed in the following analysis.

Assume that a single address variable \( x \) can be used to reference all of memory space. In the case of non-local memory accessed via inter-transputer links \( x \) could be thought of as the address in extended memory space. The memory spaces can then be mapped onto a linear scale such that memory type \( i \) occupies \( x = x_{i-1} \) to \( x = x_i \). The top of the scale is at \( x = x_u \). Let \( f(x) \) be the frequency of accessing memory at \( x \). If condition (2) is satisfied, \( f(x) \) is approximately a monotonically decreasing function of \( x \). The probability density of accessing memory at \( x \) is

\[
p(x) = \frac{f(x)}{\int_{0}^{x_u} f(x) \, dx} \quad (5.1)
\]

The actual shape of \( f(x) \) is obviously very problem-dependent, and expressions will be derived in sections 5.6 to 5.10.

### 5.4 Multiprocessor Model

Let \( N \) be the number of transputers in the network, \( T_p \) be the total processing time for a given process, assumed evenly distributed amongst the \( N \) transputers, and \( V \) be the total data volume for this process. Then the time \( T(N) \) for \( N \) transputers to perform the process is

\[
T(N) = \frac{T_p}{N} + \frac{V}{N} \sum_{i=0}^{u-1} p(i) t(i) \quad (5.2)
\]

where \( t(i) = t_j, \quad x_{j-1} < x(i) \leq x_j \), that is, the time per word for the appropriate memory reference. The time for a local memory access is \( t(0) \) and will be referred to as \( t_l \).

Define speedup by \( S(N) = T(1) - T(N) \). Then from (5.2)

\[
S(N) = T_p \left( 1 - \frac{1}{N} \right) + V t_l - \frac{V}{N} \sum_{i=0}^{u-1} p(i) t(i) \quad (5.3)
\]

If \( S'(N) > 0 \) then each additional transputer added to the network will result in an improvement in the performance. Define efficiency by \( E(N) = T(1)/(NT(N)) \). Then

\[
E(N) = \frac{T_p + V t_l}{T_p + V \sum p(i) t(i)} \quad (5.4)
\]
Both speedup and efficiency can equivalently be defined as integrals if \( p(x) \) is approximated by a continuous function.

Finally, define a cost function

\[
\xi = \xi_T T(N) + \xi_N N, \quad \xi_T, \, \xi_N > 0
\]  

(5.5)

\( \xi_T \) is the monetary expenditure that would be considered justifiable in order to achieve a unit decrease in the time \( T(N) \) taken to perform a given task. That is, \( \xi_T \) is a measure of the utility of the performance of the multiprocessor. \( \xi_N \) is the cost of including one additional transputer, including all associated hardware, in a network. If \( T(N) \) is a decreasing function of \( N \) then (5.5) will have a minimum value, found by setting \( \xi' = 0 \). The corresponding value of \( N \) will be the optimum number of transputers in a network.

5.5 A Multiprocessor Network

A PC-based transputer network consisting of twenty T414 transputers hosted by an INMOS B004 board has been described in Chapter 4. The approximate number of processor cycles required to access a word on the transputer network can be estimated by counting the appropriate entries in tables supplied in the INMOS databook ([149]). For a 15MHz transputer one cycle is 66.7 nanoseconds. The numbers of cycles have been used to construct Table 5.1 where passive channel input is defined by

\[
vchan \ ? \ x
\]

and dynamic channel input of single words is defined by

\[
vchan \ ! \ request \\
vchan \ ! \ identifier \\
vchan \ ? \ x
\]

Dynamic channel input involves a transputer sending a request for information and waiting for a reply. INMOS state that all transputer instruction times are averages. The

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Time, cycles/word</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>on-chip RAM</td>
<td>3.2</td>
</tr>
<tr>
<td>2</td>
<td>off-chip local static RAM</td>
<td>5.2</td>
</tr>
<tr>
<td>3</td>
<td>passive channel input</td>
<td>28</td>
</tr>
<tr>
<td>4</td>
<td>dynamic channel input of large blocks</td>
<td>32</td>
</tr>
<tr>
<td>5</td>
<td>dynamic channel input of single words</td>
<td>200</td>
</tr>
</tbody>
</table>

Table 5.1: Memory Access Times for a Transputer Network.
external local static RAM is based on an extra two cycles per access, while the time for
the dynamic channel input of a large block is based on the asymptotic time per word as
the number of words in the block tends to infinity. The channel timings are for accesses to
an adjacent transputer. Overheads involved in deadlock-detection algorithms are ignored.
A simplified memory model can be established as follows. Since memory accesses of types
1 and 2 in Table 5.1 are difficult to distinguish they can be combined into a composite
memory access labelled 'local' with an access time of \( t_l = 4.2 \) cycles (the mean of 3.2 and
5.2). Similarly memory accesses based on channel input can be combined into 'non-local'
or 'global' memory access with an access time of \( t_g = 30 \) cycles. Unless full details of a
proposed system are available the exact mix of non-local memory accesses is unlikely to
be available, and the figure for \( t_g \) is an estimate. Low-level image-processing operations
can usually be structured so as to avoid the necessity of transferring individual words.

The number of cycles required to perform certain image-processing operations have been
calculated, using figures supplied in the INMOS databook, and based on the actual code
employed. The figures have been verified by determining the execution times of the
relevant operations on a single transputer. In particular, subtracting a constant from
each pixel takes approximately 248 cycles per pixel, and performing a linear filter on a
3 \times 3 window takes approximately 1320 cycles per pixel. The types of memory references
involved in these operations are described in subsequent sections. These figures are very
coding-specific and could change significantly. The image is assumed to be distributed
over the network, with adjacent image segments occupying adjacent transputers in the
network. The time taken to transmit messages over the inter-transputer links is assumed
to be independent of the amount of traffic on the links. Any errors incurred through
this assumption are likely to be most apparent when dealing with global operations (see
Section 5.10).

As an estimate of the cost-function coefficients, suppose that an expenditure of \$75000
is considered justified in order to perform a sequence of image-processing tasks in 250
milliseconds. Using 15MHz transputers costing \$2000 per fully-serviced transputer\(^2\), the
mean cost (which may not be the same as the marginal cost) of providing the required
speed is

\[ \frac{\Delta \xi}{\Delta T} = 0.02 \ \text{\$/cycle} \]

and the marginal cost of providing transputers is

\[ \frac{\Delta \xi}{\Delta N} = 2000 \ \text{\$/transputer} \]

Thus \( \xi_T = 0.02 \) and \( \xi_N = 2000 \). These values are, of course, purely illustrative.

Consider a type of interconnection network known as the chordal ring network. It was
shown in Chapter 3 that the diameter \( k \) of a special class of chordal ring is related to the
number of nodes \( N \) by

\[ N = 2k^2 + 2k + 1 \quad (5.6) \]

\(^2\)These figures are based on the 1988 New Zealand costs.
and that the probability of requiring \( i \) links to reach any other node is given by

\[
p(i) = \frac{4i}{N - 1}, \quad i \leq k
\]  

(5.7)

Note that (5.7) considers only non-local memory references. In the case of the transputer it can be seen from Table 5.1 that the time taken for local references is very much less than for non-local references, so this approximation may not introduce any serious error.

Suppose that the time taken to traverse \( i \) links is proportional to the number of links. That is, \( t(i) \propto i, \quad i \geq 1 \). Then set

\[
t(i) = t_g i, \quad i \geq 1,
\]

(5.8)

where \( t_g \) is the time taken to traverse one link.

Thus, using the integral form of (5.3), the speedup for random references is approximately

\[
S(N) = T_p (1 - \frac{1}{N}) + V t_l - \frac{2 V k^2 t_g}{3 N (k + 1)}
\]

The general form of this function, based on representative values, is illustrated in Figure 5.1.

![Figure 5.1: Speedup for Random References on a Chordal Ring.](image)

For \( Z \) pixels per image and \( c \) operations per pixel, the condition \( S(N) > 0 \) becomes

\[
c(N - 1) + t_l N > \frac{2 k^2 t_g}{3 (k + 1)}
\]

(5.9)

The right-hand-side of (5.9) varies as \( k \) while the left-hand-side varies as \( k^2 \). Thus if the inequation is true for small \( k \) (say, \( k = 1 \) and \( k = 2 \)) it will be true for large \( k \). Setting
$k = 1$, the inequation becomes $4c + 5t_l > \frac{t_g}{3}$ which, if $t_l = 4.2$, is true for all likely values of $c$ and $t_g$. A chordal ring network involving random memory references will thus show a positive speedup.

Efficiency is approximately

$$E(N) = \frac{c + t_l}{c + \frac{2k^2 t_l}{3(k+1)}}$$  \hspace{1cm} (5.10)

This is illustrated in Figure 5.2.

![Figure 5.2: Efficiency for Random References on a Chordal Ring.](image)

From (5.10) it can be seen that efficiency has a maximum for small $k$, decreasing as $k$ increases. For example, if $c = 1000$, $t_l = 4.2$, and $t_g = 30$, then for $k = 2$, $N = 13$ and $E = 98\%$, while for $k = 10$, $N = 221$ and $E = 85\%$, and if $k = 20$, $N = 841$ and $E = 73\%$.

### 5.6 Memory Reference for Point Operations

Since point operations are very fine grain operations involving only local references (5.3) reduces to

$$S(N) = [T_p + V t_l] \left(1 - \frac{1}{N}\right)$$

so that there is always a speedup, and (5.4) becomes

$$E(N) = 100\%$$

Thus point operations are ideally suited to parallel computation. Speedup measurements made on the transputer network described in Section 5.5 are shown by the filled circles.
in Figure 5.3. The calculated performance depends on the estimation of the number of transputer cycles involved in the operation, based on values given in the INMOS data-book [149], and these values are shown by the empty circles in Figure 5.3. The systematic divergence of the two sets of figures can be fully explained if the actual number of transputer cycles is approximately 8% greater than that estimated. Estimation of the number of cycles required for memory reference depends on whether the memory is on-chip or external, and in the case of external memory on the number of additional cycles required for each reference [149].

Set $V = Z$ and $T_p = Zc$ where $Z$ is the number of pixels in the image and $c$ is the number of transputer cycles per pixel. Then the cost function becomes

$$
\xi = \frac{\xi_T}{N}[c + t_t] + \xi_N N
$$

and this has its minimum value at

$$
N^2 = \frac{\xi_T}{\xi_N} Z[c + t_t]
$$

For the transputer $t_t \ll c$, and (5.11) becomes

$$
N^2 = \frac{\xi_T}{\xi_N} Zc
$$

Figure 5.3: Theoretical and Experimental Speedup for Point Operations.
To illustrate the use of (5.12), take $\xi_T = 0.02$, $\xi_N = 2000$, $c = 1000$ cycles, and $Z = m^2$, corresponding to an image of size $m \times m$. Noting that $t_l \ll c$,

$$N = 0.1m$$

for minimum cost, so that if the point operation is the only one being performed on an image of, say, $100 \times 100$, then 10 transputers will minimise the cost function.

### 5.7 Memory Reference for Neighbourhood Operations

As an example of a neighbourhood operation, suppose that in performing a function on a three by three window, three pixels in adjacent transputers must be accessed for each pixel on the boundary of the segment. Consider, for simplicity, square segments of size $s$. Then

$$s^2 = \frac{Z}{N}$$

where $Z$ is the number of pixels in the image. In general $s \gg 1$. Since there are a total of $4s - 4$ boundary pixels per segment, $12(s - 1) \approx 12s$ external pixels must be accessed out of a total of approximately $9s^2$ accesses per segment (ignoring the external boundaries of the image). It is only necessary to consider references to adjacent transputers. Designate these as non-local references. Thus the probability that a pixel reference is external to this segment is $p_g = 4/(3s) = \frac{4}{3} \sqrt{\frac{N}{Z}}$. Taking $T_p = Zc$ (for $c$ cycles per operation)

$$T(N) = \frac{Zc}{N} + 9Z\left[\left(1 - \frac{4}{3s}\right)t_l + \frac{4}{3s}t_g\right]$$

$$T(1) = Zc + 9Zt_l$$

This yields

$$S(N) = Z(c + 9t_l) \left(1 - \frac{1}{N}\right) - 12\sqrt{\frac{Z}{N}}(t_g - t_l)$$

(5.13)

As an example, consider dividing an image of size $100^2$ between 100 transputers, so that $Z = 10000$ and $N = 100$. Thus $s = 10$, $p_g = 0.13$, and the probability of a pixel reference being local is $p_l = 0.87$. Take $c = 1320$ cycles. Then (5.13) becomes

$$S(N) = 1.36 \times 10^7 \left(1 - \frac{1}{N}\right) - 3096$$

Thus $S(N) > 0 \ \forall N$. Furthermore the first term is much greater than the second, and this is the case for all neighbourhood operations. Thus (5.13) can be simplified to read

$$S(N) = Z(c + 9t_l) \left(1 - \frac{1}{N}\right)$$

which is positive for all values of $N \geq 2$. The general form of this function is similar to that illustrated in Figure 5.3. From (5.4) the efficiency is

$$E(s) = \frac{c + 9t_l}{c + 9t_l + 12(t_g - t_l)/s}$$

(5.14)
When \( s \) is large (that is, few transputers) the efficiency is close to 100%. As the segment size decreases (\( N \) increases) the efficiency decreases asymptotically to the value for \( s = 1 \):

\[
E(s = 1) = \frac{c + 9t_l}{c + 12t_g - 3t_l}
\]

The general form of \( E(N) \) is illustrated in Figure 5.4. Using \( c = 1320 \), \( t_l = 4.2 \), and \( t_g = 30 \) (the units being transputer cycles), \( E(s=1) = 81.4\% \). For a segment size of 100 pixels, \( E(s=10) = 97.8\% \).

The cost function becomes

\[
\xi = \xi_T \frac{Z}{N} \left[ c + 9 \left( \left[ 1 - \frac{4}{3s} \right] t_l + \frac{4}{3s} t_g \right) \right] + \xi_N N
\]

and this has its minimum value at

\[
N^2 = \frac{\xi_T}{\xi_N} Z \left[ c + 9 + \frac{12}{s} (t_g - t_l) \right]
\]

### 5.8 Memory Reference for Medium Grain Operations

A typical medium-grain operation is object recognition in which some features extracted from the original image must be brought together in a single transputer. Consider objects which fall randomly over a number of segments. For simplicity suppose that these segments lie along a single row or a single column of the array, corresponding to a single
row or column of the transputer array. If each object is to be considered as a unit, a
transputer corresponding to one of the segments will need to communicate with all other
transputers covered by that object. Suppose the segments covered are numbered 0,1,2,...
and that the transputer corresponding to segment 0 is to derive parameters associated
with the image of the object as a whole. Then transputer 0 (corresponding to segment
0) will communicate directly with transputer 1, with transputer 2 via transputer 1, and
so on. Let the frequency distribution of the object lengths be \( \phi(x) \), where the domain is
namely \( x \in [0, \infty] \). To obtain \( f(x) \), the frequency distribution of memory references
对应的 to segments \( x = 0,1,2, \ldots \), references to each segment above segment \( x \)
must be summed. Thus transputer 1 is accessed for all references \( x = 1,2, \ldots \); that is,
\[
f(1) = \sum_{i=1}^{\infty} \phi(i).
\]
Similarly transputer 2 is accessed for all references \( x = 2, \ldots \), giving
\[
f(2) = \sum_{i=2}^{\infty} \phi(i).
\]
Thus in general
\[
f(x) = \sum_{i=x}^{\infty} \phi(i). \tag{5.15}
\]
In some cases this is more conveniently expressed as
\[
f(x) = \int_{x}^{\infty} \phi(s)ds. \tag{5.16}
\]
Various forms of \( \phi(x) \) will now be considered.

Firstly, consider the case in which \( \phi(x) \) is constant in an interval \( u \) (so that object lengths
in the interval occur with constant probability density); that is
\[
\phi(x) = \begin{cases} 
  k, & x \in [0, u] \\
  0 & \text{otherwise}
\end{cases}
\]
Then, from (5.16)
\[
f(x) = k(u - x)
\]
which is a linear memory reference model. Thus from (5.1) the probability density is
\[
p(x) = 2\frac{u - x}{u^2}
\]
Suppose that the time taken to traverse \( i \) links is proportional to the number of links.
That is, \( t(i) \propto i \), \( i \geq 1 \). Then equation (5.8) applies. That is, \( t(i) = t_0 i \), \( i \geq 1 \), where
\( t_0 \) is the time taken to traverse one link. Evaluating (5.3) as an integral,
\[
S(N) = T_p \left( 1 - \frac{1}{N} \right) + V t_0 - \frac{V t_0^2}{3N} \tag{5.17}
\]
The general form of (5.17) is similar to that shown in Figure 5.3.
With \( T_p = Zc \) and \( V = Z \) the condition \( S(N) > 0 \) becomes

\[
N > \frac{c + ut_2/3}{c + t_1}
\]  
(5.18)

Allowing for the fact that \( N \geq u \), this inequation is true for all values of \( N \). This analysis represents the most adverse possible circumstance, namely an object occupying the whole of the image plane. The cost function has its minimum value at

\[
N^2 = \frac{\xi T}{\xi N} Z[c - \frac{u^3 t_2}{3}]
\]

Next, suppose that \( \phi(x) \) is linear; that is

\[
\phi(x) = b - ax, \quad x \leq \frac{b}{a}
\]

\[
= 0, \quad x > \frac{b}{a}
\]

For this model short lengths occur with greater probability than longer lengths. From (5.16)

\[
f(x) = \frac{a}{2} x^2 - bx + \frac{b^2}{2a}, \quad x \in \left[0, \frac{b}{a}\right]
\]

Thus \( f(0) = \frac{b^2}{2a} \) is the maximum for this function in the allowed domain. From (5.1)

\[
p(x) = \frac{3a^3}{b^3} x^2 - \frac{6a^2}{b^2} x + \frac{3a}{b}
\]

Then

\[
S(N) = T_p \left(1 - \frac{1}{N}\right) + V t_l - V \frac{b}{N 4a} t_g
\]

which has a form similar to that illustrated in Figure 5.3. Setting \( T_p = Zc \) and \( V = Z \), \( S(N) > 0 \) gives

\[
c \left(1 - \frac{1}{N}\right) + t_l > \frac{1}{N} \frac{b}{4a} t_g
\]

Since \( b/a \leq N \), the maximum value of the right-hand-side is \( \frac{1}{4} t_g \). Thus

\[
c \left(1 - \frac{1}{N}\right) + t_l > \frac{1}{4} t_g
\]

which is true for all image-processing operations. The cost function becomes

\[
\xi = \xi T \frac{Z}{N} \left[c - \frac{b t_g}{4a}\right] + \xi N N
\]

and this has its minimum value at

\[
N^2 = \frac{\xi T}{\xi N} Z[c - \frac{b t_g}{4a}]
\]

Finally, suppose that \( \phi(x) \) is a Poisson distribution, that is

\[
\phi(x) = \frac{\lambda^x e^{-\lambda}}{x!}
\]

120
The Poisson distribution can be applied to problems such as the number of white blood corpuscles on slides as a function of size [294]. Using (5.15)

\[ f(x) = e^{2\lambda} - e^\lambda \sum_{i=0}^{x-1} \frac{\lambda^i}{i!} \]

(5.19)
giving

\[ f(0) = e^\lambda (e^\lambda - 1) \]
\[ f(1) = e^\lambda (e^\lambda - 1 - \lambda) \]
\[ f(2) = e^\lambda (e^\lambda - 1 - \lambda - \frac{\lambda^2}{2!}) \]

and so on. This could, in principle, be evaluated as for the constant and linear models.

5.9 Memory Reference for Large Grain Operations

An example of a large-grain problem is the determination of the convex hull [125]. An application of the convex hull to a machine vision task is given by Bailey and Hodgson [21]. The analysis presented in this section is based on the following algorithm for the computation of a row-based convex hull. The objective of the algorithm is to replace the intensity profile along each row of the image with its convex envelope, with the envelope being zero where the original image is zero. Each transputer determines the convex hull of its own segment before exchanging information on the hull with its immediate neighbours, with their neighbours, and so on until no further change occurs in the envelope.

Suppose that the number of transputers per row is restricted to, say, four or five, as is the case with this University's multi-transputer system. Then any contention on the inter-transputer links that would result in messages having to wait for other transmissions to finish will be very small and will be ignored in the following analysis.

Referring to Figure 5.5, where the horizontal axis is in terms of image segments, let the concave region of an otherwise convex profile extend from \( \text{segment} = A \) to \( \text{segment} = A + W \). The convex hull of this figure consists of the given shape with the central depression filled in. The determination of the hull at \( A + 1 \) depends on the values at \( A \), \( A + 2 \), \( A + 3 \), \( \cdots \) \( A + W \). The number of links required to reference these segments are 1, 1, 2, \( \cdots \) \( W - 1 \) respectively. A similar analysis can be made for all segments between \( A \) and \( A + W \), leading to the results in Table 5.2.

The totals in the table are the number of link messages involved. The total number of references (the sum of the bottom line of the table) is \( W(W + 1) \), giving probabilities of

\[ \frac{2W}{W(W + 1)}, \frac{2(W - 1)}{W(W + 1)}, \cdots, \frac{2}{W(W + 1)} \]
Using $x$ to represent distance from $A$, the probability density can be expressed as

$$p(x) = \frac{2W + 1 - x}{W(W + 1)}$$  \hspace{1cm} (5.20)

This is a linear model.

Consider now the case of a set $\{i\}$ of concavities of width $W_i$ and individual probability distributions for link accesses of

$$p_i(x) = \frac{2W_i + 1 - x}{W_i(W_i + 1)}.$$

The resultant probability distribution is

$$p(x) = \frac{\sum W_i(W_i + 1)p_i(x)}{\sum W_i(W_i + 1)}.$$
which is also linear.

For a single concavity, using (5.3) and (5.8), and making use of the fact that

$$\sum_{x=1}^{W} x = \frac{W(W + 1)}{2}$$

and

$$\sum_{x=1}^{W} x^2 = \frac{W(W + 1)(2W + 1)}{6}$$

the speedup is

$$S(N) = T_p \left(1 - \frac{1}{N}\right) + V t_t \left(1 - \frac{2}{NW}\right) - \frac{V t_s(W + 2)}{3N} \tag{5.21}$$

Setting $T_p = Zc$, $S(N) > 0$ means that

$$c > \frac{t_s W(W + 2)/3 - t_t(NW - 2)}{W(N - 1)}, \quad N \geq 2 \tag{5.22}$$

The right-hand-side of (5.22) has a maximum value at $N = 2$, so that

$$c > \frac{t_s(W + 2)}{3} - 2t_t \left(1 - \frac{1}{W}\right)$$

Taking $t_t = 4.2$ cycles and $t_s = 30$ cycles,

$$c > 10W + 11.6 + \frac{8.4}{W}$$

Thus for $W = 1$, $c > 30$, and for $W = 10$, $c > 112$, both of which are well below the estimated number of cycles per pixel for performing a convex hull. The convex hull will thus show a positive speedup in all likely circumstances. Furthermore, it is easy to verify that $S'(N) > 0 \ \forall N$, so that performance is improved on adding extra transputers to the network.

Using the above techniques to evaluate the efficiency as defined in (5.4),

$$E(W) = \frac{W(c + 4.2)}{Wc + 8.4 + 10W(W + 2)}$$

Taking $c = 1000$, efficiency decreases from 96.2% for $W = 2$ to 89.6% for $W = 10$ to 49.7% for $W = 100$. For a square array,

$$W \leq \sqrt{N}$$

So $N = 64$ means that $W \leq 8$, giving $E \geq 91.2\%$. Even $N = 1000$ means that $E \geq 75.5\%$.

The cost function has its minimum value at

$$N^2 = \frac{\xi_T}{\xi_N} Z(c - \frac{2t_t}{W} - \frac{t_s(W + 2)}{3}) \tag{5.23}$$

The second and third terms in the bracket are generally small compared to the first, and the expression for the minimum becomes the same as (5.12), which minimises the cost for point operations.
5.10 Memory Reference for Global Operations

In this section global memory references will be treated as an extension of memory references for medium scale operations. However these results need to be modified to take account of message contention on the inter-transputer links, and a simple model will be developed for this purpose.

Referring to section 5.8, global memory references with a constant frequency distribution correspond to the case of \( f(x) \) a constant. The interval for \( f(x) \) depends on the network configuration, and two interconnection patterns will be considered. The simplest network is a linear chain of transputers, and in this case the interval is approximated by \( N/2 \) where \( N \) is the number of transputers in the chain. Then from (5.1)

\[
p(x) = \frac{2}{N}
\]

and setting \( t(x) = t_g x \), (5.3) yields

\[
S(N) \approx T_p \left( 1 - \frac{1}{N} \right) + V t_l \left( 1 - \frac{2}{N^2} \right) - \frac{V t_g}{4}
\] (5.24)

In the case of a mesh network, the distance, which is the maximum number of links to be traversed, is approximated by \( \sqrt{N} \), and the speedup becomes

\[
S(N) \approx T_p \left( 1 - \frac{1}{N} \right) + V t_l \left( 1 - \frac{1}{N^{3/2}} \right) - \frac{V t_g}{2\sqrt{N}}
\] (5.25)

These equations will now be modified to take account of message contention. Let \( \rho \) be the probability that a message is being transmitted over links between any specified pair of transputers, and assume that \( \rho \) is the same for all pairs of transputers. Consider the case in which transputer 2 passes a message to transputer 1, transputer 3 passes a message to transputer 1 via transputer 2, and so on. The probability of any pair of messages colliding is \( \rho^2 \). Given a total of \( M \) transputers involved in the message passing, there are \( M - 1 \) transputers acting as message sources, so the number of message pairs is

\[
M-1 C_2 = \frac{(M-1)(M-2)}{2}
\]

Assume that \( \rho \ll 1 \), so that the probability \( \rho^3 \) of three messages colliding is negligible. Then the probability of a collision on the link leading into transputer 1 from transputer 2 is

\[
\rho_c = \frac{(M-1)(M-2)\rho^2}{2}
\]

The effect of these message collisions will be to increase the execution time \( T(N) \) by an amount

\[
\frac{1}{1 - \rho_c} \approx 1 + \rho_c = 1 + \frac{(M-1)(M-2)\rho^2}{2} \approx 1 + \frac{M^2\rho^2}{2}
\] (5.26)
The value of the probability $\rho$ can be estimated from the ratio of the time taken for packet transmission to the time taken for processing and local memory references. The packet transmission time is

$$\frac{V}{N} \sum_{i=1}^{u-1} p(i)t(i)$$

and the time for processing and local memory references is

$$\frac{T_p + Vt_i}{N}$$

Thus

$$\rho = \frac{Vt_gN}{2(T_p + Vt_i)}$$

The value of $M$ depends on the network configuration. For a linear network, $M = N$, while for a square mesh, $M = \sqrt{N}$.

Equation (5.24) for speedup for global operations on a linear chain of transputers becomes

$$S(N) \approx T_p + Vt_i - \left(\frac{T_p}{N} + \frac{Vt_g}{4}\right) \left(1 + \frac{V^2t_g^2N^4}{8(T_p + Vt_i)^2}\right)$$  \hspace{1cm} (5.27)$$

and (5.25) becomes

$$S(N) \approx T_p + Vt_i - \left(\frac{T_p}{N} + \frac{Vt_g}{2\sqrt{N}}\right) \left(1 + \frac{V^2t_g^2N^3}{8(T_p + Vt_i)^2}\right)$$  \hspace{1cm} (5.28)$$

where it has been assumed that $t_i/N$ is negligible compared to $t_g$. The speedup for a mesh of twenty transputers as calculated from equation (5.28) is shown by open circles in Figure 5.6, together with experimentally measured points, shown by filled circles. As can be seen the theoretically determined values, which were based of figures given in the INMOS databook [149], are consistently high. However, if the estimated number of transputer cycles per operation is decreased by approximately 14% the resultant points, shown as crosses in Figure 5.6, agree closely with the measured values. This suggests that the major difficulty in applying equations (5.27) and (5.28) without recourse to at least one transputer is the determination of the number of machine cycles involved.

The efficiency for the same set of figures is shown in Figure 5.7. The figures that have been determined theoretically and experimentally agree to within approximately 3%. From (5.28), the condition for incremental speedup, namely $S'(N) > 0$, becomes

$$8(T_p + Vt_i)^2(4T_p + 5Vt_g\sqrt{N}) > V^2t_g^2N^3(8T_p + 5Vt_g\sqrt{N})$$  \hspace{1cm} (5.29)$$

For large $N$, the left hand side of (5.29) varies as $\sqrt{N}$ while the right hand side varies as $N^{3.5}$. Thus as $N$ increases incremental speedup will eventually be lost, and the point at which additional transputers will result in a decrease in the speedup can be estimated from (5.29).
5.11 Discussion

The loss of efficiency noted in regard to neighbourhood operations can be avoided by using overlapping image segments. Pixels along the boundaries of a segment will be duplicated.
in the adjacent segment. In that case the $t_g$ in (5.14) is replaced by $t_l$, resulting in $E(s) = 1$. A price to be paid for such a scheme is that point and other operations must be applied to the duplicated pixels. For a segment size of $s$ by $s$ pixels, the additional $4s + 4$ boundary pixels result in the efficiency being decreased by a factor of $s^2 / (s^2 + 4s + 4)$ for point and other operations.

The analysis of medium and large grain operations was based on the characteristics of an array of transputers. Here, links on the array boundary not employed within the array can be used to provide shorter paths to other boundary transputers. Such networks will improve the efficiency of larger grain operations. The characteristics of one such network, the degree four chordal ring network, have been described in Chapter 3.

### 5.12 Conclusions

The techniques established in this chapter can be applied to the analysis of any parallel processing application in which the frequency distribution of memory references can be determined. Image-processing operations can be classified as point, neighbourhood, medium grain, large grain, and global. In this chapter methods have been presented for calculating the speedup and efficiency of such operations executed on a network, where the equivalent performance on a single processor is taken as the benchmark. When these operations are analysed with respect to a transputer network, the operations display a positive speedup in all likely cases, and the extent of this speedup can be estimated before the application is run on a multiprocessor network. These techniques can thus be used to estimate hardware requirements for a proposed multi-transputer system. The maximum difference between estimated and actual performance for the measurements reported in this chapter is approximately 14%. The major source of error in applying these techniques is in the estimation of the number of machine cycles involved. Access to a disassembler would help to overcome this problem. Alternatively a software utility which provides such an estimate on a block of Occam code would be very useful.
Chapter 6

Lighting and the Detection of Projections

6.1 Introduction

Machine vision, especially as it is applied to kiwifruit, is dependent for its effectiveness on the lighting used, and in this chapter the lighting of kiwifruit is examined by developing a mathematical model of the lighting and of the kiwifruit. This is then applied to a particular problem in regard to kiwifruit — the detection of the small surface projections called Hayward hooks.

The importance of the provision of suitable lighting in machine vision has been emphasized by Batchelor et al [23] who review a number of lighting systems. Specular reflection is generally undesirable in machine vision systems, and the detection of surface features can be improved by maximising diffuse reflection and minimising specular reflection, according to Paulsen and McClure [219]. Their solution is to use diffuse lighting. The subject of specular reflection has been examined in regard to computer-generated pictures by Phong [221]. Schroeder [254] has reviewed the basic concepts of illumination for machine vision.

The Hayward hook is a small projection (up to a few millimetres in extent) that is found on the surface of a small percentage of kiwifruit, and is not acceptable on export-grade fruit. Flat lighting makes projections such as Hayward hooks almost invisible unless they are seen in profile, whereas strongly directional lighting (such as is produced by spotlights) may result in shadowing which can be used as a basis for detection. The latter form of lighting is used in at least one commercially-available optical size grader, and is commonly encountered in prototype vision systems [23]. However, a kiwifruit illuminated by two lights symmetrically disposed about the camera position may show a dip between the two highlights in the intensity profile, and unless this is compensated for in the software it may lead to the 'detection' of a non-existent blemish. The problem
considered here is: what arrangement of lights will maximise the detectability of Hayward hooks on kiwifruit. Figures 6.1 and 6.2 illustrate the choice between oblique lighting,

![Oblique Lighting Diagram](image1)

**Figure 6.1:** Illustration of Oblique Lighting.

![Flat Lighting Diagram](image2)

**Figure 6.2:** Illustration of Flat Lighting.

which emphasises projections, and flat lighting, which avoids the problem of a dip in the
intensity profile.

The approach used here is based on a mathematical model of a kiwifruit. This is because there are two major difficulties with attempting to determine the optimum geometry experimentally:

1. Since the objective is to detect Hayward hooks, fruit free from other surface defects would be required in order to be certain that such features did not mask the effect of the Hayward hook. Such fruit, however, would be difficult to find.

2. A very large data volume would be required. As a rough estimate, if lighting is investigated at 2° intervals in the range [10°, 70°], and for each lighting arrangement the fruit are oriented at 2° intervals in the range [0°, 90°], 1350 images will be involved, which amounts to 88 MByte of data for 2562 images. Furthermore, if the type of lights or some other circumstance were to change the experiment would need to be repeated.

The alternative explored here is to develop a model which can be used to explore various lighting arrangements.

The characteristics of uniform light sources have been well established for many years [290] and the characteristics of light-scatterers have been studied [143,290]. In section 6.2, the basic geometry of illuminated cylinders with and without projections will be derived. The lighting model is introduced in section 6.3, and the results of fitting the model to general cylinders and to kiwifruit are presented in sections 6.4 and 6.5 respectively. The characteristics of the central dip are investigated in section 6.6. A model of shadowing is presented in section 6.7. This model is combined with the lighting model in section 6.8, which considers the question of detectability of a projection. A comparison of the model with actual projections is given in section 6.9, together with an estimation of the optimum lighting arrangements.

6.2 Geometry

Figure 6.3 illustrates the case of a cylinder illuminated at an angle \( \beta \) from the viewing direction. Let \( \delta \) be the difference between the detection and specular directions. Then

\[
\delta = -2 \sin^{-1} \left( \frac{x}{r} \right) + \beta
\]  
(6.1)

Similarly for the case in which the viewing direction is on the opposite side of the direction of specular reflection

\[
\delta = 2 \sin^{-1} \left( \frac{x}{r} \right) + \beta
\]  
(6.2)

The case of a simple projection of height \( h \) on the side of a cylinder illuminated from the opposite side of the cylinder is illustrated in Figure 6.4. Referring to Figure 6.4,
the angle $\sigma$ is given by

$$\sin \sigma = (1 + \frac{h}{r}) \sin (\beta + \phi)$$  \hspace{1cm} (6.3)

where a solution exists only if the right-hand-side does not exceed unity. As can be seen from the diagram, $\sigma \geq \frac{\pi}{2}$. If a value exists for $\sigma$ then the projected size of the shadow is given by

$$s = r \sin (\beta + \sigma) - (r + h) \sin \phi$$  \hspace{1cm} (6.4)

The case in which the lighting is on the same side of the cylinder as the projection yields the following results:

$$\sin \sigma = (1 + \frac{h}{r}) \sin (\beta - \phi), \quad \sigma \geq \frac{\pi}{2}$$  \hspace{1cm} (6.5)

$$s = r \sin (\beta + \phi) + r \sin \phi$$  \hspace{1cm} (6.6)

In the case of Hayward hooks on kiwifruit, $h \ll r$. 

131
6.3 Lighting Model

If Lambert's cosine law of emission [290] is applied to the cylinder in Figure 6.3 then the intensity as a function of $\phi$ is given by

$$I(\phi) = I_0 \sin(\phi + \beta) = I_0 \cos\left(\frac{\beta + \delta}{2}\right)$$

where $I_0$ is the intensity of the diffuse emission or scattering in the specular direction.

Horn [143, 144, 250] has suggested an expression of the form

$$I = Ak \cos(\beta - \phi) + \left(\frac{A}{2}\right)(1 - k)[2 \cos \phi \cos(\beta - \phi) - \cos \beta]^n$$

where $A$ is a constant related to illumination intensity and surface reflectance, $k$ represents the relative significance of the diffuse (Lambertian) and specular components of reflection, and $n$ determines the sharpness of the specular component. Rindfleisch [240] describes a formula proposed by Seelinger, based on an empirically determined table, and a formula by Fesenkov. Both formulae were used to describe the reflectivity of the lunar surface. Fesenkov's formula is supported by a theoretical model of the surface due to Hapke [144]. From the point of view of the work to be described in this investigation the formulae of Horn, Seelinger, and Fesenkov are mathematically unattractive.
Following work on projection screens, Schwesinger [255] has proposed

\[ B = \frac{B_0}{(1 + a^2 \theta^2)} \]

for the specular intensity \( B \), where \( \theta \) is the viewing angle. This will be used in the current work and will be shown to give a good agreement with measurements of actual reflecting surfaces.

Combining Schwesinger's formula with the expression for diffuse scattering,

\[ I = I_a \cos\left(\frac{\beta + \delta}{2}\right) + \frac{I_b}{1 + \left(\frac{\delta}{\delta_0}\right)^2} + I_c \quad (6.7) \]

where \( I_a, I_b, I_c, \) and \( \delta_0 \) are specific to a particular light source and a particular cylinder.

The case of a cylinder illuminated by two light sources symmetrically disposed about the vertical follows from (6.7) by adding the expression for positive \( \beta \) with an equivalent expression for negative \( \beta \). The result is most conveniently expressed as a function of the distance \( x \) (Figure 6.3):

\[ I = 2I_a \cos \beta \frac{\sqrt{r^2 - x^2}}{r} + \frac{I_b}{1 + \left(\frac{\delta_+}{\delta_0}\right)^2} + \frac{I_b}{1 + \left(\frac{\delta_-}{\delta_0}\right)^2} + 2I_c \quad (6.8) \]

where

\[ \delta_- = \beta - 2 \sin^{-1}\left(\frac{x}{r}\right) \]

and

\[ \delta_+ = \beta + 2 \sin^{-1}\left(\frac{x}{r}\right) \]

6.4 Fitting To General Cylinders

The lighting model of (6.8) was applied to the following cylinders:

1. a shiny polythene pipe
2. a buffed polythene pipe
3. a grindstone wheel
4. a cylinder formed from black masking tape, and
5. a cardboard roll.

The curve was fitted to minimise the root-mean-square deviation. The values of the coefficients \( I_a, I_b, \) and \( I_c \) are functions of the intensity of the incident light, but \( \delta_0 \) describes the specular nature of the cylinder. The results of the model-fitting are summarised in

133
Table 6.1: Model-Fitting to General Cylinders.

The R.M.S. error is the root-mean-square deviation of the actual intensity in gray-scale units compared to the intensity calculated from the model. Since the intensity takes integral values in the range 0 to 255 the results of the curve fitting are generally within the possible random variations resulting from the digitization of the intensity. The ratio \( \frac{I_s}{I_a} \) determines the relative significance of the scattering and specular components (the ratio being large in cases in which the specular component is dominant). The value of \( \delta_0 \) determines the ‘sharpness’ of the specular reflection.

### 6.5 Fitting To Kiwifruit

The results of fitting the model to four kiwifruit are summarised in Table 6.2.

Table 6.2: Model-Fitting to Kiwifruit.

Samples 1 and 2 are visually related to each other, as are samples 3 and 4. The validity of the lighting model is exemplified by the fact that the root-mean-square errors are less than one (ideally the original intensity data has a possible error of \( \pm 0.5 \)).

### 6.6 Approximations, and Compensation for Dip

A cylinder having a significant specular component and illuminated by two lights symmetrically placed about the viewing direction will exhibit highlights with an intervening dip in the intensity profile (Figure 6.5). As this dip may be wrongly ascribed to a surface
blemish in the case of kiwifruit it is important to find the size of the dip and to have a means of eliminating it from the image.

It is necessary to obtain an expression to describe the central part of the intensity profile. Following some experimentation it was found that this could be approximated by

\[ I(x) = -\gamma x^4 + \mu x^2 + \sigma, \quad x \in \text{dip}, \quad (6.9) \]

where \( x = 0 \) at the centre of the dip (that is, at \( \phi = 0 \)). The first term in (6.9) ensures that this curve can be fitted to the overall intensity profile at the edges of the dip, while the second term provides the actual dip. This is to be fitted to (6.8) in the vicinity of the dip. At \( x = 0 \), let \( I = I_0 \). Then

\[ I(0) = \sigma = I_0 \]

The highlights occur at \( x = \pm x_m \) where \( x_m = r \sin(\phi/2) \) is the location where \( I = I_m \).

Assume that \( I'(x) = 0 \) at \( x = \pm x_m \). From (6.9), this leads to

\[ \gamma = \frac{I_m - I_0}{x_m^4} \quad (6.10) \]

and

\[ \mu = \frac{2(I_m - I_0)}{x_m^2} \]

The objective is to transform the image in such a way that this central intensity dip is eliminated. Form a new function

\[ J(x) = I(x) + \Gamma x, \quad x \geq 0 \quad (6.11) \]

135
such that $J(x)$ is a monotonically decreasing function of $x$ for $x \geq 0$ (that is, the central dip is eliminated). This requirement is satisfied if

$$J'(x) \leq 0 \quad \forall x \geq 0$$

Thus

$$I'(x) + \Gamma \leq 0$$

Differentiating (6.9) and substituting for $\gamma$ and $\mu$,

$$\Gamma \leq -I'(x) = 4\gamma x^3 - 2\mu x$$

$$= \frac{4(I_m - I_0)x(x^2 - x_m^2)}{x_m^4}, \quad 0 \leq x \leq x_m$$

Set $\Gamma = \min(-I'(x))$. The corresponding value of $x$ can be conveniently found by setting $I''(x) = 0$:

$$I''(x) = -\frac{4(I_m - I_0)(3x^2 - x_m^2)}{x_m^4}, \quad 0 \leq x \leq x_m$$

$$= 0$$

giving $x = x_m/\sqrt{3}$. Substituting into the expression for $I'(x)$, the compensation factor is given by

$$\Gamma = \frac{-8(I_m - I_0)}{3x_m\sqrt{3}}$$

Although this value will vary from one intensity profile to another a bounding value could be found for any given application (such as kiwifruit) and the one compensation factor applied to all kiwifruit. As an example, if the intensity dip $I_m - I_0 = 10$ on an intensity scale of 0 to 255, and the half-width $x_m$ of the dip is 10 pixels, then $\Gamma = -1.54$ and (6.11) becomes $J(x) = I(x) - 1.54x$, $x \geq 0$. A similar expression holds for $x \leq 0$, and the overall correction can be expressed as

$$J(x) = I(x) - 1.54|x|$$

Note that at $x = \pm 100$ the intensity correction is -154. Thus if the original pixel intensities are byte values the use of the compensation factor may result in values that must be handled as integers. This has implications for image storage but is not likely to affect processing if a computer with a word length of 16 or more bits is used.

6.7 Geometry Of Shadows From Projections On Kiwifruit

The primary question is: what arrangement of lights will give the maximum probability of detecting a projection on the surface of a cylinder? Of interest are those cases such as Hayward hooks in which $h \ll r$, where $h$ is the height of the projection above the surface of a cylinder of radius $r$. 

136
The remainder of this section will consider the case of projections on the surface of kiwifruit. The resultant images are assumed to be analysed by the following surface-blemish detection algorithm:

1. The image is smoothed (ideally a median filter (see, for instance, [64] or [225]), but approximated here by a linear filter based on the mean in a 3x3 window).

2. The intensity envelope is taken as an approximation to the convex hull [31,125], and the difference is obtained between the envelope and the actual intensity profile.

3. The image is thresholded and the area obtained.

The analysis which follows could equally well apply to other algorithms but non-linear features such as rank filters are difficult to handle analytically [26]. Linear filters lead to loss of edge information but have only marginal effects on the detection of surface blemishes in kiwifruit. Thus the adoption of this particular algorithm is a mathematical convenience. However, experiments have shown that this algorithm produces results comparable to those obtained by other techniques and so its use in this analysis is considered justified. A further mathematical convenience is the use of continuous functions in place of the actual (discontinuous) functions. This allows, for instance, summation to be replaced by integration.

Let the unit of $x$ be the inter-pixel distance. Consider a smoothing filter in the window $[x \pm \alpha]$ where $\alpha = 1$ for a 3x3 window. In the one-dimensional case the smoothed version of some intensity function $I(x)$ is

$$f_s(x) = \frac{1}{2\alpha + 1} \int_{x-\alpha}^{x+\alpha} f(x) \, dx$$

(6.14)

For mathematical convenience the expression derived in (6.14) will be multiplied by $(2\alpha + 1)/(2\alpha)$.

Consider a general ‘defect’ in the intensity profile. Define the $x$-axis so that $x = 0$ at the centre (lowest point) of the defect (see Figure 6.6). In the vicinity of the defect $I(x)$ will be approximated by the following function:

$$I(x) = I_6x^6 + I_4x^4 + I_3x^3 + I_1x + I_0$$

(6.15)

The term in $x^6$ would be expected to have a negative coefficient and gives a ‘bathtub’ shape to the shadow profile. The remaining terms provide the correct positioning of the shadow relative to the unshadowed profile. The actual form of (6.15) was chosen after some experimentation with a range of expressions, and has been found to provide the best fit in an R.M.S. sense to actual shadow profiles that have been measured. Equation (6.15) includes the fact that actual shadow edges will not have discontinuous slopes but will be rounded since normal light sources are not point sources. Techniques for obtaining the values of $I_6$, $I_4$, $I_3$, $I_1$, and $I_0$ are discussed in the next section.
In applying (6.14) to (6.15), functions of the form

\[ Q(x, a, b) = \frac{1}{2a} \int_{x-a}^{x+a} ax^b \, dx \]

are involved. This yields

\[
\begin{align*}
Q(x, a, b) &= a[(x + \alpha)^{b+1} - (x - \alpha)^{b+1}]/(2a(b + 1)) \\
&= a((x^{b+1} + C_1^{b+1} x^b \alpha + \cdots + C_b^{b+1} x^b \alpha^b + \alpha^{b+1})
\quad - (x^{b+1} - C_1^{b+1} x^b \alpha + \cdots))/((2a(b + 1)) \\
&= a(C_1^{b+1} x^b \alpha + C_3^{b+1} x^{b-2} \alpha^3 + \cdots)/(a(b + 1))
\end{align*}
\]

Thus \( Q(x, 1, 2) = x^2 + \alpha^2/3, \ Q(x, 1, 3) = x^3 + \alpha^2x, \) and so on. Substituting and collecting terms, the smoothed intensity is

\[
I_s(x) = I_6x^6 + (5\alpha^2 I_6 + I_4)x^4 + I_3x^3 + wx^2 + ux + v \tag{6.16}
\]

where \( u = I_1 + \alpha^2 I_3, \ v = I_6\alpha^6 + I_0 + I_4\alpha^4/5 + I_6\alpha^2/3, \) and \( w = 3\alpha^4 I_6 + 2\alpha^2 I_4. \)

The intensity envelope can be approximated by the value \( I_e = (I_p + I_q)/2 \) across the width of the shadow which, from the assumption of \( h \ll r, \) will be small compared to the radius. The difference \( I_e - I_s \) is compared with some threshold value, and regions within which the difference exceeds that threshold are the shadow regions from the point of view of projection detection. This is illustrated in Figure 6.7.
The boundaries of the shadow region are obtained from the solution of the equation

\[ I_e - I_s(x) = \text{threshold} \]  

(6.17)

where threshold is some operational value below which the difference is considered to be too close to the noise level to be reliable. Equation (6.17) yields

\[-I_6x^6 - (I_4 + 5\alpha^2I_6)x^4 - I_3x^3 - \alpha^2(2I_4 + 3\alpha^2I_6)x^2 - (I_5\alpha^2 + I_1)x\]

\[+((I_p + I_q)/2 - \text{threshold} - \alpha^4(I_4/5 + \alpha^2I_6/7 + I_6/\alpha^4 + I_6/(3\alpha^2))) = 0 \]  

(6.18)

Equation (6.18) provides an expression for deriving the two values of \(x\) on either side of the dip. The enclosed area can be calculated and this provides a measure of the extent of the dip.

### 6.8 Combination of Shadows and the Lighting Model

Referring to Figure 6.8, the form of the envelope has been derived in the lighting model. Let \(I_s\) be the minimum shadow intensity and \(I_e\) be the envelope intensity at the corresponding value of \(x\). Relate these with

\[ I_s = mI_e \]  

(6.19)
where \( m \) will be a measure of the amount of scattering.

The objective is to fit the curve given in (6.15) to the well illustrated in Figure 6.8, ensuring that both \( I(x) \) and \( I'(x) \) are continuous at both P and Q. To make this problem reasonably tractable two approximations are introduced. Firstly, set

\[
I_s = I(x_m)
\]

where

\[
x_m = \frac{x_p + x_q}{2}
\]

Secondly, set

\[
I_e = \frac{I_p + I_q}{2}
\]

so that

\[
I_s = \frac{m(I_p + I_q)}{2} = mI_e
\]

Let \( x_q - x_m = x_m - x_p = \Omega \). Then

\[
\begin{pmatrix}
I_p \\
I_q \\
I'_p \\
I'_q
\end{pmatrix} =
\begin{pmatrix}
\Omega^6 & \Omega^4 & \Omega^3 & \Omega & 1 \\
\Omega^6 & \Omega^4 & -\Omega^3 & -\Omega & 1 \\
6\Omega^5 & 4\Omega^3 & 3\Omega^2 & 1 & 0 \\
-6\Omega^5 & -4\Omega^3 & 3\Omega^2 & 1 & 0
\end{pmatrix}
\begin{pmatrix}
I_0 \\
I_1 \\
I_2 \\
I_3 \\
I_4
\end{pmatrix}
\]

Figure 6.8: Combination of Shadows and the Lighting Model.
This is readily solved, yielding

\[
\begin{align*}
I_6 &= \frac{(I'_q - I'_p) - 8\Omega^3 I_4}{12\Omega^5} \\
I_4 &= \frac{6(I_q + I_p)(1 - m) - \Omega(I'_q - I'_p)}{4\Omega^4} \\
I_3 &= \frac{\Omega(I'_q + I'_p) - (I_q - I_p)}{4\Omega^3} \\
I_1 &= \frac{(I'_q + I'_p) - 6\Omega^2 I_4}{2}
\end{align*}
\]

Intensity \( I \) is given by (6.8), and \( I' \) is found by differentiating (6.8):

\[
I'(x) = \frac{-2I_0 x \cos \beta}{r\sqrt{r^2 - x^2}} + \frac{4I_b}{\delta_0^2 \sqrt{r^2 - x^2}} \left( \frac{\delta_+}{1 + \left(\frac{x}{\delta_0}\right)^2} - \frac{\delta_-}{1 + \left(\frac{x}{\delta_0}\right)^2} \right)
\]

(6.21)

The shape of the dip described by the above equations must be reduced to some single measure of detectability. Three features are involved: the distance of the feature from the edge of the cylinder, the maximum contrast, and the shadow area. Suitable criteria which have something of a practical basis are as follows. Designate detectability by the parameter \( \Phi \). The value of \( \Phi \) as a factor of distance from the edge could be expressed as

\[
\Phi_e = 1 - \left(\frac{x}{x_0}\right)^2, |x - x_0| > g
\]

\[
= 0 \quad \text{otherwise.}
\]

This indicates that a shadow within a distance \( g \) (the 'clipping distance') of the edge will be disregarded, and that detection is given increased weight towards the centre.

Defining contrast as

\[
c = \frac{I_e - I_m}{I_e} = 1 - m
\]

the contribution of contrast to detectability could be expressed as

\[\Phi_c = 1 - m\]

Finally the detectability of an area \( A \) could be expressed as

\[\Phi_a = A, \quad A \geq A_0\]

\[= 0, \quad A < A_0\]

That is, above some threshold \( A_0 \) the detectability is proportional to the shadow area.

These expressions can then be combined into a single measure:

\[
\Phi = \Phi_e \Phi_c \Phi_a
\]

(6.22)

This ensures that if one of the component detectabilities, for example area, is zero then the overall detectability will be zero. Furthermore detectability is proportional to each
of the component detectabilities. It corresponds closely to the way in which the kiwifruit blemish detection is applied.

The full procedure is as follows:

1. From the geometry of the lighting and the projection find \( x_p \) and \( x_q \). In general two shadows must be investigated. Hence determine \( x_m \).

2. Find \( I_p, I_q, I'_p, \) and \( I'_q \) from (6.8) and (6.21).

3. Use (6.20) to obtain the coefficients of the polynomial.

4. From the geometry find the distance from the edge, use (6.8) and (6.19) to determine the maximum contrast between shadow and background, and use (6.18) to determine the enclosed thresholded area. These three factors are combined in (6.22) to obtain a single value representing the detectability \( \Phi \) of the projection.

6.9 Results

Experimental verification of the lighting model and of the derivation of optimum lighting conditions is a very time-consuming task and was only undertaken for a limited number of lighting geometries. In section 6.9.1 a comparison is made between the model and an actual Hayward hook shadow profile. Section 6.9.2 examines the results of applying the methodology outlined above to determine the detectability of projections. Finally in section 6.9.3 the results of an experiment to measure detectability is reported.

6.9.1 Shadow Profile

A visual comparison between the model and an actual Hayward hook is given in Figure 6.9 where the upper curve is a full profile for \( \phi = 10^\circ \) and \( \beta = 45^\circ \), the shadows being the dips on either side of the dashed line, and the lower curve is the intensity profile through an approximately-comparable Hayward hook illuminated by a single light. When allowance is made for the irregularities in the actual image the model can be seen as an acceptable approximation. It is easy to find cases where Hayward hooks cast shadows that differ appreciably from the model but there is such a diversity of actual shadows that very few generalisations can be made. The model of projections assumed a projection of zero thickness, which is a reasonable approximation in some cases but is clearly invalid in others.
6.9.2 Detectability — Theory

The procedure outlined in section 6.8 for determining the detectability of a projection can be applied to a model based on the known kiwifruit parameters. Given a projection of known size (a specific Hayward hook) various lighting configurations can be explored to determine the best arrangement for the detectability of Hayward hooks, and to determine how critical light placement is for projection detection.
The values of the optimum lighting angle $\beta_0$ for a number of examples are given in Table 6.3. Given that $\beta_0$ is the important value to be deduced, since this will minimize

<table>
<thead>
<tr>
<th>$I_a$</th>
<th>$I_b$</th>
<th>$\delta_0$</th>
<th>h</th>
<th>Clipping distance</th>
<th>Scatter $m$</th>
<th>Min. contrast</th>
<th>Min. area</th>
<th>Intensity threshold</th>
<th>$\beta_0$ (°)</th>
<th>Mean detectability</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>40</td>
<td>40</td>
<td>1</td>
<td>0</td>
<td>0.5</td>
<td>0.1</td>
<td>2</td>
<td>0</td>
<td>62</td>
<td>1.18</td>
</tr>
<tr>
<td>40</td>
<td>15</td>
<td>25</td>
<td>1</td>
<td>0</td>
<td>0.5</td>
<td>0.1</td>
<td>2</td>
<td>0</td>
<td>56</td>
<td>1.19</td>
</tr>
<tr>
<td>20</td>
<td>30</td>
<td>65</td>
<td>1</td>
<td>0</td>
<td>0.5</td>
<td>0.1</td>
<td>2</td>
<td>0</td>
<td>60</td>
<td>1.25</td>
</tr>
<tr>
<td>40</td>
<td>40</td>
<td>40</td>
<td>2</td>
<td>10</td>
<td>0.75</td>
<td>0.25</td>
<td>4</td>
<td>10</td>
<td>34</td>
<td>1.86</td>
</tr>
<tr>
<td>40</td>
<td>40</td>
<td>40</td>
<td>4</td>
<td>0</td>
<td>0.5</td>
<td>0.1</td>
<td>2</td>
<td>0</td>
<td>42</td>
<td>2.43</td>
</tr>
<tr>
<td>40</td>
<td>40</td>
<td>40</td>
<td>1</td>
<td>5</td>
<td>0.5</td>
<td>0.1</td>
<td>2</td>
<td>0</td>
<td>54</td>
<td>1.17</td>
</tr>
</tbody>
</table>

Table 6.3: $\beta_0$ and Mean Detectability For a Range of Examples.

the angle of the lighting subject to maximizing the detectability, it can be seen that the parameters that describe the light-scattering nature of the kiwifruit, namely $I_a$, $I_b$, and $\delta_0$, have little effect on the optimum value of $\beta_0$. If larger Hayward hooks (represented by the height $h$) are to be detected the optimum value of $\beta_0$ decreases. Similarly, if the conditions on detectability, namely the minimum contrast, the minimum area, and the intensity threshold are tightened, the optimum value of $\beta_0$ decreases. Since large values of $\beta$ in a kiwifruit optical detection system may lead to problems with a central dip in the intensity profile (section 6.6), it would seem prudent to adopt as small a value of $\beta$ as possible, subject to optimising the detectability of Hayward hooks. The next section compares these model-based projections with the results of an experiment.

### 6.9.3 Detectability — Experimental

Referring to Figure 6.10, a buffed polythene tube of radius 33mm had a thin cardboard strip of width 4mm attached in a spiral fashion. There is thus a direct relation between the longitudinal position on the tube and the value of $\phi$ (refer to Figure 6.3). The tube was viewed from the vertical position and was illuminated by two lights symmetrically placed on either side of the vertical. Images of size $256^2$ were obtained. An example of the intensity profile through one of these images is shown in Figure 6.11. These images were analysed using the Department's VIPS$^1$ system via the following steps. First a median filter was applied before a constant value was subtracted to remove the background. The resultant images were expanded to fill the full intensity range of 0 to 255. The convex hull of these images were formed and the differences obtained, these images consisting of the shadow areas with intensity being proportional to the depth of the shadows in the original images. For each of three lighting arrangements six intensity profiles were measured, corresponding to values of $\phi$ of 29°, 44°, 56°, 66°, 76°, and 86°. Four values of

---

$^1$VAX Image Processing System

144
threshold were used (namely 40, 60, 80, and 100) and the one-dimensional equivalent of area (and hence \( \Phi_a \)) was obtained by counting the number of pixels exceeding the given threshold. The maximum intensity was used to determine \( \Phi_c \) by setting \( \Phi_c = \text{max}/256 \). The value of \( \phi \) was used to determine \( \Phi_e \). Overall detectability was then calculated using \( \Phi = \Phi_a \Phi_c \Phi_e \), and these were summed over all values of \( \phi \) to obtain mean detectability.

The results of these measurements are summarised in Tables 6.4, 6.5, and 6.6.

<table>
<thead>
<tr>
<th>Thresholds</th>
<th>40</th>
<th>60</th>
<th>80</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \phi )</td>
<td>Max</td>
<td>( \Phi_c )</td>
<td>Area</td>
<td>( \Phi )</td>
</tr>
<tr>
<td>29°</td>
<td>81</td>
<td>0.24</td>
<td>*</td>
<td>0</td>
</tr>
<tr>
<td>44°</td>
<td>62</td>
<td>0.48</td>
<td>7</td>
<td>0.81</td>
</tr>
<tr>
<td>56°</td>
<td>52</td>
<td>0.69</td>
<td>4</td>
<td>0.56</td>
</tr>
<tr>
<td>66°</td>
<td>43</td>
<td>0.83</td>
<td>3</td>
<td>0.42</td>
</tr>
<tr>
<td>76°</td>
<td>36</td>
<td>0.94</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>86°</td>
<td>34</td>
<td>1.00</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

An asterisk indicates that the shadow extends to the boundary of the object and has been taken as zero in these calculations.

Table 6.4: Detectability for \( \beta = 24.0° \).
Table 6.5: Detectability for $\beta = 48.4^\circ$.

Table 6.7 and Figure 6.12 summarise the results for mean detectability.

The final value of interest is $\beta_0$, the value of $\beta$ for which $\Phi$ is a maximum, and to obtain an estimate of this a parabola was fitted to each triplet of points apart from that for a threshold of 100. The resultant values of $\beta_0$ are given in Table 6.8. A comparison of Tables 6.3 and 6.8 shows a broad measure of agreement, certainly within the range of uncertainty imposed by experimental error.
An asterisk indicates that the shadow extends to the boundary of the object and has been taken as zero in these calculations.

Table 6.6: Detectability for $\beta = 61.4^\circ$.

<table>
<thead>
<tr>
<th>Thresholds</th>
<th>40</th>
<th>60</th>
<th>80</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\phi$</td>
<td>Max</td>
<td>$\Phi_e$</td>
<td>Area</td>
<td>$\Phi$</td>
</tr>
<tr>
<td>$29^\circ$</td>
<td>91</td>
<td>0.24</td>
<td>13</td>
<td>1.11</td>
</tr>
<tr>
<td>$44^\circ$</td>
<td>99</td>
<td>0.48</td>
<td>17</td>
<td>3.16</td>
</tr>
<tr>
<td>$56^\circ$</td>
<td>99</td>
<td>0.69</td>
<td>21</td>
<td>5.60</td>
</tr>
<tr>
<td>$66^\circ$</td>
<td>85</td>
<td>0.83</td>
<td>*</td>
<td>0</td>
</tr>
<tr>
<td>$76^\circ$</td>
<td>71</td>
<td>0.94</td>
<td>*</td>
<td>0</td>
</tr>
<tr>
<td>$86^\circ$</td>
<td>59</td>
<td>1.00</td>
<td>*</td>
<td>0</td>
</tr>
</tbody>
</table>

6.10 Conclusions

The effects of the illumination of kiwifruit from two lights symmetrically placed about the camera direction have been explored. As the angle between the camera and the lights is

<table>
<thead>
<tr>
<th>Threshold</th>
<th>40</th>
<th>60</th>
<th>80</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\beta = 24.0^\circ$</td>
<td>0.30</td>
<td>0.14</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$\beta = 48.4^\circ$</td>
<td>3.68</td>
<td>2.93</td>
<td>2.12</td>
<td>1.16</td>
</tr>
<tr>
<td>$\beta = 61.4^\circ$</td>
<td>1.65</td>
<td>2.64</td>
<td>1.16</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6.7: Mean Detectability.
<table>
<thead>
<tr>
<th>Threshold:</th>
<th>40</th>
<th>60</th>
<th>80</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\beta_0$:</td>
<td>$45.0^\circ$</td>
<td>$51.7^\circ$</td>
<td>$46.3^\circ$</td>
</tr>
</tbody>
</table>

Table 6.8: Experimentally Determined Values of $\beta_0$.

increased surface projections such as Hayward hooks become more readily detectable. At the same time a dip appears in the intensity profile and this may need to be compensated for in order to avoid the apparent detection of a false surface blemish. The use of a simple linear scaling is sufficient to compensate for the intensity dip but may result in intensity values that cannot be represented by eight bits. Subsequent digital processing would then need to be carried out as integer values (as represented on 16 or 32 bit microprocessors). A compromise between the conflicting demands for such a system would result in placing the lights at approximately $45^\circ$ on either side of the camera direction. This is a significantly larger angle than is used in current commercial optical graders in kiwifruit packhouses in New Zealand.
Chapter 7

Algorithms for Defect Detection in Kiwifruit

7.1 Introduction

As outlined in the introduction to this thesis, kiwifruit are subject to a range of surface and shape blemishes. Images of kiwifruit may be captured either from a television-style camera such as a CCD camera or from a linescan camera. In either case a frame-grabber and a frame-store are likely to be part of the hardware. The resultant image, stored in memory that can be accessed by one or more computers, is then subjected to a sequence of operations whose purpose is to extract attributes that can be used to assess whether the fruit is defective.

If a shape defect is seen in profile the boundary between the fruit and the background can be used as the basis for shape analysis. Surface blemishes will be evident as changes in intensity. Some blemishes such as proximity marks or insect chewings possess a clearly-defined boundary between defective and unmarked surfaces of the kiwifruit. However blemishes such as sunburn may be very diffuse, merging gently into the surrounding kiwifruit surface. A useful simplification is the fact that most blemishes appear as dark against the lighter kiwifruit surface. Blemishes vary in size from a few pixels up to a significant fraction of the kiwifruit surface, and may be located anywhere on the surface.

The production of software for the detection of shape and surface defects in kiwifruit is based on a number of developments, including the development of filters such as the median filter [137], the use of model building techniques such as the convex hull [31,125], and general research into the detection of defects in images. Batchelor [22] developed a method for detecting spots and streaks in digital images. The initial form of the algorithm was as follows. Let $W$ be the upper bound on pixel intensities in an image $X$, and let $Y$ be a copy of $X$. Then the algorithm consist of the following steps:
1. $X \leftarrow \{W - \text{MAX}(3 \times 3 \text{ neighbourhood of } e), \forall e \in X\}$

2. $X \leftarrow \{W - \text{MAX}(3 \times 3 \text{ neighbourhood of } e), \forall e \in X\}$

3. $X \leftarrow X - Y$

4. Threshold $X$

Step (2) reverses step (1) apart from spots. The algorithm is suitable for defects one pixel wide, but must be modified for wider defects such as bands. In the modified algorithm steps (1) and (2) are both repeated $m$ times where, for an anomaly of width $l$, $2m \geq l$. The computation load will rapidly become severe with increasing defect width.

The remainder of this chapter is organized as follows. In Section 7.2 a survey is made of prior work in the field of blemish detection in horticultural products. Some algorithms to detect surface blemishes are developed in Section 7.3, while algorithms relevant to shape defect detection are studied in Section 7.4. The use of linescan cameras for image capture is analysed in Section 7.5.

### 7.2 Survey of Prior Work

In a review of industrial applications of computer vision in the period 1982 to 1988, Wallace [289] states that the automated inspection of products that are not machine parts is not well established. The well-defined geometric properties of products such as machine parts leads to automated inspection techniques that are not generally applicable to highly-variable objects such as horticultural products. First the development of automated inspection of products such as tomatoes will be described. Work on kiwifruit inspection will be reviewed in subsequent paragraphs.

Rehkugler and Throop [234] have applied a linescan camera to the problem of bruise detection in apples. They report a correlation between machine-based and manual measurements of bruise area ranging from 0.63 to 0.84. Apples pass through the machine at 30 per minute, and are rotating at 60 rpm when they pass the linescan camera. The resultant image of 64 by 250 pixels has a low-pass, row-based filter applied, and this is subtracted from the original image. The thresholded difference is used as the basis of the measurement of the bruise area.

Sarkar and Wolfe [251] describe techniques for the extraction of appropriate features from images of tomatoes. They found moments to be computationally expensive and to be insensitive to fine differences in shape. The thinness ratio, equal to $4\pi A/P^2$ for a shape of area $A$ and perimeter $P$, was not sufficiently sensitive. The most satisfactory shape technique was the minimum curvature of the chain-coded tomato boundary. They explored the use of a global grey-level threshold, but this proved to be too unreliable. Instead they used a thresholded intensity gradient. They needed to mask out the stem
scar, the placement of this mask being critical. The pixel sizes of their 120 by 128
pixel images corresponded to 1.2mm on the tomato surface. They found that an image
of 60 by 64 by 6 bits was sufficient for shape analysis, with 2.1% of the fruit being
wrongly classified. The full image, digitized to 6 bits, was used for the determination
of orientation and for surface blemish detection. Orientation determination had an error
rate of 3.1% and detection of blemishes at the blossom end of the tomato yielded an
error rate of 7.6%. The prototype hardware used by Sarkar and Wolfe [252] was based on
an LSI-11/23 computer and took approximately six seconds per fruit. They employed a
conveyor passing through a diffusing tunnel to achieve uniform, diffuse lighting. Paulsen
and McClure [219] also employ a diffusing tunnel in order to maximise body reflection
and minimise specular reflection on a number of horticultural products.

Wolfe and Swaminathan [296] describe a technique for determining the shape and ori­
etentation of bell peppers. Their technique involves capturing six orthogonal views of the
peppers as they fall through an arrangement of cameras. Orientation was determined by
searching for stem and blossom end centres and computing an axis passing through these
centres. They found that the use of the general axis of symmetry as a measure of sym­
metry was very sensitive to noise on the boundary. Their technique involved dividing the
computed pepper axis into eleven equal intervals, establishing lines perpendicular to the
axis for each interval, and finding the midpoints of these lines. These midpoints defined
the medial axis.

Teh and Chin [278] have used moments for general shape analysis as part of pattern
recognition. They show that higher order moments are more vulnerable to noise, and
that the optimal order of moment is generally less than ten, depending on the level of
noise.

The hairy nature of kiwifruit skins, combined with the relatively subtle nature of some
of the defects, provides problems that are not present in products such as tomatoes.
However the strongly specular nature of tomato skins is absent in kiwifruit, and this
makes it feasible to use directional lighting. In the remainder of this section the work
carried out in the Department of Electrical and Electronic Engineering since 1983 by a
group of students supervised by Dr R M Hodgson will be reviewed.

7.2.1 Area Defect Detection Algorithms

Bailey [19] and Blick [36] looked at three alternative approaches to surface defect de­
tection. The first was based on Batchelor's streak and spot detector [22], but this was
found to be very sensitive to noise and to be unable to detect larger defects. The other
approaches were both based on the development of a model against which the original
image would be compared. A fixed model cannot be used for horticultural products so a
dynamic modelling process was adopted in which the defects were removed and the origi­
nal was subtracted from the defect-free image. Two methods were explored for producing
The first, called valley fill, is illustrated in Figure 7.1. Bailey [19] found that this technique was sensitive to noise which, in the case of kiwifruit, can arise from the hairs on the fruit, and this required prefiltering. Another problem was that the technique was dependent on the illumination and on the position of the defect, with problems being experienced near the edges of the fruit.

The second method uses the convex hull \([31,125]\) of the intensity, and this is illustrated in Figure 7.2. For this to be applied to kiwifruit the background must be excluded from the model, and this is most conveniently done by setting the background to zero. The algorithm proceeds from left to right along rows, with the hull being 'anchored' to zero for any zero pixels. One consequence of this approach is that the convex hull can be easily determined for an image consisting of more than one kiwifruit, provided that there is at least one pixel of zero intensity between each fruit at their closest approaches. For non-zero pixels the slope of the intensity profile is determined. This slope should decrease monotonically in proceeding from left to right, and any regions where the monotonicity is broken corresponds to concavities in the profile. The hull is the convex shape corresponding to the given image with the concavities being replaced by regions of constant slope. The advantage of this method as reported by Bailey [19] was that, compared to the valley fill algorithm, it was significantly less sensitive to the fact that the intensity is reduced near the edges. However it has the same noise and texture sensitivity as does the valley fill, and prefiltering is still required. If the convex hull algorithm is applied to columns as well as rows there is usually some improvement, although this can introduce artifacts because of the stalk. There is also an improvement if a filter such as a median filter is applied to the image obtained by subtracting the filtered original from the convex hull.
Bailey [19] and Blick [36] have summarised their work through the following algorithm:

- Capture image
- Subtract background (assumed to be a constant)
- Normalise intensity (to handle fluctuations in illumination)
- Prefilter (median filter)
- Convex hull
- Compare original with model (subtract)
- Postfilter (median filter)
- Find maximum (reject if above a critical value)
- Threshold (threshold value is experimentally based)
- Measure area (reject if above a critical value)

The thresholded area is a plan area only, with no account being taken of the fact that blemishes near the edges have an actual area significantly greater than their projected area. When Bailey applied this algorithm to a sample of 36 fruit, three fruit, or eight percent, were misclassified. Two of these were because of artifacts near the stalk, so the algorithm needs to be extended in some way to allow for the stalk, or only a single pass of
the convex hull employed. The third misclassification was a point defect that was removed at the prefiltering stage. This matter is pursued in section 7.3. Bailey also suggests the fitting of non-linear segments such as cubic splines to make the convex hull a better model, but points out that such an approach would be very computation-intensive. He suggests the alternative approach of adjusting the convex hull intensities using a histogram modification technique, in which the intensity histogram of fruit without defects is used to modify the histogram of the fruit with defects.

McNeill [188] has studied this surface defect detection algorithm on the Department of Electrical and Electronic Engineering's HIPS\(^1\). His objective was:

firstly to implement a solution on HIPS so that an objective measure of the algorithm's performance could be made, secondly to determine one or more of the bottlenecks that exist in the HIPS solution and suggest ways of overcoming these problems, and thirdly to propose any significant shortcuts that might be applied to the solution to improve the run time of the algorithm, should it prove to be unsatisfactory\([188]\).

As implemented on HIPS the rank filter and the convex hull provide the most serious restrictions on the performance of the algorithm. McNeill noted the work reported by Naylor [205] and by MacKenzie [180] on the development of a VLSI\(^2\) chip for the rank filter. The serial version of the VLSI rank filter chip is projected to perform an operation in 160 milliseconds [180]. According to McNeill [188] other operations in the algorithm take 60 milliseconds, leaving 30 milliseconds for the convex hull. McNeill [188] notes that

Clearly, a simple multiprocessor architecture is most unlikely to succeed in achieving this speedup, and the only alternatives appear to be either to reject the use of the convex hull filter, or to attempt to implement it in hardware.

### 7.2.2 Shape Defect Detection Algorithms

Freeman [105] analysed kiwifruit shapes using two algorithms, the first involving the cosine of the angle between two vectors on the outline, and the second dividing the image into four approximately symmetrical sections and superimposing them. The first technique involves determining, for every pixel in the boundary, the cosine between the vector pointing \(k\) pixels forward and the vector pointing \(k\) pixels backward along the boundary. Since this cosine is \(-1\) for a straight line, values significantly greater than \(-1\) indicate regions of rapid change. Using \(k = 5\), the maximum cosine provided an indication of the presence of a Hayward hook, with the boundaries given in Table 7.1.

---

\(^1\)High-resolution Image Processing System, an 8086-based development system

\(^2\)Very Large Scale Integrated Circuit
<table>
<thead>
<tr>
<th>Class</th>
<th>Cosine Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acceptable</td>
<td>-1.0 to -0.60</td>
</tr>
<tr>
<td>Marginally acceptable</td>
<td>-0.60 to -0.40</td>
</tr>
<tr>
<td>Unacceptable</td>
<td>$\geq -0.40$</td>
</tr>
</tbody>
</table>

Table 7.1: Cosine Detection of Hayward Hooks as Reported by Freeman.

The second method explored by Freeman assumed that the kiwifruit was aligned along its principal axis, and proceeded by finding the centre of gravity and dividing the image into quarters. Asymmetric fruit have non-perpendicular axes, with equi-division of the boundaries being used to determine the longitudinal axis. Freeman used visual analysis of superimposed images.

The work which I carried out as part of a project for a Master of Engineering degree will now be summarised [43]. The shape defect algorithm was based on a combination of two principal measures of asymmetry. The first involves point symmetry as illustrated in Figure 7.3. Given the outline of the fruit image as a set of points $((x_i, y_i))$, the centre of gravity $(x_c, y_c)$ can be determined. For a set of points $(x_a, y_a)$ there exists another set $(x_b, y_b)$ such that $y_b - y_c = y_c - y_a$. The deviation is then defined as

$$Deviation(AB) = |(x_b - x_c) - (x_c - x_a)|$$
and can be calculated for every point on the boundary. The following three statistics are then gathered:

- The mean deviation.
- The maximum deviation.
- The count of ‘excess deviations’, which are defined as the number of deviations which exceed half the maximum deviation.

The second method [43] is based on scan lengths and is illustrated in Figure 7.4. Define

\[
D(n) = \Delta(n + 1) - \Delta(n)
\]

where \( \Delta(n) = L(n)^2 - L(n-1)^2 \)

for scan lines of length \( L \). This method proved to be rather sensitive to noise and not as good as point symmetry at detecting defects. The statistic carried forward to the next stage was the variance in \( D(n) \).

The total of four statistics are combined as follows. Let the measured statistics be \( m_i, i \in [1, 4] \), let \( c_i \) be a set of comparison values, and let these measures have weights \( w_i \) where
\[ \sum w_i = 1. \text{ Then the fruit is rejected if } \]
\[ \sum_{i=1}^{4} w_i c_i > 1 \]

The values of \( c_i \) can be set dynamically.

The use of moments for pattern recognition was explored by Alt [10] who used binary images and calculated moments up to degree 6. His objective was to establish shape characterisers that were invariant with respect to location, size, stretch in a single direction, rotation through a small angle, and that would operate in the presence of some noise. McNeill [188] explored the use of moments as kiwifruit shape descriptors. A binary image is used with the fruit pixels being represented by 1 and the background being represented by 0. The simple moment \( M_{jk} \) is defined as

\[ M_{jk} = \int \int_R x^j y^k \, dx \, dy \]

for the region \( R \), where the coordinate axes have an arbitrary origin and orientation. The area is equal to \( M_{00} \), and the centre of area is at \((\bar{x}, \bar{y})\) where

\( \bar{x} = \frac{M_{10}}{M_{00}} \) and \( \bar{y} = \frac{M_{01}}{M_{00}} \)

A central moment can be defined:

\[ m_{jk} = \int \int_R (x - \bar{x})^j (y - \bar{y})^k \, dx \, dy \]

New axes \((x', y')\) can then be established, centred at \((\bar{x}, \bar{y})\) and rotated through an angle \( \theta \):

\[ \begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{pmatrix} \begin{pmatrix} x - \bar{x} \\ y - \bar{y} \end{pmatrix} \]

Moments with respect to \((x', y')\) are \( \mu_{jk} \). The technique adopted by McNeill was to find the maximum value of \( \mu_{02} \) with respect to \( \theta \) by setting \( \mu_{02} = 0 \), yielding

\[ \tan(2\theta_p) = \frac{2m_{11}}{(m_{20} - m_{02})} \]

The solution to this equation yields two values of \( \theta_p \), corresponding to a maximum and a minimum value of \( \mu_{02} \). The axes \((x', y')\) corresponding to \( \theta_p \) are the principal axes [256] and moments calculated about these axes are invariant to object translation and rotation.

Alt [10] found that a small number of moments is adequate to characterise certain patterns such as characters and to discriminate among patterns of a certain set. However Alt states that moments are sensitive to global rather than local features. This would suggest that when applied to kiwifruit defects such as dropped shoulder would be readily detectable whereas local features such as Hayward hooks would not be evident. The advantage of using moments as shape descriptors is that they are relatively insensitive to noise [188]. However McNeill found that the technique described above for determining
the principal axes would sometimes fail when applied to fruit with severe flats, fans, or dropped shoulders. McNeill reports that $\mu_{04}$ gives a good discrimination for flats and fans, but was unable to find a value of $\mu_{jk}$ that provided good detection of dropped shoulder or Hayward hooks. The criterion suggested from McNeill's work for the rejection of fruit on the basis of flats and fans is

$$\frac{\mu_{04}}{r^6} < 0.35$$

where $r$ is the mean radius.

McNeill [188] explored alternative techniques for shape discrimination. Defining compactness $C$ of a closed curve by $C = P^2/A$ where $P$ is the perimeter and $A$ is the enclosed area, the variation of compactness was too small to be a useful discriminator. For principal axes of length $a, b$ the approximate eccentricity is $\sqrt{1 - (b/a)^2}$, and this can distinguish flats and fans but not dropped shoulder. Defining left-right side deviation by

$$\frac{|d_2 - d_1|}{\frac{1}{2}(d_1 + d_2)}$$

for slices across the fruit, where $d_1, d_2$ are distances to the fruit edge on either side of the principal axis. This measure is satisfactory for flats and fans but is too sensitive to minor boundary variations [188]. McNeill used curvature as a detector for dropped shoulder and for Hayward protrusions. Define the curvature $\kappa$ by

$$\kappa(s) = \lim_{\delta s \to 0} \frac{\theta_{s+\delta s} - \theta_s}{\delta s}$$

where $\theta_s, \theta_{s+\delta s}$ are the pointing angles for arclengths of $s, s + \delta s$ from a point on the boundary. The use of $\kappa$ is too sensitive to noise for $s < 6$, but $s$ needs to be as small as possible for best sensitivity, suggesting that the best results will be obtained using $s = 6$. McNeill's method is to compare $|\kappa|$ with those of a model shape consisting of an ellipse with the same principal axes as the particular kiwifruit being analysed. Hayward hooks can be readily detected using this measure, which also gives reliable results for dropped shoulder [188].

### 7.3 Area Defect Detection

The most promising area defect detection algorithm developed to date is that proposed by Bailey [19] and described above. Two significant operations in this algorithm are the pre- and post-filtering stages, based on median filters. The performance of the median filter as a pre-processor has been analysed by Lai et al [168]. They found that median filtering is effective for images corrupted by impulse-type noise but is not so effective in the case of images corrupted by Gaussian noise. Herman [131] has compared the action of the median filter with analogous linear filters using sifting theory models. As reported by Bailey [19], small surface blemishes can be filtered out of the original image, so the performance of these filters is of importance in considering the minimum size of detectable defects. This issue will be analysed in section 7.3.1.
7.3.1 Performance of Median Filters in the Presence of Small Defects

The median filter is a non-linear filter that takes a full $3 \times 3$ window or a $3 \times 3$ cross window centred on the given pixel, as illustrated in Figure 7.5. The intensity values within the window (nine or five respectively) are ranked and the median value (the fifth or the third value respectively) is taken as the pixel value for the output image. Properties of the median filter have been studied by Hodgson et al [137]. If a full window is used, the median value can be calculated either directly or by separating the operation into two stages in which the median of each of the three rows within the window are calculated, then the median of these three row medians is calculated. This technique has been analysed by Narendra [204] and by Nodes and Gallagher [209]. These methods will be referred to as the full and separable techniques respectively, while the median filter based on a cross window will be referred to as a cross filter.

Consider the case of a dark blemish against a lighter kiwifruit skin (the opposite case, which relates to a few defects such as cracked fungal scars, scale, or some surface deposits, is similarly handled). For a dark pixel to be output from the full technique at least five dark pixels must be present. Similarly, for a dark pixel to be output from the cross technique at least three dark pixels must be present, and from the separable technique at least four dark pixels must be present. For a large rectangle of dark pixels aligned with the row/column directions the effect of the median filter applied to the full technique is illustrated in Figure 7.6 which shows only the darker pixels. The four corner pixels are replaced by lighter pixels, and this shape is then invariant to any further median filtering.
Figure 7.6: Effect of a Full Median Filter on a Large Rectangle.

(The original rectangle is on the left, with the median-filtered rectangle on the right).

in the sense that the dark area will retain its existing boundary. The effect of both the cross and separable techniques is to preserve the shape of large rectangles aligned with the row/column directions, while rounding the corners of other rectangles [204]. Of particular interest is the image resulting from the application of any one of these three filters. Since Bailey's algorithm [19] involves two filtering stages, the following analysis will look at the outcome of a single and a double application of each of the filters on images for small numbers of dark pixels.

A connected pattern of dark pixels, which will henceforth be referred to simply as a connected pattern, is defined as a pattern in which every dark pixel has at least one other dark pixel in either eight-way or four-way contact with it. More formally, if \( P(x, y) \in \{\text{dark}\} \) then eight-way connectivity exists if \( \exists i, j \in [-1, 1], i \times j \neq 0, \) such that \( P(x+i, y+j) \in \{\text{dark}\} \) and four-way connectivity exist if \( \exists i \in [-1, 1] \) such that \( P(x+i, y) \in \{\text{dark}\} \) or \( P(x, y+i) \in \{\text{dark}\} \). Any pattern that is not a connected pattern can be made into a connected pattern simply by removing any isolated dark pixels. For instance, referring to Figure 7.7, when the isolated pixels in 7.7(a) have been removed the result is as shown in 7.7(b).

Since any pattern can be made into a connected pattern, and any isolated pixels would be interpreted as arising from noise, only connected patterns need be analysed. This proceeds as follows. Consider all possible connected patterns based on a two by two grid of dark pixels surrounded by white pixels. There are eleven such patterns, as is illustrated in the top row of Figure 7.8 where '0' represents a light pixel and '1' represents a dark pixel. The technique not included, namely the full filter, produces zero output for all
two by two input patterns. The right-hand column gives the total number of non-zero patterns for that row, and expresses this number as a percentage of the total number of patterns. This procedure can be followed for larger connected patterns, although the time cost of the procedure increases exponentially with the pattern size. To make this problem computationally feasible a sampling technique was used, with a number of samples being used to check convergence.
Of particular interest is: what is the minimum number of pixels in a connected pattern of dark pixels such that the output of a given filter or combination of filters contains at least one dark pixel in a given percentage of occurrences. Obviously patterns such as a line one pixel wide will always be filtered out, so that 100% of such patterns cannot be retained. Table 7.2 shows the appropriate number of pixels in a combination of cases.

<table>
<thead>
<tr>
<th>Occurrences</th>
<th>Full</th>
<th>Cross</th>
<th>Separable</th>
<th>Full$^2$</th>
<th>Cross$^2$</th>
<th>Separable$^2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>50%</td>
<td>9</td>
<td>3</td>
<td>6</td>
<td>11</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>75%</td>
<td>9</td>
<td>3</td>
<td>9</td>
<td>13</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>90%</td>
<td>9</td>
<td>3</td>
<td>9</td>
<td>15</td>
<td>10</td>
<td>14</td>
</tr>
<tr>
<td>95%</td>
<td>9</td>
<td>3</td>
<td>9</td>
<td>15</td>
<td>10</td>
<td>14</td>
</tr>
</tbody>
</table>

Table 7.2: Minimum Numbers of Pixels in Connected Patterns.

Suppose that a given percentage of all kiwifruit blemishes of size 1 $mm^2$ are to be detected. Then the number of pixels that will correspond to this area will be as shown in Table 7.2. Suppose that the extent of the field in the object plane is 10 cm square. Then the minimum number of pixels in the image can be easily calculated and is as shown in Table 7.3. Included in the table is the linear size of the image for square images. There

<table>
<thead>
<tr>
<th>Occurrences</th>
<th>Full</th>
<th>Cross</th>
<th>Separable</th>
<th>Full$^2$</th>
<th>Cross$^2$</th>
<th>Separable$^2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>50%</td>
<td>90,000 $(=300^2)$</td>
<td>30,000 $(=173^3)$</td>
<td>60,000 $(=245^2)$</td>
<td>110,000 $(=332^2)$</td>
<td>70,000 $(=265^2)$</td>
<td>90,000 $(=300^2)$</td>
</tr>
<tr>
<td>75%</td>
<td>90,000 $(=300^2)$</td>
<td>30,000 $(=173^3)$</td>
<td>90,000 $(=300^2)$</td>
<td>130,000 $(=360^2)$</td>
<td>90,000 $(=300^2)$</td>
<td>100,000 $(=316^2)$</td>
</tr>
<tr>
<td>90%</td>
<td>90,000 $(=300^2)$</td>
<td>30,000 $(=173^3)$</td>
<td>90,000 $(=300^2)$</td>
<td>140,000 $(=374^2)$</td>
<td>90,000 $(=300^2)$</td>
<td>120,000 $(=346^3)$</td>
</tr>
<tr>
<td>95%</td>
<td>90,000 $(=300^2)$</td>
<td>30,000 $(=173^3)$</td>
<td>90,000 $(=300^2)$</td>
<td>150,000 $(=387^2)$</td>
<td>100,000 $(=316^2)$</td>
<td>140,000 $(=374^3)$</td>
</tr>
</tbody>
</table>

Table 7.3: Minimum Image Sizes for Detection of Small Blemishes.

are three points to be noted from the table:

1. The use of any filter will prevent a 1 $mm^2$ blemish from being detected using an image of size $100^2$ pixels.

2. The use of single filters on a $256^2$ image will approximately achieve a 75% detection level.

3. The 'cheapest' filter to implement is the cross filter, and a double application of the cross filter can approximately achieve a detection level of 75% on a $256^2$ image.
7.3.2 Surface Blemish Detection Algorithms

The Bailey algorithm provides a reliable but computation-intensive technique for the detection of surface blemishes on kiwifruit. Various alternatives were proposed and examined. There were two objectives to this. Firstly, if an algorithm could be developed that would run faster than the Bailey algorithm on given hardware, or alternatively required less computing resources, then it would have obvious economic significance. Secondly, there was the possibility of finding algorithms that operate on data ‘on the fly’ – that is, in a dataflow mode. The objective of constructing hardware that will perform arithmetic operations on a digitized video stream at full video rates and without using frame grabbers and frame stores is an attractive one. This requires that the value to be written must not depend on values in any other row. If it is a pixel in an intermediate image then it must not depend on values more than a few pixels away in the row. Values may, however, be accumulated to produce some sort of row-based shape measure.

A number of schemes were explored:

- A simple difference operator (approximating the first derivative).
- Curvature, which is approximated by the second derivative.
- Measures of compactness.
- Curve fitting.

These approaches will be examined in the remainder of this section, together with some aspects of the standard convex hull algorithm.

First Derivative

This method is based on differences between adjacent pixels and yields an image in which blemishes are clearly outlined. This difference image can be used directly, or the outlines can be used as the basis for delineating enclosed blemishes. The first method was compared with the standard convex hull algorithm for seven fruit encompassing a wide range of blemishes. A measure of detectability equal to the product of the maximum blemish intensity and the thresholded blemish area was used, the blemish being the filtered difference between the hull and the original. The correlation coefficient between the two sets of results is only 0.453, suggesting at best a weak relationship. The alternative of interpreting thresholded differences as blemish outlines proved unworkable because a minor isolated discontinuity could produce a false edge and accordingly a false blemish which could appear to be of considerable extent.
Curvature

The second differences provide a measure of curvature. If $v[i]$ is the intensity of the $i^{th}$ pixel in a row then the output image consists of pixels given by the absolute value of $v[i - 1] - 2v[i] + v[i + 1]$. The edge of the kiwifruit image produces a large curvature, and this must be stripped off before the resultant image is evaluated. For seven kiwifruit the correlation coefficient for maximum pixel intensity of curvature against the filtered difference between the hull and the original image was 0.945, but the correlation coefficient for area was 0.667. The latter figure could be substantially improved for images with large blemishes by performing a large scale smoothing operation using windows of up to five by five pixels, but the algorithm time became very slow, and the advantage of the dataflow concept of computation was largely lost.

Compactness

Comparing (a) and (b) of Figure 7.9, it seems reasonable to suppose that an effect of the dip in the profile in (a) would be to decrease the area and increase the perimeter relative to the unblemished version in (b). Thus the ratio of area to perimeter should give a measure of intensity proportional to shape and hence of the existence of blemishes. Each row produces one measure and these need to be aggregated in some way. Seven different measures based on this ratio were tried, but none showed a sufficiently large correlation to prompt any further work.
Curve Fitting

As illustrated in Figure 7.9(b), the intensity profile of a kiwifruit approximates to a parabola. This fact has been used to develop an algorithm that accumulates the sums of intensity, intensity squared, and so on, for each row. These sums are then used to calculate a best RMS fit of a parabola, and the difference between the parabola and the actual points is calculated. This technique requires the full row to be stored. Applied to 26 kiwifruit images this technique yielded a linear correlation coefficient of 0.726 for blemish area when compared with the standard algorithm. Improved results were obtained by using higher order curves but the run times were very much increased.

Convex Hull Algorithm

The final step of the blemish detection algorithm based on the convex hull is thresholding. An experimental method has been used to determine the optimal threshold value, and adaptive thresholding might be envisaged in a kiwifruit packhouse. Blemishes which are intense and localised are relatively insensitive to changes in threshold level, but many blemishes such as water stain blend smoothly into the surrounding kiwifruit skin, with the result that the thresholded area varies markedly with the threshold value. A measure of the sensitivity of an image to threshold is provided by sensitivity = ΔA/ΔT where A is the area and T is the threshold value. Three values of ΔT were examined: 1, 5, and 10. The maximum values found in a sample of 26 kiwifruit images are shown in Table 7.4. These present a limit on the accuracy with which thresholded areas can be established.

<table>
<thead>
<tr>
<th>ΔT</th>
<th>Maximum Sensitivity</th>
<th>Percent Change in Area</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>35</td>
<td>10%</td>
</tr>
<tr>
<td>5</td>
<td>130</td>
<td>39%</td>
</tr>
<tr>
<td>10</td>
<td>240</td>
<td>81%</td>
</tr>
</tbody>
</table>

Table 7.4: Sensitivity to Changes in Threshold.

The sensitivity at ΔT = 1 is the absolute limit since this is established by the integral nature of the data. Experiments using VIPS\(^3\) suggest that the very best that could realistically be achieved for setting a threshold value is to the nearest five on an intensity scale of 0 to 255, and that a setting to the nearest ten is more realistic. The change in area is the fraction of the area at a given threshold that is lost when the threshold is increased by the given amount. For instance, for the image showing the greatest sensitivity to changes of threshold of five units, the actual area of the image decreased by 39% when the threshold value was increased by five units. The images showing the largest values of sensitivity were images of kiwifruit having sun burn or sun scorch marks. Some branch rub and leaf roller images also showed a high sensitivity.

\(^3\)VAX Image Processing System

165
7.4 Shape Defect Detection

A variety of measures of shape or profile defect were developed. Most of these were based on intuitive ideas about the ways in which specified defects are evident. For instance, asymmetry between left and right sides can be used as a detector of dropped shoulder. The approaches explored were as follows:

- Profile curvature.
- The RMS fitting of an ellipse to the profile, and looking at features such as goodness of fit and flatness ratio.
- The use of Fourier descriptors.

The remainder of this section will examine each of these in turn.

7.4.1 Profile Curvature

The profile of a kiwifruit can be used to establish the principal axis of the fruit, and the distance of the fruit edge from the axis can be found for the full profile. The differences for this distance between adjacent rows provides a measure of the slope, while the second differences provides a measure which varies as curvature. The stem region must be discarded in this assessment. It was found that large values of the second derivative corresponded to defective fruit, especially of flats and dropped shoulder, but the converse was not true. Large differences in curvature between left and right hand sides did not bear any significant relationship to profile defects. This technique is likely to be sensitive to noise.

7.4.2 Comparison with an Ellipse

The root-mean-square best fit of an ellipse provides a measure which is relatively immune to noise. Three measures were used: the flatness ratio, defined as the ratio of major to minor axes, and both the first and second derivatives of the differences between the actual profile and the ellipse. In all cases these measures provided no useful shape discriminator.

7.4.3 Fourier Descriptors

Zahn and Roskies [303] have developed a system of Fourier descriptors for plane closed curves. The following description of this method is adapted from that of Zahn and Roskies. Let $l$ designate arc length from some given point on the curve, and let $\theta(l)$ designate the
angular direction of the point on the curve that is distant \( l \) along the arc from the current point, such that \( \theta \) is zero at a nominated starting point on the curve. Define a normalized angle

\[
\phi(t) = \theta(\frac{Lt}{2\pi}) + t
\]

where \( L \) is the total curve length and \( t \) is a parameter that defines a point on the curve and varies in \([0, 2\pi]\). The authors show that \( \phi \) is invariant under translations, rotations, and changes of perimeter \( L \). For a circle \( \phi \equiv 0 \), and nonzero values of \( \phi \) measure deviations from a circle. The Fourier components of \( \phi \) are established using

\[
\phi(t) = \mu_0 + \sum_{k=1}^{\infty} A_k \cos(kt - \alpha_k)
\]

where \( A_k \) is the \( k^{th} \) harmonic amplitude, and \( \alpha_k \) is the \( k^{th} \) harmonic phase angle. Thus the set of values \( A_k, \alpha_k \) provide an alternative description of the curve. Zahn and Roskies suggest the use of

\[
R_k = \sum_{j \neq 0} A_j
\]

as a measure of rotational symmetry of order \( k \). They say that in practice it is not normally necessary to go beyond order nine components.

This technique was applied to twelve kiwifruit. The value of high-order Fourier components provide a reasonable determinant of the existence of sharp changes in the profile, but the technique seems to be excessively sensitive to noise.

### 7.4.4 Conclusions on Shape Discriminators

There are two major approaches to deriving shape discriminators:

- Use profile based descriptors, perhaps based on chain codes.
- Use area based descriptors such as moments.

Profile based measures are likely to be sensitive to noise, and a practical application of such techniques is likely to require careful attention to noise suppression. In contrast, the area based descriptors such as moments are much less sensitive to noise, although there is the added computation burden.

### 7.5 The Use of Linescan Cameras

Linescan cameras are used extensively in industrial applications of machine vision [289], being well suited to the flow of objects or materials along a conveyor belt. In the case
of horticultural products such as kiwifruit their three-dimensional character means that if normal cameras or linescan cameras are used directly at least three fruit views are required to provide a full 360° view. An alternative approach is illustrated in Figure 7.10 which shows a sequence of kiwifruit being rotated as they are moved under a linescan camera. The direction of rotation of the top of the fruit is the same as the direction of movement. A similar approach has been described by Rehkguler and Throop [234]. It is not possible for the whole of the fruit right up to the axis of rotation to be surveyed in this manner. However the ends cannot be conveniently analysed by machine vision (at least at this stage), and current algorithms work more reliably if each end is blanked out before analysis commences. Thus only the outer third or half of the maximum radius of rotation is of interest, and it is possible to arrange the speed of rotation in terms of the speed of translation such that the whole surface of interest can be captured.

The remainder of this section is organized as follows. Firstly, a model will be developed to describe the general behaviour of the system. This model will then be used to derive the conditions relevant to the capture of kiwifruit images.

Referring to Figure 7.11, define three sets of axes:

- Axes fixed with respect to the camera are labelled $X, Y, Z$, with the $X$ axis being in the direction of motion of the conveyor and the $Y$ axis being vertical. The origin is placed such that $X = 0$ at the camera.

![Detector of linescan camera](image)

**Figure 7.10:** Kiwifruit Moving and Rotating Under a Linescan Camera.
Axes fixed with respect to the axis of rotation of the kiwifruit but not rotating with it are labelled $x_a, y_a, z_a$.

Axes fixed with respect to the kiwifruit and having their origin on the axis of rotation are labelled $x, y, z$.

**Figure 7.11: Axes for Linescan Camera.**

The $Z, z_a, \text{ and } z$ axes are fixed with respect to each other and can be disregarded in the initial stages of the analysis. Define time $t$ such that $X = x_a = x$ and $Y = y_a = y$ at $t = 0$. Let the linear velocity of the conveyor be $v$ and the rotational velocity of the kiwifruit be $\omega$. Then the following relationships hold:

\[
\begin{pmatrix}
Y \\
X
\end{pmatrix} = \begin{pmatrix}
1 & 0 & 0 \\
0 & 1 & vt
\end{pmatrix} \begin{pmatrix}
y_a \\
x_a \\
1
\end{pmatrix}
\] (7.1)

and

\[
\begin{pmatrix}
y_a \\
x_a
\end{pmatrix} = \begin{pmatrix}
\cos(\omega t) & -\sin(\omega t) \\
\sin(\omega t) & \cos(\omega t)
\end{pmatrix} \begin{pmatrix}
y \\
x
\end{pmatrix}
\] (7.2)

These can be conveniently combined to yield

\[
\begin{pmatrix}
Y \\
X
\end{pmatrix} = \begin{pmatrix}
\cos(\omega t) & -\sin(\omega t) & 0 \\
\sin(\omega t) & \cos(\omega t) & vt
\end{pmatrix} \begin{pmatrix}
y \\
x \\
1
\end{pmatrix}
\] (7.3)
Since $X = 0$ in the plane of the camera, it follows that

$$ysin(\omega t) + xcos(\omega t) + vt = 0 \quad (7.4)$$

If the kiwifruit is assumed to have circular symmetry then

$$x^2 + y^2 = r^2 \quad (7.5)$$

for all points on the surface, where $r$ is constant for any particular value of $z$ but varies along the fruit, that is as $z$ varies. Let $\delta x$, $\delta y$, and $\delta s$ be the increments in the $x$, $y$, and surface directions respectively for a single scan period of the linescan camera, where

$$(\delta s)^2 = (\delta x)^2 + (\delta y)^2. \quad (7.6)$$

Let $T$ be the period of a single scan, and let a prime indicate differentiation with respect to time. Then $\delta x/T = x'$, and similarly for $y'$. Combining (7.4) and (7.5) with (7.6), an expression for $\delta s$ can be obtained as a function of time $t$:

$$\delta s = \left(\frac{v}{\sqrt{r^2 - v^2 t^2}} + \omega\right)Tr \quad (7.7)$$

An example of $\delta s$ is given in Figure 7.12 which shows the spacing for two radii: the inner curve being for a radius equal to 75% of that of the outer curve. Note that the time axis maps on to the $x$-axis, so that the curves in Figure 7.12 can be interpreted as scan spacing versus circumferential position. As can be seen the mapping from kiwifruit surface to linescan image is relatively linear over much of its range, and only at the extreme ends of the range are significant losses in resolution likely. This can be overcome by ensuring that the begin and end of the image capture overlap, so that a small band of the kiwifruit surface is captured twice. This will also ensure that any blemishes that happen to coincide with the start of the scan, and hence are incompletely captured, will be fully captured at the start of the second revolution.

From (7.7), $\delta s \to \infty$ as $t \to \pm r/v$. Thus the total image capture time is $T_{total} = 2r/v$, and the number of rotations performed in this time is

$$\text{rotations} = \frac{\omega r}{\pi v}. \quad (7.8)$$

The condition that the number of rotations must exceed one thus translates into $\omega > \pi v/r$ where $r$ is the radius of the fruit at any specified point. If $r_{min}$ is the smallest radius of the fruit that is to be scanned then the angular velocity can be specified in terms of the linear velocity by

$$f > \frac{v}{2r_{min}}$$

where $f$ is the number of rotations per second ($f = \omega/2\pi$). Some representative values for the minimum number of revolutions per second and the corresponding image capture times are shown in Table 7.5. One objection to this scheme is that seriously out-of-round fruit will, at best, produce seriously distorted images. However such fruit will not be export quality and thus will be rejected on shape criteria. The higher rotational speeds quoted...
Figure 7.12: Example of the Scan Line Spacing for a Linescan Camera.

(The inner radius is 75% of the outer radius).

<table>
<thead>
<tr>
<th>Minimum radius (mm)</th>
<th>Linear velocity (m/second)</th>
<th>Revolutions (per second)</th>
<th>Total time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0.1</td>
<td>10</td>
<td>0.1</td>
</tr>
<tr>
<td>20</td>
<td>0.1</td>
<td>2.5</td>
<td>0.4</td>
</tr>
<tr>
<td>5</td>
<td>0.4</td>
<td>40</td>
<td>0.025</td>
</tr>
<tr>
<td>20</td>
<td>0.4</td>
<td>10</td>
<td>0.1</td>
</tr>
</tbody>
</table>

Table 7.5: Requirements for a Linescan System.

in Table 7.5 may be impractical on rotating rollers, but modern fruit-handling techniques include projecting fruit through the air, and such fruit could easily be given sufficient rotation. Alternatively more than one kiwifruit image could be captured at any one time. Suppose that the system must process an average of four kiwifruit per second and the minimum radius to be scanned is 5 mm. Table 7.6 sets out the requirements for such a system, including the required number of linescan elements under the assumption that each fruit requires 256 elements. The rotational speeds are expressed in RPM (revolutions per minute).
Table 7.6: Some Options for a Multi-Fruit Linescan System Handling Four Fruit Per Second.

<table>
<thead>
<tr>
<th>Number of fruit</th>
<th>Velocity (cm/second)</th>
<th>Frequency (RPM)</th>
<th>Linescan elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
<td>240</td>
<td>256</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>120</td>
<td>512</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>60</td>
<td>1024</td>
</tr>
<tr>
<td>6</td>
<td>1.3</td>
<td>40</td>
<td>1536</td>
</tr>
<tr>
<td>8</td>
<td>1.0</td>
<td>30</td>
<td>2048</td>
</tr>
</tbody>
</table>

An example of an image captured on a linescan camera at the Auckland Industrial Development Division of the Department of Scientific and Industrial Research is shown in Figure 7.13. The image-processing operations outlined in the preceding sections of this chapter can easily be applied to linescan images, with the considerable advantage that only one image is required per fruit.

Figure 7.13: Example of an Image of a Kiwifruit Captured by a Combination of Translation and Rotation.
7.6 Conclusions

Work carried out by Bailey and McNeill in the field of area blemish detection and shape defect detection in kiwifruit has established algorithms that are reasonably reliable when measured against the objectives of the application of machine vision to kiwifruit packhouses. Furthermore, these algorithms are relatively insensitive to noise, which is likely to be an important factor when this technology is moved from the laboratories to the factories. Attempts to find alternative techniques were unsuccessful. Whenever a new algorithm started showing promise, images were found on which it was unsuccessful in its simplest state. This lead to various enhancements, each exacting a time penalty, to cater for the variations in the kiwifruit images. However it is to be noted that the basic algorithms do not always perform well, and results to be reported in Chapter 8 provide some enhancement to the basic surface blemish detection algorithm. The use of linescan images provides a considerable simplification to the problem of capturing images for a full kiwifruit.
Chapter 8

Pyramidal Vision Applied to Automatic Inspection

8.1 Introduction

The human visual system contains approximately $10^7$ cones, which are sensitive to colour, and approximately $10^8$ rods, which are sensitive to intensity only, in the retina of each eye [283]. Behind the retina lies the optic nerve, the lateral geniculate, and the visual cortex, which itself is stratified [274]. This system is capable of recognizing complex scenes in from 30 to 800 milliseconds; this is in spite of the fact that the amount of time taken to bridge the synapses over which neurons fire other neurons is approximately 1 to 2 milliseconds [283]. The implication is that vision involves at most a few hundred sequential processing steps, and that the high processing rate is achieved through the simultaneous actions of a very large number of neurons.

A processing structure that is variously called a pyramid, a quadtree, a recognition cone, or a processing cone has been proposed as a means of achieving a high degree of parallelism for tasks such as object recognition in machine vision. In its simplest form the pyramid consists of a base with $p^m$ by $p^n$ pixels, where typically $p = 2$ or $p = 3$. This is called a 'retinal memory' by Uhr [284]. Subsequent layers contain $p^{m-1}$ by $p^{n-1}$ values, $p^{m-2}$ by $p^{n-2}$ values, and so on until a single row or column exists at the apex. Only the base need contain picture elements, as values in subsequent layers can be any derived numbers. The total number of layers is $\min\{m,n\}$. Each layer is built from the layer below using some rule such as the mean of the $p$ by $p$ contributing pixels, which are called the 'sons' of the given pixel. Each son regards the derived pixel as its 'father' in the pyramid. The pyramid may not need to be calculated right to the apex, but may instead be truncated at a final layer, this layer containing the minimum amount of information that is considered useful for a particular application.

A useful feature of pyramids is as follows. Let the distance between a pair of pixels in
an array be defined as the total number of pixels that must be traversed to go from one pixel to the other. Then the distance between any random pair of pixels in an $N^2$ array varies as $O(N)$ where $N = p^n$ for some integer $n$. However if the path between the pair of base pixels proceeds via the apex of the pyramid, the number of moves required is $2n = 2 \log_p N$. Thus interpixel distance via the apex of the pyramid varies as $O(\log N)$.

Uhr [284] describes pyramids in terms of layers, the layers being derived by transforms which are applied to every pixel in a layer. Each transform derives a value from a window or set of windows. Several transforms might be applied before an output value is written into the next layer. The output could be either a numerical or a Boolean value. Thus pyramid layers other than the base need not contain images; they could contain various data abstractions such as edge information.

Consider the problem of implementing a pyramid in hardware. Let a single 'processor' be devoted to each element in the pyramid where in this context a processor could be a discrete device such as a microprocessor or could be part of a device such as a VLSI chip. Suppose a pixel in a layer other than the base has been derived from $p^2$ pixels in the layer below. Each pixel will be connected to one father, unless an overlapping pyramid is constructed, in which case the number of fathers would be greater than one. If connectivity has been preserved, so that this pixel's neighbours correspond in direction to the corresponding sons in the layer below, then each pixel will need either 4-way or 8-way nearest-neighbour connections, and these two cases will be considered separately. For four-way connectivity the total number of links required between processors is $p^2 + 5$ and if, as is commonly the case, $p = 2$, then each processor must possess 9 links. Similarly, for eight-way connectivity, the total number of links required between processors is $p^2 + 9$, giving a total of 13 links processors per processor. Thus a full hardware implementation of a pyramidal architecture will involve specially-designed processors, and considerable interest will continue to be focussed on techniques for achieving a partial hardware implementation, commonly based on shared links.

The remainder of this chapter is organized as follows. In section 8.2 a brief review of the literature is presented, with emphasis on those aspects that are of direct relevance to the material to be presented later in this chapter. Section 8.3 specifies the objectives of pyramidal vision as applied to machine vision. In section 8.4 the problem of using a processor such as the transputer for the construction of pyramids is analysed. The specific application of pyramids to surface blemish detection in kiwifruit is presented in section 8.5. Some aspects of the application of special-purpose hardware to pyramid construction are explored in section 8.6.

8.2 Literature Survey

The initial proposals for pyramidal vision systems were made by Kelly, Klinger, Dyer, and Uhr, and their contributions are summarised in [283]. A proposal for a pyramidal vision
system has been made by Tanimoto and Pavlidis [276]. They investigated the tree structure properties of pyramids, and proposed an edge detection algorithm. Their pyramid is based on the mean of the 2 by 2 son nodes. A general introduction to pyramidal vision is also contained in [249,274,283]. Tanimoto [275] provides a wide-ranging review, including the design of a pyramid-machine architecture, software tools, types of algorithms, the construction of pyramids, the use of pyramids as search trees, and the propagation and relaxation of pyramids.

A critical aspect of pyramidal vision is the way in which the pyramid is constructed. This is concerned with the way in which a node value is derived from the values of its sons. An iterative algorithm in which the link pattern is under software control has been proposed by Burt et al [55]. Their algorithm has three phases. In the first phase the normal son-father step is modified by associating four values in a given layer with a single son in the layer below. The 'legitimate' father is the father whose value is closest to that of the son. In the second phase image properties and associated areas are calculated, working from the base to the apex of the pyramid. Finally, in the third phase, new segment values are calculated, where a segment is defined as a homogeneous region. These three phases are iterated until there are no further changes, and this typically involves about ten iterations. A similar scheme to Burt et al has also been used by Hong et al [140]. Antonisse [13] has proposed a scheme in which there is no linkage from a pixel if the most similar parent differs from this pixel by more than $m\sigma$ where $m$ is specified and $\sigma$ is the standard deviation in the neighbourhood. Unlinked pixels become the roots of new sub-pyramids.

The particular issue of node linking in overlapped pyramids has been examined by a number of other authors. Cibulskis and Dyer [69] have proposed an image segmentation algorithm for multiple levels of resolution, such that at increasingly lower levels objects become isolated or 'rooted'. The result of this procedure is an embedded 'forest', with one tree rooted at the pyramid's apex for the background, and other trees representing objects. However uneven illumination and highlighting can cause artifacts. Ferretti [102] considers mapping efficiency and the use of dynamically-reconfigurable links. Burt and Adelson [54] apply a Gaussian filter over a 5 by 5 window to obtain a smoothed, compressed image and subtract an expanded version of this from the original image to obtain a 'Laplacian-like' image. The objective of this technique is to obtain data compression. This approach has been modified by Peleg and Federbusch [220] who use adaptive sampling based on a technique in which the Laplacian is used as a measure of the 'busyness' of parts of the image. 'Busy' parts of the image are passed to upper levels of the pyramid in preference to comparatively featureless parts of the image. To preserve connectivity the average 'busyness' is computed for each row and for each column. The reduction proceeds in two stages: by row, then by column.

General algorithms for the construction of pyramids have been developed by a number of researchers. Levine [176] uses colour, texture, or edges, with picture segmentation consisting of the iterative application of the processes of clustering, projection, and se-
lection. Hong et al [140] provide a scheme in which smoothing is carried out within regions. The use of pyramids for edge detection has been an area of extensive development. Tanimoto and Pavlidis [276], Hanson and Riseman [126], Levine [176], Hong et al [141], Tanimoto [274], and Rosenfeld [243] discuss edge detection algorithms. The method presented by Hong et al will extract edges in noisy environments. Their technique involves edges being detected high in the pyramid (at reduced resolution) and following these down to the lowest level. Rosenfeld's technique is based on the determination of bimodality in which values are partitioned into two subsets in such a way that the variance of each subset is minimised. According to Rosenfeld, this is thought to be the way in which human vision operates. Samet [248] and Dyer et al [94] have examined the way in which boundary codes such as chain codes can be used in pyramidal vision. Uhr et al [284] have compared pyramid algorithms as executed on various systems including a single processor, pipelines, parallel arrays, and pyramids of computers. They conclude that a large degree of parallelism is essential for real-time image processing.

8.3 The Potential for the Application of Pyramidal Vision

In this section the application of pyramidal vision to machine vision systems in general and kiwifruit inspection in particular will be analysed. The objective of the work described in this section is to determine under what circumstances pyramidal vision is applicable, and to provide an overview of the constraints on such a system.

Industrial vision systems must extract certain features from an image, and this should occur in minimum time for given hardware. The problem of judging the usefulness of a pyramidal vision system can thus be stated as the determination of the speed with which required image features can be extracted. The overall concept of pyramidal vision involves data compression while retaining relevant features such as edges in the case of shape analysis and surface features in the case of surface blemish detection in kiwifruit. The resultant image can then be analysed with less computing resource. The remainder of this section is organized as follows. In Section 8.3.1 some basic aspects of the pyramid are analysed from the point of view of the application of the pyramidal vision system. The asymptotic behaviour of pyramidal vision provides an understanding of the choices in designing a system, and this subject is analysed in Section 8.3.2. Three examples are used as the basis of pyramid design in Section 8.3.3.

8.3.1 Optimum Structure of the Pyramid

Define the computing load $L$ for a given image processing algorithm as the time taken to perform that image processing algorithm per image pixel. Let the $v^{th}$ level of an image pyramid have $r_v$ rows and $c_v$ columns, so that the base (the original image) is of size $r_0$ by $c_0$. Let the pyramid be such that a $p$ by $p$ square of pixels at level $v - 1$ maps onto a
single pixel at level \( v \) (typically \( p = 2 \) or \( 3 \)). Then

\[
r_v = \frac{r_0}{p^v} \quad \text{and} \quad c_v = \frac{c_0}{p^v}
\]

Let the time to map \( p^2 \) pixels on layer \( v - 1 \) to a single pixel on layer \( v \) be \( t \), where \( t \) is a function of \( p \) but is assumed to be independent of the pixel values. Then the time taken to map layer \( v - 1 \) onto layer \( v \) is \( r_v c_v t \) if this mapping occurs sequentially. Thus the time to map from the base to layer \( v \) is

\[
\sum_{i=1}^{v} r_i c_i t = r_0 c_0 t \sum_{i=1}^{v} \frac{1}{p^{2i}}
\]

Suppose that the image processing algorithm is applied at level \( v \). Then the total time is

\[
T = r_v c_v L + r_0 c_0 t \sum_{i=1}^{v} \frac{1}{p^{2i}}
\]

(8.1)

Let \( \gamma = \frac{t}{L} \), that is, the ratio of the layer mapping time per pixel to the image processing algorithm time per pixel. Then (8.1) becomes

\[
T = r_0 c_0 L \Gamma
\]

(8.2)

where

\[
\Gamma = \frac{1 + \gamma \sum_{j=0}^{v-1} p^{2j}}{p^{2v}}
\]

(8.3)

\( \Gamma \) is a monotonically decreasing function of \( v \) and a monotonically increasing function of \( \gamma \). Given that \( T \propto \Gamma \), the minimisation of \( \Gamma \) is of primary concern. In machine vision applications of pyramidal vision it is likely that the values of \( \gamma \) and \( v \) will be closely related since the top layer of the pyramid is determined by the way in which the required data are extracted from that layer. For example, for the kiwifruit surface blemish detection algorithm, the simple rank-based mapping discussed below means that images of at least, say, \( 20^2 \) pixels are required, whereas it could be envisaged that a mapping which maintained a non-zero boundary around the kiwifruit outline would allow the convex hull to be based on a still smaller image. For a layer containing a given number of pixels, a large value of \( p \) implies a small value of \( v \), and vice versa. As \( p \) is increased, the value of \( t \), and hence of \( \gamma \), is increased. Alternatively, as \( v \) is increased, \( \gamma \) is decreased, thus decreasing the number of pixels \( r_v c_v \) in the layer, and hence decreasing the contribution of \( L \) to the total time \( T \). The optimum strategy, which will minimise \( \Gamma \), can only be determined by reference to specific timings for the mapping and algorithm.

8.3.2 Asymptotic Performance

To obtain expressions for the behaviour of the pyramid as the number of layers tends towards the maximum possible, let this maximum be represented by infinity. The resultant error is typically of the order of one in \( p^{2v_m} \) where \( v_m \) is the value of \( v \) for the apex of the
pyramid. Consider the asymptotic behaviour of $\Gamma$ as $v \to \infty$, that is, towards the apex of the pyramid. Equation (8.3) gives
\[
\lim_{v \to \infty} \left( \frac{\Gamma}{\gamma} \right) = \sum_{i=1}^{\infty} \frac{1}{p^{2i}}
\]
Set $x = 1/p^2$, where $|x| < 1$. Then
\[
\lim_{v \to \infty} \left( \frac{\Gamma}{\gamma} \right) = \sum_{i=1}^{\infty} x^i = \frac{x}{1-x}, \quad |x| < 1
\]
Substituting for $x$:
\[
\lim_{v \to \infty} \left( \frac{\Gamma}{\gamma} \right) = \frac{1}{p^2 - 1} \quad (8.4)
\]
For a single-layer pyramid (with $v = 0$), $\Gamma = 1$. Therefore there will be a net time saving if $\Gamma < 1$. From (8.4),
\[
\Gamma < 1 \Rightarrow \gamma < p^2 - 1 \quad (8.5)
\]
and since $p \geq 2$, these yield a minimum value of $\gamma = p^2 - 1 = 3$.

If the pyramid must be reconstructed, starting at some layer $v$ and working downwards to the base, then (8.1) becomes
\[
T = \frac{\tau_0 c_0}{p^2 v} (L + t_u \sum_{j=0}^{v-1} p^{2j} + t_d \sum_{j=0}^{v-1} p^{2j})
\]
where $t_u$ is the time per pixel to go up the pyramid and $t_d$ is the time per pixel to go down the pyramid.

In the case of the Transputer Image Processing System (TIPS) operating on a single transputer, $L \approx 836\mu s$ for the standard kiwifruit surface blemish detection algorithm\(^1\) and $t \approx 340\mu s$ for $p = 3$, where the pyramid is constructed purely in software. Thus $\gamma = 0.41$ and $p^2 - 1 = 8$, so that the right-hand side of (8.5) is satisfied. If the pyramid were constructed using special-purpose hardware it is probable that $t$ would be much less than $L$. Thus use of pyramidal vision will always show a positive speedup when applied to surface defect detection in kiwifruit. The remainder of this section will explore the upper bounds for the number of layers in various machine-vision pyramids.

Suppose $t \propto p^{2j}$, where $f = 1$ for worst-case sorts such as bubble sorts, and $f \leq 1$ in general. Set $t = \alpha p^{2j}$. Then (8.3) becomes
\[
\Gamma = \frac{1 + \frac{\alpha p^{2j}}{L} \sum_{j=0}^{v-1} p^{2j}}{p^{2v}} = p^{-2v} + \alpha \frac{1}{L} \sum_{j=0}^{v-1} p^{2(j+f-v)}
\]
Since $j \leq v - 1$ and $f \leq 1$, it follows that $j + f - v \leq 0$. Therefore $\Gamma$ is a monotonically decreasing function of $p$. The prudent approach in deciding how to construct the pyramid would thus appear to be to choose $p$ as large as possible, subject to the various other restrictions such as those imposed by the technique used to construct the pyramid.

\(^1\)See Chapter 7.
8.3.3 Data Requirements in a Pyramid

The factor to be determined now is the layer of the pyramid corresponding to a specified data volume, $u$. If the original image has $r_{0c0}$ pixels then the layer number $v$ is given by the solution of

$$u = \frac{r_{0c0}}{p^{2v}}.$$ 

This is

$$v = \lfloor \log(\frac{r_{0c0}/u}{2\log p}) \rfloor \quad (8.6)$$

where $\lfloor \cdot \rfloor$ represents the truncation of the expression. Some example values for a 256 by 256 initial image are given in Table 8.1.

<table>
<thead>
<tr>
<th>Minimum pyramid elements, $u$</th>
<th>Number of Layers, $v$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>for $p = 2$</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>6</td>
</tr>
<tr>
<td>100</td>
<td>4</td>
</tr>
<tr>
<td>1000</td>
<td>3</td>
</tr>
<tr>
<td>10000</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 8.1: Maximum Number of Layers Starting From a 256² Image.

Now consider the problem of determining the minimum amount of data required at the final layer of the pyramid. Note that in a general pyramid only the first layer need contain pixels, so that the final layer can contain any derived information such as measures of shape or area. These layers can be encoded in any fashion, although it is likely to be in the form of bytes, 16-bit integers, or 32-bit integers depending on the hardware implementation. Let the required information in this layer be $r$ bits. Let this level be encoded in $e$ bits per pyramid element, where $e = 8$ for one-byte pixels and $e = 32$ for one-word T414 transputer values. Note that in this context a pyramid element need not be a pixel, but could be some abstraction. Then the minimum number of pyramid elements required to hold this information is $u = r/e$. This will be analysed with respect to three applications in the field of machine vision applied to kiwifruit — chain-coded outlines, areas, and surface blemish detection.

Firstly, consider the case of a chain-coded outline, consisting of, say, 500 points encoded at 3 bits per point. The total data volume is thus 1500 bits, which can be encoded in a minimum of 188 pyramid elements at 8 bits per pyramid element or 47 pyramid elements at 32 bits per pyramid element. These represent absolute minimums, and if each link of the chain code occupies one byte the data volumes become 500 pyramid elements of one byte per element or 132 pyramid elements of 32 bits per element respectively. These values can then be used to determine the final layer of the pyramid. Using (8.6) and an original image of 256² pixels, a pyramid based on $p = 2$ will extend to $v = 3$, thus
containing four layers including the base. Similarly, if \( p = 3 \) then the pyramid will contain a total of three layers. If \( p \) is in the range of four to eleven, the pyramid will contain just two layers, while for \( p > 11 \) a pyramid cannot be constructed.

As a second example, consider the detection of surface blemishes. This will require a totally different pyramid from that used for chain-coding the outline. The required information is approximate area, say 8 bits, and approximate intensity or depth of the blemish, say 8 bits. There is thus a total data volume of 2 bytes. Based on an original image of \( 256^2 \), (8.6) yields \( v = 7 \) (eight layers) for \( p = 2 \) and \( v = 4 \) (five layers) for \( p = 3 \).

Finally, consider the data needed for the application of the blemish detection algorithm based on the convex hull. In this application image pixels have been carried through to the final layer, so that these pyramid elements can be interpreted as intensity levels. The minimum blemish area is one pixel, and this pixel must have non-blemished pixels on at least three sides for the convex hull algorithm to work. Referring to Figure 8.1(a), the lower limit is seen to be four pixels. A more realistic limit is shown in Figure 8.1(b), that is, a lower limit of six pixels. If each layer of the pyramid is constrained to be square the minimum area will be nine pixels. An original image size of \( 256^2 \) pixels means that \( v = 6 \) (seven layers) for \( p = 2 \) and \( v = 4 \) (five layers) for \( p = 3 \).

### 8.4 Embedding Pyramids in Flat Arrays

Various schemes have been proposed for implementing pyramids in specially designed hardware, and many machines have been or are being built. Tanimoto [274] describes a pyramid machine called PCLIP, based on a set of simple cellular processors and a single controller. Uhr [282] has explored ways in which one or more pyramids can be combined with MIMD network nodes, with each pyramid occupying one node. The PAPIA pyramidal machine described by Cantoni [58] and by Gerardi [117] uses one-bit processors, each with 256 bits of RAM, and connected by switching elements. The Erlangen General Purpose Architecture (EGPA) described by Fritsch [108] contains 21 processing nodes, each consisting of one Intel 8086/8087 pair. The number of nodes in each layer is 16, 4, and 1 respectively. Efficiency is commonly measured as processor utilisation, and the lowest efficiency measured on this machine for any task has been 65%. Merigot et al [192] describe an architecture based on a specially designed one-bit processor with 128 bit dual-port memory. Their goal is to develop a machine with a base of 128 by 128 bits. Burt et al [56] have described a pyramid machine based on five computational units: a filter for 5 by 5 convolution, a decimator which is used to discard parts of the image, an expander to insert zeroes, an ALU to form the sum and difference between images, and several image frame stores. The base consists of 256 by 256 processors, and the machine contains a total of 85K processors. Their plan was to place 16 processors onto a single VLSI chip. Maloberti [181] has reviewed the use of VLSI technology for the implementation of pyramid architectures.
In general, inter-processor links involve at least four nearest-neighbours, at least four sons, and one father, giving a minimum of nine links. More complex schemes [275] will require more links. The hardware costs can be substantial if all desirable interprocessor links are provided. An alternative approach is to embed the pyramid in an existing multiprocessor structure. Stout [271] has investigated the mapping of pyramids onto hypercubes, and points out that pyramids cannot be embedded in hypercubes in such a way that neighbours in the pyramid are mapped to neighbours in the hypercube. Reeves [233] makes use of the ‘perfect shuffle’ in embedding pyramids in flat arrays. Referring to Figure 8.2 which illustrates the perfect shuffle for a 4 by 4 base, the rearrangement of nodes is designed to improve the efficiency of internode communication in moving up or down the pyramid.

Two approaches will be described in the next two sections, and their efficiencies will be compared to the scheme of Reeves. They are both based on the use of the transputer. The first is likely to be more acceptable if the processing load in the upper parts of the pyramid are comparatively light, while the second is optimised for heavier processing loads in upper layers.
8.4.1 A Transputer Pyramid With Overlapping Layers

One possible mapping, which leads to unequal loads on transputers but may be acceptable for problems in which most of the computation is done on the base of the pyramid, is as follows. The upper layers are placed on the same processors as the base layer, so that certain designated processors will be both layer 0 and layer 1 processors, some will be layer 0, 1, and 2 processors, and so on. In principle this allocation could be done dynamically, so that a comparatively lightly loaded processor was nominated to hold subsequent layers. However in this thesis only static allocation will be considered. Edge connections will be excluded from this analysis, so that only a pure cellular array will be considered.

The criterion for optimality will be the minimisation of communications path lengths within and between layers in the pyramid. Since the base layer is a cellular array, nearest-neighbour communications exist in this layer. The problem therefore is to place other layers onto this array such that:

- Processors within any one layer are as close together as possible.
- Communications from one layer to the next are as short as possible.

Let the transputer cellular array be \( r_0 \) by \( c_0 \). The father of transputer \((r, c)\) on layer \( v - 1 \) is \((r_v, c_v)\) on layer \( v \). Define the mid-row \( r_m = r_0/2 - 1 \), and similarly for the mid-column \( c_m \). Let the window size for the mapping from one layer to the next be \( p \). Define the truncated row \( r_t = p(\lfloor \frac{r}{p} \rfloor) \), and similarly for the truncated column \( c_t \), where \( \lfloor \cdot \rfloor \) is the truncation operator, so that only the integer part of the operand is used. Consider the following scheme:

\[
\begin{align*}
    r_v &= r, \quad r_t > r_m \\
    &= r + 1 - \frac{p}{2}, \quad r_t = r_m \\
    &= r + p - 1, \quad r_t < r_m
\end{align*}
\]

with a similar expression for \( c_v \). An example of this mapping is shown in Figure 8.3 where \( r_0 = c_0 = 8 \) and \( p = 2 \).
Proposition: The placement of pyramid nodes defined by \((r_v, c_v)\) provides a pyramid with communications costs minimised.

The communications costs, defined by the number of links required between different layers and within layers, will now be derived. Firstly, consider the communications costs for \(p = 2\). To build one processor’s contents of layer 1 from layer 0, that processor’s layer 0 contents do not need to be sent over any interprocessor link, two processors are adjacent, and one processor is at distance 2. This could be summarised as:

1 communication of distance 0  
2 communications of distance 1  
1 communication of distance 2,

a total distance of 4. Similarly, to construct one processor’s contents of layer 2 the communications costs are:

1 communication of distance 0  
2 communications of distance 2  
1 communication of distance 4,

a total distance of 8. Similarly, layers 3, 4, \(\ldots\) have communications costs of 16, 32, \(\ldots\). In general, layer \(v\) requires a total of \(2^{v+1}\) links. A similar analysis for \(p = 3\) leads to the conclusion that layer \(v\) requires a total of \(2 \times 3^{v+1}\) links. In general, with a pyramid ratio
of $p$, the total number of links required by layer $v$ is given by
\[ T_p = (p - 1)p^{v+1} \]  
(8.8)

Since $p^2$ messages are involved in constructing a single processor’s contents in layer $v$, the mean link length is $(p - 1)p^{v-1}$. The total communications cost of constructing layer $v$ is thus
\[ \kappa_v = \frac{r_0c_0}{p^2v} (p - 1)p^{v+1} = r_0c_0(p - 1)p^{1-v} \]  
(8.9)

and the total cost of constructing the pyramid up to layer $v$ is
\[ \kappa = r_0c_0(p - 1) \sum_{i=1}^{v} p^{1-i} = r_0c_0p(p - 1) \sum_{i=1}^{v} \frac{1}{p^i} \]  
(8.10)

The asymptotic behaviour of (8.10) will be used to give an indication of the overall performance. In particular, as $v \to \infty$,
\[ \sum_{i=1}^{v} \frac{1}{p^i} \to \frac{1}{p - 1} \]

Therefore
\[ \kappa \to r_0c_0p \]  
(8.11)

Thus the total cost decreases as $p$ increases, and a strategy based on minimising communications costs would choose $p$ as large as possible.

This will be compared with the communications costs for an ‘ideal’ pyramid constructed from processors equipped with sufficient links to service all connections. For a layer mapping of $p$ by $p$ there are $p^2$ sons, four nearest neighbours, and one father, giving a total of $p^2 + 5$ links. Then the cost of constructing layer $v$ is
\[ \kappa_v = \frac{r_0c_0}{p^{2(v-1)}} \]

so that the total cost for the complete pyramid to layer $v$ is
\[ \kappa = r_0c_0p^2 \sum_{j=1}^{v} \left( \frac{1}{p^2} \right)^j \]

Thus
\[ \kappa \to \frac{r_0c_0p^2}{p^2 - 1} \]  
(8.12)

The ratio of communications costs, the actual cost to the ideal cost, can be obtained from (8.11) and (8.12), yielding
\[ R_{comm} = \frac{p^2 - 1}{p} \]  
(8.13)

This is to be compared to the ratio of actual to ideal links required, namely
\[ R_{links} = \frac{4}{p^2 + 5} \]  
(8.14)
Since these links are implemented in hardware they represent a real cost. In order to provide an overall comparison of these two schemes a composite cost \( C \) can be formed as the product of the number of links and the communications costs. Thus taking the asymptotic behaviour of \( \kappa \),

\[
C_{\text{actual}} = 4r_0c_0p
\]

and

\[
C_{\text{ideal}} = \frac{p^2(p^2 + 5)r_0c_0}{(p^2 - 1)}
\]

These measures are compared using the composite ratio

\[
R_{\text{composite}} = \frac{C_{\text{actual}}}{C_{\text{ideal}}} = \frac{4(p^2 - 1)}{p(p^2 + 5)}
\]  

(8.15)

The values of these ratios, and the relation between them, is shown in Table 8.2. As can be seen from the table, if the combination of number of links and communications cost as measured by composite cost is used as the basis of comparing the two schemes, the pyramid based on mapping onto a flat array appears to be increasingly attractive as the layer ratio \( p \) increases.

### 8.4.2 A Full Transputer Pyramid

An alternative mapping in which each node in the pyramid corresponds to an individual processor is as follows. Let the image be segmented so that there is a one to one correspondence between segments and processors for each layer of the pyramid. Let the base segments be of size \( p \) by \( p \). The base will be arranged in such a way that each segment can be directly linked to processors in the next layer. This means that the base is either one or two segments wide since each segment must have an external edge in order to provide links external to the segment. Let the subsequent layers be such that \( \rho_j \) processors in the \( j \)th layer become a single processor in the \( (j+1) \)th layer. In particular, \( \rho_0 = p^2 \). Let the top layer with a single processor be layer \( v \). For efficiency is is preferable that there be no uncommitted links, apart possibly from edge links in the base. Then layer \( v - 1 \) has \( \rho_{v-1} \) processors, layer \( v - 2 \) has \( \rho_{v-1}\rho_{v-2} \) processors, and so on. In general, layer \( i \) has

\[
N_i = \prod_{j=i}^{v-1} \rho_j
\]  

(8.16)
processors. The total number of processors in the pyramid is thus

$$T = 1 + \sum_{i=0}^{v-1} \prod_{j=i}^{v-1} \rho_j$$  \hspace{1cm} (8.17)$$

Two examples of such a scheme are shown in Figure 8.4. The first uses $p = 2, \rho_1 = 2, \rho_2 = 2, \text{and } v = 3$, giving $T = 23$. The second uses $p = 3, \rho_1 = 3, \text{and } v = 2$, giving $T = 31$. Of particular interest are those that involve $\rho_1 = 3$, since this matches the number of links in the transputer. In that case $N_i = 3^{v-i}$. As can be seen in Figure 8.4 there is a choice of using two rows of segments as in (a) or a single row as in (b).

For any given set of $\rho_i$, an important question is: what set of links between the base and layer one will minimise the total communications cost? The number of processors accessible in a single base layer segment of size $p^2$ is $2p - 1$ if two sides are available and $3p - 2$ if three sides are available (that is, two opposite sides plus an end). Thus for a single link the number of choices is $2p - 1$ for two sides and $3p - 2$ for three sides. Similarly the number of choices for two links are $(2p - 1)(2p - 2)$ and $(3p - 2)(3p - 3)$ respectively, and for three links are $(2p - 1)(2p - 2)(2p - 3)$ and $(3p - 2)(3p - 3)(3p - 4)$ respectively. The problem of determining the total path length for any one choice is a graph path-length

---

**Figure 8.4:** Two Examples of Mapping onto a Flat Array.

(The base segments are of size 2 and 3 respectively.)
minimisation problem and can easily be solved using Dijkstra's algorithm [86]. If each of
the choices is similarly analysed the minimum can then be determined. Some results of
this analysis are shown in Table 8.3. The shortest-path algorithm for a graph is $O(n^2)$ [86]

<table>
<thead>
<tr>
<th>$p$</th>
<th>Total path lengths</th>
<th>Mean path lengths</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>24</td>
<td>18</td>
</tr>
<tr>
<td>4</td>
<td>56</td>
<td>40</td>
</tr>
<tr>
<td>5</td>
<td>105</td>
<td>75</td>
</tr>
<tr>
<td>6</td>
<td>180</td>
<td>126</td>
</tr>
<tr>
<td>7</td>
<td>280</td>
<td>196</td>
</tr>
<tr>
<td>8</td>
<td>416</td>
<td>288</td>
</tr>
<tr>
<td>9</td>
<td>585</td>
<td>405</td>
</tr>
<tr>
<td>10</td>
<td>800</td>
<td>550</td>
</tr>
<tr>
<td>11</td>
<td>1056</td>
<td>726</td>
</tr>
<tr>
<td>12</td>
<td>1368</td>
<td>936</td>
</tr>
</tbody>
</table>

Table 8.3: Minimum Total and Mean Path Lengths.

and the number of links to be tested is $O(n^l)$ for $l$ links. Thus the overall time cost is
$O(n^{2l})$ which, for $l = 3$, implies a prohibitive computation cost for large $n$. The total
path length for the cases of one and two links can, however, be determined analytically.
This will be done for one case, and the results for the other cases reported without proof.

Consider the case of a single link to a square segment with an odd number of rows and
columns. Set $p = 2w + 1$. Then the minimum total distance will occur when the link
is to a processor in the centre of one side, distances increasing incrementally from this
processor. Thus the distances for the complete segment can be constructed as shown in
Table 8.4. Referring to Table 8.4, the sums of each row are easily determined, yielding
the results shown in Table 8.5. This yields an overall total of

$$\text{Sum} = 6w^3 + 11w^2 + 6w + 1$$

Substituting $p = 2w + 1$, the total for odd $p$ and a single link becomes

$$S(1, \text{odd}) = p(p + 1)(3p - 1)/4$$  \hspace{1cm} (8.18)

A similar technique has been applied to the case of a single link to a segment with even
$p$ and to the case of two links to a segment. The results of these analyses are shown
in (8.19) and (8.20).

$$S(1, \text{even}) = p^2(3p + 2)/4$$  \hspace{1cm} (8.19)

$$S(2) = p^2(p + 1)/2$$  \hspace{1cm} (8.20)

188
Table 8.4: Distances for Odd $p$ and a Single Link.

<table>
<thead>
<tr>
<th>Row</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>$\ldots$</th>
<th>2$w$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$w + 1$</td>
<td>$w + 2$</td>
<td>$w + 3$</td>
<td>$w + 4$</td>
<td>$\ldots$</td>
<td>$3w + 1$</td>
</tr>
<tr>
<td>$\ldots$</td>
<td>$\ldots$</td>
<td>$\ldots$</td>
<td>$\ldots$</td>
<td>$\ldots$</td>
<td>$\ldots$</td>
<td>$\ldots$</td>
</tr>
<tr>
<td>$w - 2$</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>$\ldots$</td>
<td>$2w + 3$</td>
</tr>
<tr>
<td>$w - 1$</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>$\ldots$</td>
<td>$2w + 2$</td>
</tr>
<tr>
<td>$w$</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>$\ldots$</td>
<td>$2w + 1$</td>
</tr>
<tr>
<td>$w + 1$</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>$\ldots$</td>
<td>$2w + 2$</td>
</tr>
<tr>
<td>$w + 2$</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>$\ldots$</td>
<td>$2w + 3$</td>
</tr>
<tr>
<td>$\ldots$</td>
<td>$\ldots$</td>
<td>$\ldots$</td>
<td>$\ldots$</td>
<td>$\ldots$</td>
<td>$\ldots$</td>
<td>$\ldots$</td>
</tr>
<tr>
<td>2$w$</td>
<td>$w + 1$</td>
<td>$w + 2$</td>
<td>$w + 3$</td>
<td>$w + 4$</td>
<td>$\ldots$</td>
<td>$3w + 1$</td>
</tr>
</tbody>
</table>

Table 8.5: Sums of the Row Distances for Odd $p$ and a Single Link.

<table>
<thead>
<tr>
<th>Row</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$(2w + 1)^2$</td>
</tr>
<tr>
<td>$\ldots$</td>
<td>$\ldots$</td>
</tr>
<tr>
<td>$w - 2$</td>
<td>$(2w + 1)(w + 3)$</td>
</tr>
<tr>
<td>$w - 1$</td>
<td>$(2w + 1)(w + 2)$</td>
</tr>
<tr>
<td>$w$</td>
<td>$(2w + 1)(w + 1)$</td>
</tr>
<tr>
<td>$w + 1$</td>
<td>$(2w + 1)(w + 2)$</td>
</tr>
<tr>
<td>$w + 2$</td>
<td>$(2w + 1)(w + 3)$</td>
</tr>
<tr>
<td>$\ldots$</td>
<td>$\ldots$</td>
</tr>
<tr>
<td>2$w$</td>
<td>$(2w + 1)^2$</td>
</tr>
</tbody>
</table>

I have been unable to find expressions for the case of $p = 3$. The values for mean distance are obtained by dividing the total distance by $p^2$, the number of ‘son’ pixels. These means are to be compared with a mean distance of 1.0 for an ‘ideal’ pyramidal processor.

The results derived above will now be compared with those obtained from the scheme of Reeves [233]. Reeves uses the mapping shown in Figure 8.5 and applies a perfect shuffle to this 8 by 8 mesh as illustrated in Figure 8.2 for a 4 by 4 mesh. Refer to nodes in terms of the row and column numbers shown along the side and top respectively. Assuming that nodes 00, 01, 10, and 11 in the base have node 04 as their father, and so on, the total link distances for each of the four fathers in layer 1 can be calculated by reference to Figure 8.2. The results of this are 8, 12, 16, and 20 to father nodes 04, 05, 14, and 15 respectively. The total is thus 56, giving a mean of 14 links per father pixel in layer one. This is to be compared to the scheme described previously in which the worst-case distance is 5 (see Table 8.3).
8.5 Pyramids Applied to the Detection of Surface Blemishes in Kiwifruit

The VIPS\textsuperscript{2} system was used to simulate pyramids in which a kiwifruit image was placed in the base. Dark blemishes on the surface of kiwifruit are preferentially selected by choosing a low rank for the given mapping between the layers. For instance, if layer $j$ is constructed by ranking the 9 pixels in the 3 by 3 window in the $j - 1$ layer corresponding to a given pixel in the $j$ layer, then dark blemishes will be projected up the pyramid. At any layer in the pyramid the resultant image can be extracted and the standard surface blemish defect detection can then be applied\textsuperscript{3}. Some additional noise immunity is obtained by selecting rank 2 rather than rank 1.

An example of the use of this pyramid is shown in Figures 8.7 and 8.8. Figure 8.6 shows the original image, which is of a kiwifruit damaged by leaf roller. Figure 8.7 shows layer one of a pyramid based on a reduction ratio of three and a rank of one. The image has been enlarged to allow easy comparison with the original. Layer two is shown in Figure 8.8, which is one-ninth the size (1/81 as many pixels) of the original in Figure 8.6. The correlation with the original is not so visually obvious, but as is reported below, the linear correlation coefficient between thresholded area in layer two and the corresponding area in the original image is approximately 0.95.

\textsuperscript{2}VAX Image Processing System

\textsuperscript{3}See Chapter 7
The procedure outlined above was applied to 25 kiwifruit images, and the resultant figures for maximum difference and for thresholded blemish area were correlated with the values using the standard algorithm on the full size image. The results are shown in Table 8.6. The improved correlation for maximum pixel intensity using rank two instead of rank one, which is an extreme value of the range, is evident from the table. Some of the loss in correlation for area may have arisen from two particular fruit in the sample which had a fairly diffuse water stain. These were classed as reject fruit by the kiwifruit packhouse but were not rejected by the standard form of the algorithm. However they were both rejected by the first layer of the pyramid, and one was rejected by the second layer. This illustrates the way in which diffuse blemishes are 'concentrated' by the pyramid, which thus represents an enhancement to the standard algorithm.

The technique of adaptive sampling, outlined in Section 8.2, in which a 'busyness' index was computed and averaged over rows and columns of an image, was applied to ten

<table>
<thead>
<tr>
<th>Rank</th>
<th>Maximum Pixel Intensity</th>
<th>Area</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Layer 1</td>
<td>Layer 2</td>
</tr>
<tr>
<td>1</td>
<td>0.856</td>
<td>0.841</td>
</tr>
<tr>
<td>2</td>
<td>0.937</td>
<td>0.891</td>
</tr>
</tbody>
</table>

Table 8.6: Correlation of Pyramid Values Against Standard Algorithm.
kiwifruit images. The ‘busyness’ pyramid algorithm involves the successive mapping from a ‘source’ layer onto a ‘destination’ layer until the full pyramid has been constructed. The steps of this algorithm are as follows:

1. The column differences in the source layer are derived as a measure of busyness.

2. These column differences are summed over all rows to obtain a measure of busyness by columns.

3. A set of column dividers is obtained to segment the image such that the total busyness within all segments is the same (subject to rounding because of digitization).

4. Steps one to three are repeated for row differences, thus yielding a set of row dividers.

5. For each rectangle generated by the intersection of the column and row dividers in the source layer, the enclosed pixels are rank, and the appropriate rank value is passed to the destination layer.

Since this technique yields variable sized rectangles, differing numbers of pixels are available for ranking. If a rank greater than the maximum is specified, the maximum pixel is returned. Using a single reduction factor of five, so that a 100² image is reduced in one step to a 20² image, and applying the standard algorithm to the resultant image, the correlation coefficient for area was 0.930. As implemented in VIPS this reduction imposed a
substantial overhead, and it is not clear how it might be implemented in hardware. However the resultant image was as seen in Figure 8.9, in which the original image showing leaf roller damage (Figure 8.6) is to be compared with an enlarged version of the five-fold reduced image. The blemish is seen to occupy much more of the kiwifruit area, and the kiwifruit itself occupies most of the area that was background in the original. Thus a $20^2$ image based on adaptive sampling will contain much more blemish information than will a $20^2$ image established by a simple rank-based pyramid. The same original image is shown in Figure 8.10 in which layers one and two of a pyramid constructed using a reduction factor of three have been extracted. In Figure 8.11 the difference between a layer one image and its convex hull is shown for a pyramid constructed using a reduction factor of eight. In Figure 8.12 the first layer of a pyramid based on a reduction factor of five has been purposely blurred in order to make recognition of the object visually easier. There is, of course, no additional information in such an image as compared to Figure 8.9, and Figure 8.12 has been produced purely to illustrate the amount of information present in the image.

The nature of a rank one or two pyramid on small, dark blemishes has been studied using the technique illustrated in Figure 8.13. An image containing a connected pattern of dark pixels against a white background is placed in the base (layer zero) of the pyramid. The pyramid proceeds by taking the darkest of each three by three group of pixels to form one pixel in the next layer. The number of dark pixels in layers one and two are counted before and after median filtering. A single dark pixel will propagate through the pyramid
Figure 8.9: Example of Pyramid Construction Using 'Busyness'.

(Pyramid was constructed using a reduction factor of five).

but will be filtered out when a median filter is applied. The minimum sized group of pixels in the input that produces at least one dark pixel in the output of layer one after median filtering is a 3 by 5 block of pixels. However, a block of 4 by 4 dark pixels will be filtered out. For layer two the first non-white filtered output requires a block of 8 by 17 black pixels in the original image, and a block of 16 by 16 dark pixels will be filtered out by a layer two median filter.

Thus realistic use of pyramidal vision for the detection of surface blemishes will involve a median filter for the original image, but no further filtering. If noise proves to be a problem, the experience with kiwifruit images suggests that a rank two pyramid will give results comparable to a rank one pyramid. The actual behaviour of small groups of dark pixels depends on their alignment in the three by three mesh. An example of a group of three dark pixels is illustrated in Figure 8.14. In (a) the group lies completely within one three-by-three block, giving an output of just one pixel. In (b) the group straddles the boundary between blocks, with the result that the output is three dark pixels. If the original image is 100 by 100 pixels then the original blemish is 0.03% of the total number of pixels, the output in Figure 8.14(a) is 0.09%, and the output in (b) is 0.27%. For larger blocks of dark pixels the relative increase in area is much less, tending to unity as the size of the block tends towards the size of the full image.
8.6 Hardware Implementations

A realistic use of pyramidal vision for surface blemish detection in kiwifruit is likely to involve a hardware implementation of a two- or three-layered pyramid, with the resultant image being examined by one or more processors. If lighter blemishes such as greedy scale are to be detected a second pyramid would need to be applied in which are high rank (eight or nine for a three by three reduction) would be used.

The following is a suggestion for a full pyramid image-processing system. A rank filter has been implemented as a VLSI chip by Naylor [205]. Chaplin [65] has indicated that a rank filter can be constructed from a field programmable gate array. In either case a minor modification would involve rejecting eight out of every nine pixels in the output, and by pipelining such devices a pyramid of any specified number of layers could be established at video rates. Suppose the input image from the CCD or video camera is digitized at the rate of $512^2$ every 40 milliseconds – that is, 6.5536 MHz. The digitized output
is first passed to a median filter before being passed to the first pyramid constructor. Then layer one of size $170^2$ will be constructed at a rate of 728.2 KHz, and so on. The characteristics for a number of layers are shown in Table 8.7. The data volume of layer two

<table>
<thead>
<tr>
<th>Layer</th>
<th>Size</th>
<th>Data volume</th>
<th>Data rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>512</td>
<td>262144</td>
<td>6.5536 MHz</td>
</tr>
<tr>
<td>1</td>
<td>170</td>
<td>28900</td>
<td>728.2 KHz</td>
</tr>
<tr>
<td>2</td>
<td>56</td>
<td>3136</td>
<td>80.91 KHz</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>324</td>
<td>8.99 KHz</td>
</tr>
</tbody>
</table>

Table 8.7: Characteristics of Pyramid Layers.

is approximately one third of that handled by the array of twenty transputers described in Chapter 4. I would anticipate that such an image could be handled by ten transputers within a quarter of a second. Alternatively if the image at layer three is considered to contain sufficient information for the purpose of defect detection, it seems likely that a single microprocessor having processing comparable to that of the transputer would be able to handle the required image-processing. The output of the final pyramid layer should be into either double-ported memory or into a bank of switched-bank memory so that the microprocessor can proceed with one image while the other is being loaded.
As mentioned above if light-coloured blemishes are also to be detected a second pyramid would need to be constructed, together with its processor. Unless, as outlined in the previous chapter, a linescan camera is used, a full view of a kiwifruit would require three images, with one pyramid being devoted to each image. With three field-programmable logic devices, a block of memory, a microprocessor, and 'glue' chips devoted to each pyramid, the complexity of a system incorporating three or six pyramids is not great.

8.7 Conclusions

The introduction to this thesis outlined the major objective of this research, namely the development of techniques for the real-time inspection and sorting of kiwifruit. One major approach, involving large processing power, was dealt with in preceding chapters4. The alternative approach, based on data compression, has been explored in this chapter. Pyramids, involving images at successively lower resolution, are particularly applicable in the case of surface blemishes in kiwifruit, since these blemishes are displayed as an intensity level that is different to that of the surrounding kiwifruit surface. This means

4In particular, Chapter 4.
that if nonlinear constructors such as a ranking scheme are used to construct successive layers of the pyramid, blemishes can be emphasized relative to the unblemished fruit. Thus although a blemish may occupy as little as a single pixel in the original image, that blemish will be present in every layer of the pyramid. This is a significant advantage with respect to surface blemish detection compared to pyramids based on linear constructors such as means.

Investigations were made into the feasibility and value of patenting this process, since it would be straightforward to construct it using VLSI chips. However the limited protection offered by patenting did not appear to warrant the considerable cost and the matter was not pursued. At current hardware costs a blemish detection system using pyramidal vision could be constructed for considerably less than a multi-processor system able to perform the algorithm within the same time span. However, it is important not to prejudge future hardware developments, since new generations of micro-processors, based on, for instance, gallium arsenide [104], or further enhancements in signal processing chips, could make the implementation of a full surface blemish detection algorithm in software
a relatively attractive operation.

There are a number of options in the construction of pyramidal processors:

1. The processing elements can be specially made for the pyramid, or can be off-the-shelf processors.

2. The processing elements can be discrete, or can be overlapping, so that one processor performs the work of two or more pyramidal processing elements.

3. The choice of the inter-layer ratio $p$ determines many important characteristics of the pyramid, including the number of layers and number of elements in the pyramid, and the communications costs, especially if the pyramid has been mapped onto a flat array.

Specially-made processing elements will probably have $p^2 + 5$ links per processor. If transputer-type serial links are used, the construction of a processor with the requisite number of links would represent only an incremental change in VLSI technology. In regard to option number two, the ultimate form of an overlapping pyramidal processor is one in which all processing elements are resident on a single processor, and this is the way in which the pyramidal processing techniques presented in this chapter have been developed.
Several possibilities exist for the construction of pyramidal processors from transputers. Analysis of two such schemes has shown that pyramids based on transputers can have mean path lengths less than twice that of the equivalent 'ideal' pyramid based on custom-made processing elements, while having a significantly simpler hardware design.

Pyramidal processing techniques based on approaches such as measures of 'busyness' offer opportunities for adaptive pyramid construction that can help avoid the wastage of processing power resulting from the analysis of 'uninteresting' image regions such as blank backgrounds. A circular 'kiwifruit' in a square image area arranged so that there is a half-radius separation of the circle from the edges will occupy just 35% of the total area. However operations such as rank filtering will be applied to the 65% of the area that will not be contributing to the final image analysis. Most current adaptive pyramid algorithms are fairly computation-intensive, and the development of efficient algorithms, perhaps ones that can be implemented in hardware, is an exciting field for future research.
Chapter 9

Discussion and Conclusions

9.1 Introduction

As stated in the introduction, this thesis deals with the application of machine vision to the quality control inspection and grading of kiwifruit. In the introduction the broad objective of the thesis was stated to be:

To consider a system which will detect surface blemishes and shape defects in kiwifruit with an accuracy of at least 75% and at a rate of four fruit per second.

Machine vision is subject to the twin constraints of large data volumes and restricted processing time. Two broad avenues were pursued in attempting to overcome these constraints:

- High processing power can be achieved by means of multiprocessing, such as with networks of transputers.
- The reduction in the volume of data can be achieved by means of pyramidal vision.

The remainder of this section is organised as follows. In Section 9.2 the work presented in this thesis is reviewed, and original contributions are highlighted. Suggestions for future research and development, perhaps leading to practical machine vision systems, are presented in Section 9.3.

9.2 Review of This Thesis

In this section each chapter will be reviewed, and contributions that are believed to be original are outlined.
9.2.1 Overview

The work on multiprocessor networks in general, and transputer networks in particular, is a natural progression from the work of previous contributors working under Dr Bob Hodgson at the University of Canterbury. A principal target of this research was the application of machine vision to kiwifruit sorting and grading. In particular, McNeill [188] investigated the way in which a system based on a single processor could be applied to tasks such as surface blemish detection in kiwifruit. This work was extended by MacKenzie [180] who studied the characteristics of a multi-processor system based on conventional processors. The multi-processor work presented in this thesis further extends this work by distributing images across a network of processors, dispensing with a global bus.

The work on data reduction through pyramidal vision is an alternative approach, and is presented as a response to the time constraints of machine vision. The studies of blemish detection algorithms are based on the work of Bailey [19] and McNeill [188], while the lighting research was carried out because this appeared to be a neglected area.

9.2.2 Chapter 2 : Multiprocessor Networks

The literature on a wide variety of multiprocessor systems is reviewed, with particular reference to the processing requirements for real-time image-processing. One alternative, based on the idea of data flowing through a processing system, is conceptually attractive but has proven difficult to implement. A more promising approach is based on the use of multiple processors. An extension of traditional bus-based computers provides easy access to shared variables but suffers from bus and memory contention problems when the number of processing units on a bus becomes large. The alternative of network-based multi-processors takes two forms — those in which inter-processor links can be switched and those in which the links are fixed. There is thus a tradeoff between shorter access paths, and greater hardware and software complexity.

A major problem in multiprocessor networks is that of deadlock, and a great deal of research has gone into deadlock detection, prevention, and avoidance. The easiest but least efficient way of dealing with deadlock is to allocate sufficient resources in terms of buffers and so on to ensure that within specified constraints deadlock should never occur. A more efficient approach is to use deadlock avoidance in which processes do not proceed until all resources are available. The most potentially efficient is deadlock detection and resolution, but this problem has resulted in a large number of incorrect algorithms.

The work of this chapter is used as the basis of Chapters 3 and 4.
Symmetric chordal networks of degree four offer a practical means of interconnecting processors having four inter-processor links. The symmetry of the networks makes the mapping of algorithms onto the network a comparatively simple task. In particular, for image processing applications, an image will map very simply and conveniently onto the network. In this chapter techniques have been given for determining the diameter and mean interprocessor distance for a given network. It has been shown that if a network is chosen to have a minimum diameter (by appropriate choice of the number of nodes and/or the chord length) then the mean internode distance will also be a minimum. A particular subset of these networks is optimal in the sense of having the greatest possible number of processors for a given diameter. In optimal networks the diameter and the mean are both of $O(\sqrt{n})$.

Although optimal networks are not incrementally extensible, any number of processors can be added to a general chordal ring network. If the number of additional processors is large the optimum chord length will probably change. Single faults result in a comparatively small degradation in performance and a minimum of three faults can be tolerated in the network before any nodes become isolated. An algorithm has been presented to provide interprocessor communication in an optimal network. Routing in non-optimal and broken networks might best be achieved by means of look-up tables. The theoretical performance of chordal rings has been verified by measurements on an actual chordal ring network based on transputers. A degree four chordal ring network is used as the basis of the multitransputer image processing system reported in Chapter 4.

Chordal rings of degree three have been analysed by other authors, but the material presented in this Chapter is believed to be original. The method used in this Chapter could be applied to any static interconnection network, and as such is believed to be an original contribution.

9.2.4 Chapter 4: Multitransputer Image Processing

An image-processing facility has been described. This is based on a network of transputers configured as a chordal ring. The software is based on a link shell which makes the details of the network configuration invisible to the applications programmer and to the system user. The main features of this shell are as follows.

- A set of link buffers is provided to allow deadlock-free message passing.
- The imposition of a specific message format provides consistency checks.
- A message log is provided as an aid to software debugging.
A range of image-processing operations have been implemented, providing experience in the writing of distributed software. These make use of a specially-written utility that allows the PC character attributes and graphics screens to be controlled by the host transputer from within the transputer development system.

The performance of individual sections of Occam code have been analysed and some of these have been used to improve the efficiency of the image processing system. Some measurements of the performance of the image processing system on the network of transputers have been obtained. These measurements have been used to estimate the performance of a transputer network devoted to machine vision, suggesting that the objective of running the kiwifruit surface blemish detection algorithm in less than a quarter of a second could be achieved on a network of ten transputers if the rank filter was implemented in hardware, and on a network of twenty transputers if the rank filter was implemented in software.

At the time of its development the link shell was believed to be a novel approach to dealing with multi-transputer networks, although similar approaches have subsequently been described in the literature. The image processing system encompasses a number of developments:

1. A facility has been provided to allow the IBM PC graphics and text attributes to be accessed from within the Transputer Development System.

2. An algorithm for moving an image in a multi-processor environment, based on the movement of three rectangles per processor, has been developed.

3. An algorithm for rotating an image in a multi-processor environment, based on combining row and column dividers to produce a network of square image segments which are individually moved and rotated, has been developed.

4. An algorithm for a row-based distributed convex hull in which a 'central' processor in each row is specified as the hull former and distributor, has been developed.

9.2.5 Chapter 5: Mapping Image-Processing Operations Onto Multitransputer Networks

The efficient mapping of problems onto a multi-processor architecture has received widespread attention, and although the general problem is NP-complete, various methods exist for approximating the solution. Image-processing operations can be classified as point, neighbourhood, medium grain, large grain, and global. In this Chapter, methods have been developed for calculating the speedup and efficiency of such operations executed on a network, where the equivalent performance on a single processor is taken as the benchmark. When these operations are analysed with respect to the transputer network described in Chapter 4, the operations display a positive speedup in all likely cases.
The methodology and results presented in this Chapter are believed to be original. Although a specific network, namely a chordal ring transputer network, was used as the basis of the analysis, the results are of much more general application.

9.2.6 Chapter 6 : Lighting for the Optical Inspection of Kiwifruit

The effects of the illumination of kiwifruit from two lights symmetrically placed about the camera direction have been explored. As the angle between the camera and the lights is increased surface projections such as Hayward hooks become more readily detectable. At the same time a dip appears in the intensity profile and this may need to be compensated for in order to avoid the apparent detection of a false surface blemish. A compromise between the conflicting demands for such a system would result in placing the lights at approximately 45º on either side of the camera direction. This is a significantly larger angle than is used in current commercial optical graders in kiwifruit packhouses in New Zealand.

The topic of lighting has been comparatively neglected in the literature on machine vision, and the analysis presented in this Chapter is believed to be original. In particular, the optimum placement of lights for the detection of surface projections on objects such as kiwifruit has not, to the best of my knowledge, been previously reported.

9.2.7 Chapter 7 : Algorithms for Defect Detection in Kiwifruit

Work carried out by Bailey and McNeill in the field of area blemish detection and shape defect detection in kiwifruit has established algorithms that are reasonably reliable when measured against the objectives of the application of machine vision to kiwifruit packhouses. Furthermore, these algorithms are relatively insensitive to noise, which is likely to be an important factor when this technology is moved from the laboratories to the factories. Attempts to find alternative techniques were unsuccessful. Whenever a new algorithm started showing promise, images were found on which it was unsuccessful in its simplest state. This lead to various enhancements, each exacting a time penalty, to cater for the variations in the kiwifruit images. However it is to be noted that the basic algorithms do not always perform well, and results reported in Chapter 8 provide some enhancement to the basic surface blemish detection algorithm. The use of linescan images provides a considerable simplification to the problem of capturing images for a full kiwifruit.

The original contribution of this Chapter consists of the provision of several algorithms which were investigated and subsequently rejected when compared to the previously-developed algorithms, and to the analysis of linescan images of rotating objects.
9.2.8 Chapter 8: Pyramidal Vision Applied to Automatic Inspection

The alternative to the use of large processing power for machine vision, based on data compression, has been explored in this Chapter. Pyramids, involving images at successively lower resolution, are particularly applicable in the case of surface blemishes in kiwifruit, since these blemishes are displayed as an intensity level that is different to that of the surrounding kiwifruit surface. This means that if nonlinear constructors such as a ranking scheme are used to construct successive layers of the pyramid, blemishes can be emphasized relative to the unblemished fruit. Thus although a blemish may occupy as little as a single pixel in the original image, that blemish will be present in every layer of the pyramid. This is a significant advantage with respect to surface blemish detection compared to pyramids based on linear constructors such as means.

Several possibilities exist for the construction of pyramidal processors from transputers. Analysis of two such schemes has shown that pyramids based on transputers can have mean inter-processor path lengths less than twice that of the equivalent 'ideal' pyramid based on custom-made processing elements, while having a significantly simpler hardware design. Pyramidal processing techniques based on techniques such as measures of 'busyness' offer opportunities for adaptive pyramid construction that can help avoid the wastage of processing power resulting from the analysis of 'uninteresting' image regions such as blank backgrounds.

The use of pyramids using pixels of low rank as a means of 'growing' regions that were dark relative to their surroundings is believed to be a novel approach to the use of pyramidal vision for surface blemish detection.

9.3 Suggestions for Future Research and Development

In this section possible avenues for future research are presented. Many of these suggestions have the potential to lead to developments that could be prototyped in industrial locations such as kiwifruit packhouses. The remainder of this section is organised as follows. Potential developments in transputer networks are reviewed in Section 9.3.1. Pyramidal vision offers a potential for data reduction that is very attractive for machine vision, and possible developments are reviewed in Section 9.3.2. Total systems, possibly based on a combination of transputers and pyramidal vision, are reviewed in Section 9.3.3.

9.3.1 Transputer Networks

In this section potential enhancements to the software of the Transputer Image Processing System are presented, a proposal is made for a 'meta-Occam', and suggestions are made for alternative hardware systems that are oriented to image processing.
Software Development of TIPS

As detailed in Chapter 4, the use of channel buffers for the link shell follows the technique recommended by INMOS. However, an alternative method for using channel buffers was reported in Chapter 4 and, although it involved some added complexity, significant improvements in performance are anticipated. Some aspects of the link shell were developed to provide a ‘friendly’ environment, and an embedded system would not require these facilities. An more accurate assessment of the performance of a transputer network applied to surface blemish detection would be obtained if software, once fully developed using a system such as TIPS, was rewritten without the link shell. The use of deadlock detection and resolution might provide a significant performance improvement.

The software for image distribution and gathering operates in a manner that offers a compromise between efficiency and simplicity, and there are likely to be considerable opportunities to provide better performance.

The provision of a TIPS program facility, in which a sequence of TIPS commands can be loaded on the system before running, would allow more accurate performance measurements as well as providing a convenient facility when repetitive sequences of commands are involved.

General Software Development

The majority of low-level image processing tasks in a multi-processing environment employ the network processors in a symmetric fashion. Such tasks consist of repetitions of the pattern illustrated in Figure 9.1 in which \( P_1, P_2, \ldots \) represent Occam processes, and \( C_{ij} \) represents a channel communication linking process \( i \) with process \( j \). This can be thought of as a single flow of code, although the writing of separate code for host and network processors tends to obscure the overall form. A technique that would permit Occam code to be written in this form would be based on software that generated the appropriate Occam code from an extension of Occam that might be labelled ‘meta-Occam’. An example of such code is shown in Figure 9.2. The objective of the Occam generator would be to produce two blocks of code. The code for the host can be produced quite simply in the case of the meta-Occam code given in Figure 9.2. However the code for the network must contain

- The full configuration, based on the single statement at the beginning of Figure 9.2.
- The network code given in Figure 9.2.
- A suitable harness such as the link shell that provides for the routing of messages throughout the network, and provides protection against deadlock.
A special feature of the code in Figure 9.2 is that no distinction can be made between one network processor and another. This simplification should make such a program generator a practical objective. A similar approach has been adopted by Morrow et al [198] who have developed a Pascal-based high-level language called *Latin* that allows the software for a network to be expressed in terms of a single flow of code. Their system is of more general application, but this seems to be at the cost of increased complexity.

**Program Generator**

The following scheme for a program generator seems to be a practical objective. Although initially thought of in terms of low-level image processing operations, there appears to be no reason why it could not be extended to more complex tasks such as pattern recognition. The scheme would operate at two levels: a systems level and a user level. At the systems level a library of image processing operations would be established, the code being verified
configuration=chordal.ring(20,4) -- 20 transputers, chord = 4

... declarations

{host

... host PROCs

SEQ

write(" x = ")

read.integer(x)

%broadcast; 2; hn; x; end.of.message -- % = communicate

{network

-- this begins network software

SEQ

x2 := x*x

%host; 3; nh; processor; x2; end.of.message

-- this generates x2[0], x2[1], etc. on the host

}network

-- this ends the network software

sum := 0

SEQ i = 0 FOR full.network

sum := sum+x2[i]

write(" Sum = ")

print.integer(sum)

-- this ends the host software

Figure 9.2: An Example of a Meta-Occam Program.

through a system such as TIPS. The performance of these operations would be measured: for instance, the convex hull algorithm would be specified in terms of the execution time on a network of a specified size and topology, on a single column of transputers, and on a single transputer, all for various image sizes. In addition, the time taken to distribute images of various sizes over the network, to collect those images into a single column of transputers or a single transputer, or to redistribute those images over some other network configuration would be measured.

A user of the program generator would enter basic information such as the sequence of operations to be followed, in much the same way as a VIPS program is specified [19]. The program generator then determines the optimum placement of images to minimise the total execution time. For instance, some operations may execute almost as quickly on a single transputer as on a full network, and the added time cost of distributing and
gathering the image may make it more efficient to use a single transputer.

As a general expression of this problem, define a set of *locations*, such as 'full network' and 'host', and a set of *operations*. Let the time for operation $j$ at location $i$ be $t(i, j)$ and let the communications cost, measured as elapsed time, for image transfer between locations $l$ and $m$ be $c(l, m)$. Let $s$ be the start location for the program, with $f$ being the finish location. Then the total time for $n$ operations is given by

$$ T = \sum_{i=1}^{n} t(m_i, i) + c(s, m_1) + \sum_{i=1}^{n-1} c(m_i, m_{i+1}) + c(m_n, f) $$

(9.1)

where the $m_i, i \in [0, n]$, refer to any valid location. The problem of determining the optimum placement of images then consists of choosing the $m_i$ such that $T$ is minimised. Expressed in graph-theoretic terms, this problem is a minimum-path problem for a weighted graph.

Having determined the optimum placement, the program generator then produces appropriate host and network code, with the communications required by (9.1). The user is not concerned with the placement, and ideally would see the system as a monolithic processor.

**Special Hardware**

Referring to Figure 9.3, special hardware constructed using double-ported memory [206] in which one port of each memory block might be connected to a common bus fed by the image capture system, while the other ports were connected to local transputer buses, would allow image storage and image processing to occur in parallel. It seems likely that such boards will soon become commercially available since image processing is a major application of transputers.

Another development, which is already commercially available, is the use of dynamic network routing based on chips such as the INMOS C004 which provides a 32-way crossbar link switch.

The transputer is an evolving family of micro-processors, and subsequent generations may provide facilities for *wormhole* routing [17], which offers the potential for extremely fast message transfer. In this scheme, messages pass directly through intermediate processors in a network without needing to interact with any software, so that the delay involved in message passing becomes independent of the size of the network. Transputer link speeds are likely to increase from the present 10 or 20Mbits/second. These developments may allow images to be loaded directly onto a transputer network at video rates, thus overcoming many of the drawbacks of existing transputer-based boards. Transputers may be made in gallium arsenide [287], with a consequent increase in processing power of possibly an order of magnitude.

210
9.3.2 Pyramidal Vision

A suggestion offered in Chapter 8 involved the use of field-programmable logic devices plus support chips to construct a pyramid for machine vision. In this scheme each layer was produced from the preceding by a modification of the rank filter. More complex logic devices would allow more complex inter-layer mapping functions to be used. In particular, hardware-based adaptive pyramids may become feasible, and since more information relevant to the application would be preserved from one layer to the next the final layer could be made smaller. It is not at all obvious how an adaptive pyramid might be implemented in hardware, and this is an interesting challenge for future research.
9.3.3 Prototype Systems

This section contains a suggestion for a prototype kiwifruit blemish detection system. It encompasses components from all the work reported in this thesis. Referring to Figure 9.4, a stream of kiwifruit pass along a conveyor and are aligned and rotated by a system of rollers. The kiwifruit, suitably illuminated, pass under a linescan camera, thus providing a single image of the full surface, with any surface projections such as Hayward hooks being highlighted. The video stream from the linescan camera is divided, one part going to simple edge extraction hardware which outputs an edge description for subsequent analysis by a processor. The other part of the video stream is passed via an analog-to-digital converter to the base of a pyramid that has been implemented in hardware. Images are extracted from one or more layers in the pyramid and passed to a multiprocessor network which performs the final steps of surface blemish detection. The assessments from the processors on the two parts of the video stream are combined in an accept/reject decision for each kiwifruit. The hardware is available to make such a prototype system a reality.
9.4 Epilogue

It is somewhat unconventional to wait until middle age before commencing work towards a PhD. Accordingly I conclude this work with a few general observations.

Although the thought of returning to university had risen from time to time over a number of years, it was not until 1984 that I decided to pursue such an objective more positively. I examined the latest edition of the university calendar, noting from the lists of publications who was working in what field. In terms of my particular interests one name stood out: Dr R M Hodgson. Accordingly I discussed my plans with Bob, and he was good enough to accept me as a student. He did, however, issue a warning. I would, he told me, be amongst many bright young graduates, and in particular my command of mathematics might be sorely tested. Throughout the subsequent years I never forgot this warning, and am pleased to have survived the experience more or less intact. Certainly my twenty year old physics degree was frequently found lacking, and knowledge that had been gathering dust over the years needed cleaning and polishing, but in the end these deficiencies were more than compensated for by the rich experiences that I had gathered over the intervening years in the 'real world'.

It seems to me that not only have these broadened horizons enabled me to bring more to my research, but that I have also derived more from the experience as a consequence.
Bibliography


217


218


219


221


224


225


231


Acknowledgements

My supervisor, Professor Bob Hodgson, has devoted much time and energy to ensuring that I did not deviate too much in this research programme, and to ensuring that my work was of as high a standard as possible. To Bob I give my sincerest thanks.

Following Bob’s appointment to Massey University two-thirds of the way through my PhD, Bill Kennedy and Phil Bones have spent some considerable time with me and have done some much valued proof reading for the thesis. I am very grateful for their support.

Other members of the Electrical and Electronic Engineering Department staff at Canterbury University have provided information, encouragement, and support. In particular Richard Bates has provided frequent words of encouragement. My fellow post-graduate students have always been ready and willing to help when asked. The assistance of all these people is very much appreciated.

Other departments in the university have provided valued assistance from time to time. The Audio-Visual Centre has provided a prompt and friendly service on numerous occasions. The Engineering Library has provided excellent support, and in particular the assistance of Adrianna de Groot is gratefully acknowledged.

The financial assistance of the New Zealand Kiwifruit Authority, through the Roly Earp Scholarship, eased the considerable financial burden involved in spending three years on this research, and the support of the NZKA is deeply appreciated.

The transputer equipment described in this thesis was purchased with grants from the University Grants Committee, and further financial support has come from the NZKA. I am grateful for the support of these organisations.

Without Wendy’s support it would not have been possible for me to undertake this PhD programme. At times the long hours spent at university and the financial stringency seemed to be taking its toll, but Wendy ensured that the achievement of the final goal was never in doubt. To Wendy goes my deepest thanks.
Appendix A

Set Notation

A set is a collection of items [185]. For example, if $A$ consists of $a_0, a_1, \cdots$, then write $A = \{a_0, a_1, \cdots\}$. Given two sets $X$ and $Y$, the union $X \cup Y$ is the set containing all elements that are in either $X$ or $Y$ or both. The intersection $X \cap Y$ is the set containing elements that are in both $X$ and $Y$. If $X$ is a set whose elements are all members of the set $Y$ then $X$ is a subset of $Y$, and this is written $X \subset Y$. In particular:

\[
X \subset X \\
X \subset Y \quad \text{and} \quad Y \subset X \quad \Rightarrow \quad X = Y \\
X \subset Y \quad \text{and} \quad Y \subset Z \quad \Rightarrow \quad X \subset Z
\]

Such a relation is called a partial ordering.

Two special sets are the null set $\emptyset$ and the universal set $U$. The complement $X'$ of the set $X$ is the set whose elements are in the universal set but not in $X$.

The following properties can be listed:

\[
X \cup Y = Y \cup X \\
X \cap Y = Y \cap X \\
X \cap (Y \cup Z) = (X \cap Y) \cup (X \cap Z) \\
X \cup (Y \cap Z) = (X \cup Y) \cap (X \cup Z) \\
\emptyset \cup X = X \\
U \cap X = X \\
X \cup X' = U \\
X \cap X' = \emptyset
\]

Set algebra is formally identical to Boolean algebra, with $\cap \equiv \cdot$ and $\cup \equiv \dag$. 

236
Appendix B

Graph Theory

B.0.1 Basic Definitions

A graph is defined as $G = (V, E)$ where the set $V = \{v_1, v_2, \ldots \}$ is called the set of vertices and the set $E = \{e_1, e_2, \ldots \}$ is called the set of edges, such that each edge $e_k$ is associated with an unordered pair $(v_i, v_j)$ of vertices. Vertices are sometimes called nodes and edges are sometimes called arcs. If an edge is associated with a vertex pair $(v_k, v_k)$ then the edge is called a loop. Edges may have associated with them a weight, and weighted graphs are of significance in distributing tasks over a multi-processor network [38], but in the remainder of this appendix only unweighted graphs will be considered. If more than one edge is associated with one pair of vertices than such edges are parallel edges. A graph without loops or parallel edges is a simple graph. The degree of a vertex is the number of edges incident at that vertex. A graph in which all vertices are of equal degree is a regular graph. A complete graph has an edge between every pair of vertices. A pendant vertex is a vertex of degree one. A graph $G$ is planar if there exists some geometric representation of $G$ on a plane such that no two edges intersect (vertices are excluded).

B.0.2 Isomorphism

Two graphs $G$ and $G'$ are isomorphic if there is a one-to-one correspondence between their vertices and between their edges such that the relationship between edges and vertices is preserved. The determination of whether two graphs are isomorphic is not necessarily a simple task, and it is generally easier to show that two graphs are not isomorphic by finding whether they have the same number of vertices, the same number of edges, the frequency of the degrees of the vertices are the same, and so on. Since the checking of all $n$ vertices of one graph against another graph through reordering of the vertices can take up to $n!$ reorderings, much attention has been given to finding efficient algorithms [86].
B.0.3 Subgraphs

A subgraph $g$ of a graph $G$ is a graph whose vertices and edges are all in $G$ and whose edges have the same end vertices in $g$ as in $G$. Edge-disjoint subgraphs do not have any edges in common, while vertex disjoint subgraphs do not have any vertices in common.

B.0.4 Walks, Paths and Circuits

A walk is an alternating sequence of vertices and associated edges such that no edge is traversed more than once. A path is a walk in which no vertex including the start and finish vertices occur more than once. A circuit is a walk in which no vertex appears more than once apart from the start and finish, which are the same. A graph is connected if there is at least one path between every pair of vertices. A Hamiltonian circuit is a circuit that traverses every vertex of the graph once. A complete graph with $n$ vertices, $n$ an odd number greater than three, has $(n - 1)/2$ edge-disjoint Hamiltonian circuits. A graph is minimally-connected if the removal of any one edge disconnects the graph.

B.0.5 Trees

A tree is a connected graph without any circuits. There is only one path between every pair of vertices in a tree, and conversely a graph having only one path between every pair of vertices is a tree. A tree with $n$ vertices has $n - 1$ edges. A graph is a tree if and only if it is minimally connected. A rooted tree is a tree in which one vertex, called the root, is distinguished from all other vertices. A binary tree is a tree in which one vertex is of degree two and all other vertices are of degree three. Binary trees are rooted trees, have an odd number of vertices $n$, and have $(n + 1)/2$ pendant vertices. A spanning tree of a connected graph $G$ is a subgraph of $G$ containing all the vertices of $G$. A spanning tree is also called a maximal tree subgraph of $G$. A collection of trees is called a forest, and a disconnected graph has a spanning forest.

B.0.6 Cut-Sets

A cut-set of a connected graph $G$ is a set of edges whose removal from $G$ leaves $G$ disconnected. A cut-set $C$ is a minimal cut-set if no subset of $C$ other than $C$ itself is a cut-set of $G$. Every edge of a tree is a cut-set. The number of edges in the smallest cut-set of a connected graph $G$ is the edge connectivity of $G$. Vertex connectivity is analogously defined. A connected graph is separable if its edge connectivity is one.
Directed Graphs

A directed graph or digraph is a graph in which each edge $e_k$ is associated with an ordered pair $(v_i, v_j)$ of vertices. An adjacency matrix of a digraph is a matrix $X = [x_{ij}]$ such that $x_{ij} = 1$ if there is an edge directed from the $i^{th}$ vertex to the $j^{th}$ vertex, and zero otherwise. A digraph having no directed circuit is called acyclic. A digraph is acyclic if and only if its vertices can be ordered such that the adjacency matrix $X$ is a triangular matrix. Given a digraph $G$ having one or more directed circuits, the smallest set of edges whose removal makes $G$ acyclic is the minimum-feedback arc set. One algorithm [86] for minimal decyclization of a digraph makes the adjacency matrix as near triangular as possible, and then uses the remaining non-triangular elements as the basis of a minimum-feedback arc set.
Appendix C

Statistics

C.0.8 Test for Linearity of Regression

The following technique is based on that described by Wine [294].

\[ y \]

\[ x \]

Figure C.1: Groups of Points for Regression Analysis.

(Each vertical line segment represents a group of points with fixed \( x \) and variable \( y \)).

Given a set of \( k \) groups of data, with \( n_i \) points in the \( i^{th} \) group, as illustrated in Figure C.1, it is desired to determine whether linear regression line can be applied to the points. The total number of points is given by

\[ n = \sum_i n_i \]

Let the data be \( (x_i, y_{ij}) \) where \( i \) corresponds to groups, and \( j \) corresponds to points within groups. That is, the points in a given group have fixed \( x \) and variable \( y \).

The mean within a group is

\[ \bar{y}_i = (\sum_j y_{ij})/n_i \]
The overall means are given by

\[
\bar{y} = \frac{\left(\sum_i \sum_j y_{ij}\right)}{n}
\]
\[
\bar{x} = \frac{\left(\sum_i n_i x_i\right)}{n}
\]

Define the following quantities (using Wine's terminology):

1. Sum of products

\[SP = \sum_i [(x_i - \bar{x}) \sum_j y_{ij}]\]

2. Sum of squares of \(x\)

\[SSX = \sum_i [n_i (x_i - \bar{x})^2]\]

3. Sum of squares among group means

\[SSTr = \sum_i [n_i (\bar{y_i} - \bar{y})^2]\]

4. Sum of squares within groups

\[SSW = \sum_i \sum_j (y_{ij} - \bar{y_i})^2\]

The sum of squares of deviates of group means about the estimated regression group mean, \(SSD\), is

\[SSD = SSTr - Sp^2 / SSX\]

Since two variables, and hence two variances, are involved, it is appropriate to use the F (or Fisher) distribution. Wine shows that

\[F = \frac{SSD/(k - 2)}{SSW/(n - k)}\]

Tables can then be used to test the hypothesis of linearity, that is

\[H_0 : y_{ij} = a + bx_i \quad \forall i, j\]

for a specified confidence interval. The relevant table entry is \(F(k - 2, n - k)\).
Appendix D

Occam and the Transputer

D.0.9 Folding Editor

The program listings in this thesis are as seen using INMOS’s folding editor [151]. Analogous to a fold in a sheet of paper, the folding editor produces text that when folded is visible only as a single line of three dots plus some text identification. The enclosed text can be made visible either by unfolding so that the text becomes continuous with the enclosing text, or by entering the fold. The program listings in this thesis employ the latter technique. A fold may be filed, in which case the three dots will be followed by an ‘F’.

As an example, consider

SEQ

... initialise
WHILE continuing
    interrogate (continuing, count)
:

This can be listed either by unfolding the fold, with the edges being visible as 'crease' lines, marked by three curly brackets:

SEQ

{{{ initialise
    continuing := TRUE
    count := 0
    write ("Proceeding")
}}}
WHILE continuing
interrogate (continuing, count)

or as the contents of the fold by itself:

continuing := TRUE
count := 0
write ("Proceeding")

The folding editor allows the structuring of programs over and above that imposed by the use of procedures.

D.0.10 Occam

Occam deals with processes which communicate with each other via channels [152]. A process is a finite collection of actions, ultimately based on assignment, input, and output. Processes can operate completely independently of each other, and hence asynchronously. The scope of processes is determined by indentation. Communication between processes occurs when one process is ready to output a value onto a channel and another process is ready to input from that same channel. Thus channels provide synchronization on asynchronous processes.

Processes

Assignment is achieved by means of

variable := expression

where the expression is constructed from constants, variables, and operators. Channel input is achieved by means of

channel ? variable

while channel output is achieved using

channel ! expression
Constructs

A combination of processes forms a *construct*. A *sequential* construct is of the form

\[
\text{SEQ} \quad \text{process1} \quad \text{process2} \quad \ldots
\]

with the component processes, which may themselves be constructs, being executed in the given order. A *parallel* construct is in the form

\[
\text{PAR} \quad \text{process1} \quad \text{process2} \quad \ldots
\]

with the component processes being executed concurrently. The merging of processes is achieved by the *alternative* construct

\[
\text{ALT} \quad \text{input1} \quad \text{process1} \quad \text{input2} \quad \text{process2} \quad \ldots
\]

in which the first input to become available determines which process is executed. The *conditional* construct is in the form

\[
\text{IF} \quad \text{condition1} \quad \text{process1} \quad \text{condition2} \quad \text{process2} \quad \ldots
\]

with the first satisfied condition in the list determining which process is executed.

Repetition

This is achieved by means of
WHILE condition
  process

Replication

Replication of a process can be based on any of SEQ, PAR, ALT, or IF, although not all of these are available in the version of the Occam compiler used in the production of the software for this thesis. The general form of the sequential replicator is

SEQ index = start FOR number
  process

Types and Declarations

The basic data types in Occam are CHAN, TIMER, BOOL, BYTE, and INT. These are used as the basis of variable declarations. For instance, INT x defines an integer x whose scope is the process that follows the declaration, while [100]BYTE y defines a byte array with elements y[0] to y[99].

Procedures

An example of a procedure is

PROC cube(VAL INT x, INT y)
  y := x*(x*x)
  : -- cube

An example of the invocation of this procedure is cube(5,p). The quantity represented by x is passed by value, while the quantity represented by y is passed by reference, so that its value is changed by the procedure. Anything following a double hyphen is a comment.

Timer

A timer is a special type of channel. An example of the use of the timer is for timeout:

  VAL timeout IS 1000 :
  TIMER clock :
  SEQ
    clock ? now
ALT

    channel ? x
    process
    clock ? AFTER now PLUS timeout
    SKIP

The value of now is set to the time on commencing the process, and if the specified channel has not provided a value within the elapsed time specified by timeout the dummy process SKIP is executed and the ALT process terminates.

Configuration

The distribution of Occam processes over a network of transputers is achieved by means of the PLACED PAR construct, and declared channels are associated with actual links ('hard' channels) by means of PLACE channel AT link.

D.0.11 Transputer

A useful description of the INMOS transputer is given by Whitby-Strevens [292]. A block diagram of the IMS T414 transputer is provided in Figure D.1. It is based on a 32-bit central processing unit, and has four serial links to provide inter-transputer communication. Each transputer has 2K Bytes of on-chip static RAM, and has a memory management unit to provide support for a wide range of external memory types. Error handling is supported by means of an error pin and an analyse pin. The transputer can be bootstrapped from ROM or from a link, depending on the state of an input pin.

Inter-Transputer Links

In the case of the system described in this thesis the links have been set at 10 MBits per second. The hardware link protocol is as follows. The data values are sent as bytes, each byte being prefixed by two high bits and followed by a single low bit. An acknowledge is sent as soon as transmission commences, and consists of a high bit and a low bit. Thus the theoretical data rate for each transputer, assuming bidirectional data transfer over all four links, is 3.2MBytes per second.
Figure D.1: T414 Transputer.
Appendix E

Transputer Image Processing System

The software of the transputer image processing system (TIPS) consists of the host, which provides an interface to the PC computer, and the network, where the images reside and where the image-processing operations are performed. The software has been significantly compressed in order to provide brevity at the same time as providing sufficient information to enable the software to be substantially reconstructed from these listings. Most lines have been provided as a fold, with the contents of the fold being either explicitly or implicitly obvious. Where a fold is subsequently opened a page reference is provided.

Some examples follow to help clarify this approach.

... exiting, report.progress, proc.details[all procs.] := FALSE

can be expanded to produce

exiting := FALSE
report.progress := FALSE
SEQ proc = 0 FOR full.network
  proc.details[proc] := FALSE

Similarly the code

... IF choice = 'A' : for.add()
... similarly for other possible choices

will expand into
IF
c choice = 'A'
   for.add()
choice = 'B'
   for.border()
... and so on for 'C', 'D', etc.

Where the software is that provided by INMOS as part of the transputer development system it has not been included.

E.1 Host Software

E.1.1 Program for the Host

Main program  PROC host.tips(CHAN keyboard, screen, [4]CHAN from.uf, to.uf)
VAL version.date IS [26, 09, 1988] : -- identify version
... COMMENT on the Transputer Image Processing System (See Page 249)
... COMMENT on channel message protocols (See Page 251)
... filers.tsr -- TDS header, etc
... VALUES (See Page 254)
... variables (See Page 258)
... utilities -- PROCs for I/O (See Page 260)
... file handling -- PROCs for file handling (See Page 267)
... local -- local house-keeping PROCs (See Page 268)
... host images -- handling images on the host (See Page 270)
... distribution -- distribution of images over the network (See Page 279)
... miscellaneous -- various PROCs on the host (See Page 280)
... network images -- images on the network (See Page 288)
... main program (See Page 259)
: -- host.tips

COMMENT on the Transputer Image Processing System  TDS access to the IBM-PC text attributes and graphics

Change to graphics mode by writing specific eight-bit characters to the screen. This is to over-ride systems such as the Transputer Development System (TDS) which does not provide access to the graphics through the normal procedures and will attempt to reset the screen to text if it has been placed in graphics mode. Accordingly while in these special modes access to the SET-MODE video-service function is blocked apart from the desired mode (controlled via the setup.flag). Two modes are provided in both low (320x200) and high (640x200) resolution graphics:

1. writing individual dots (of a specified colour in the case of low resolution graphics)
2. 'painting' sequential points (with specified colours in the case of low resolution graphics).

In addition, control is provided of the text-mode attributes.
The software is placed in protected memory once (e.g. by placing GR in the autoexec.bat file). The routine is entered from the video services interrupt and the special modes can be invoked by any one of

- **WRITE ATTRIBUTE/CHARACTER** (type 09H)
- **WRITE CHARACTER ONLY** (type 0AH)
- **WRITE TELETYPET** (type 0EH)

using the following codes:

- **SET LOW-RESOLUTION GRAPHICS MODE** 0FEH (254)
- **SET HIGH-RESOLUTION GRAPHICS MODE** 0FCH (252)
- **SET LOW-RESOLUTION PAINTING MODE** 0FBH (251)
- **SET HIGH-RESOLUTION PAINTING MODE** 0FAH (250)
- **SET CHARACTER-ATTRIBUTE MODE** 0F9H (249)
- **SET CHARACTER-ATTRIBUTE MODE WITH CURSOR UPDATING** (NO SCROLLING) 0F8H (248)
- **SET CHARACTER-ATTRIBUTE MODE WITH CURSOR UPDATING AND SCROLLING** 0F7H (247)
- **SET MIXED LOW-RESOLUTION MODE** 0F6H (246)
- **SET MIXED HIGH-RESOLUTION MODE** 0F5H (245)
- **SET ACTIVE-DISPLAY-PAGE MODE** 0F4H (244)
- **SET CURSOR-POSITION MODE** 0F3H (243)
- **SET SCREEN-CLEAR MODE** 0F2H (242)
- **SET CONTINUOUS-ATTRIBUTE MODE** 0F1H (241)
- **CLEAR CONTINUOUS-ATTRIBUTE MODE** 0F0H (240)
- **SET BLANKING MODE** 0EFH (239)

Video service calls of the SET_MODE type are disabled while in graphics mode. Other video service calls require that relevant flags be disabled for the duration of the call. The WRITE_DOT call is established by sending

```plaintext
row_low row_high col_low col_high [colour]
```

as 8-bit values, where

- **row_low** is a 8-bit value equal to \((row \mod 256)\)
- **row_high** is a 8-bit value equal to \((row \div 256)\)
- **col_low** is a 8-bit value equal to \((col \mod 256)\)
- **col_high** is a 8-bit value equal to \((col \div 256)\)
- **colour** is a 8-bit value representing colour (low-resolution mode only)

This is invoked using either low (0FEH) or high (0FCH) resolution. The video is returned to text mode on encountering any address which is outside the valid screen values.

Alternatively a sequence of locations can be filled by means of painting in which low resolution is encoded as

250
where the screen is returned to text mode on the next write-character command. Pixels that would
overflow the screen overwrite the current bottom line. There are four two-bit colours per byte, starting at
the lower-order bits. The high resolution paint is encoded as for low resolution but using eight single-bit
(on-off) pixel values per byte. In both cases the count consists of two bytes and is the number of bytes
being written (i.e. pixels/4 or pixels/8 respectively).

The mixed modes involve a mixture of painting and direct graphics and is encoded as

\[ \text{row1 column1 row2 column2 [paint_bytes]} \mid \text{[graphics_bytes]} \]

where the rows and columns are encoded as low_byte, high_byte, and the '1' and '2' refer to the top-left
and bottom-right corner of the window respectively.

The text-mode attribute for single characters involves sending 0F8H or 0F9H followed by the sequence

\[ \text{character attribute} \]

The character with the specified attribute is placed at the current cursor position and the cursor remains
at that position (0F9H) or is updated (0F8H). Subsequent writes with specific attributes require the
appropriate prefix for each character sent. Alternatively a sequence of characters with a specified attribute
can be sent by prefixing with 0F1H and following with 0F0H. Blanking is achieved using 0EFH followed by

\[ \text{rows columns attribute} \]

and is relative to the current cursor position.

The active display page can be selected by sending 0F4H followed by a byte containing the page number.
The cursor position can be set for the currently active page by sending 0F3H followed by two bytes
representing row and column. The currently active page can be cleared by sending 0F2H. The code
occupies approximately 2390 bytes in memory.

COMMENT on channel message protocols  Channel protocols: (number in brackets is
number of values, including tag)
Host to Network: <processor> <number> <tag>
- host-to-network tags and protocols

\[
\begin{align*}
tag = 1 & \implies \text{load an image : hn.load.image} \quad \text{(rows*cols+4)} \\
tag = 2 & \implies \text{send an image to host : hn.send.image} \\
tag = 3 & \implies \text{add two images : hn.add.images} \quad \text{(image 1) (image 2) (truncate(BOOL))} \\
tag = 4 & \implies \text{move an image : hn.move.image} \quad \text{(image 1) (image 2) (row shift) (col shift) (border)}
\end{align*}
\]
tag = 5 ==> copy an image : hn.copy.image
  <image 1> <image2>

(3)
tag = 6 ==> subtract two images : hn.subtract.images
  <image 1> <image2> <truncate(BOOL)>

(4)
tag = 7 ==> subtract a constant from an image : hn.constant
  <image> <constant> <truncate(BOOL)>

(4)
tag = 8 ==> shrink an image : hn.shrink
  <image1> <image2> <width>

(4)
tag = 9 ==> obtain the outline of an image : hn.outline
  <image>

(3)
tag = 10 ==> subtract a border from an image : hn.border.
  <image> <width>

(3)
tag = 11 ==> perform a rank filter : hn.rank.filter
  <image 1> <image2> <rank>

(4)
tag = 12 ==> perform a logical operation on an image : hn.logical
  <image 1> <image2> <operation>(BYTE)>

(4)
tag = 13 ==> obtain statistics of an image : hn.statistics
  <image> <border>

(3)
tag = 14 ==> threshold an image : hn.threshold
  <image> <threshold>

(3)
tag = 15 ==> expand an image : hn.expand.image
  <image 1> <image2> <maximum> <minimum> <border>

(6)
tag = 16 ==> form the convex hull of an image : hn.convex.hull
  <image 1> <image2> <border>

(4)
tag = 17 ==> rotate an image : hn.rotate
  <image 1> <image2> <full image size> <border>

(5)
tag = 18 ==> find maximum and minimum pixel values : hn.request.max.min
  <image> <border>

(3)
tag = 19 ==> set threshold value for use by fast threshold : hn.set.threshold
  <iterations>

(2)
tag = 20 ==> perform fast threshold (binary) : hn.fast.threshold
  <image> <border>

(3)
tag = 21 ==> setup for subtract constant : hn.set.subtract.constant
  <constant>

(2)
tag = 22 ==> perform fast subtract constant : hn.fast.constant
  <image>

(2)
tag = 40 ==> set iterations count : hn.iterations
  <iterations>

(2)
tag = 41 ==> provide information on location in network : hn.neighbours
  <L.H.S.> <R.H.S.> <top> <bottom> <number of columns> <network row>
  <network column> <centre of row (BOOL)> <single column (BOOL)>

(10)
tag = 42 ==> zero inactivity registers : hn.zero.inactivity

(1)
tag = 43 ==> read inactivity registers : hn.read.inactivity

(1)
tag = 44 ==> set details flag : hn.set.details

(1)
tag = 44 ==> clear details flag : hn.clear.details

(1)
tag = 44 ==> send details : hn.send.details

(1)
tag = 50 ==> purge network : hn.purge
  <seconds>

(2)
tag = 51 ==> return log : hn.log

(1)
tag = 52 ==> check network status : hn.check

(1)
tag = 53 ==> test the network : hn.test

(1)

Network to Host: <processor> <number> <tag>
- network-to-host tags and protocols

252
tag = 101 ==> sending image : nh.sending.image
(image number) <processor> <pixel count> [pixels]
(4+pixelcount)
tag = 102 ==> sending statistics : nh.sending.statistics
(image) <area> <total> <processor> <sum> <max> <min>
<max> <cmax> <min> <cmin>
tag = 201 ==> finished load an image : nh.finished.load.image
(image number) <processor>
tag = 202 ==> finished send an image to host : nh.finished.send.image
(image number) <processor>
tag = 203 ==> finished add two images : nh.finished.add.images
(image1) <image2> <processor>
tag = 204 ==> finished move an image : nh.finished.move.image
(image1) <image2> <processor>
tag = 205 ==> finished copy an image : nh.finished.copy.image
(image1) <image2> <processor>
tag = 206 ==> finished subtract two images : nh.finished.subtract.images
(image1) <image2> <processor>
tag = 207 ==> finished subtract a constant from image : nh.finished.constant
(image number) <processor>
tag = 208 ==> finished shrink an image : nh.finished.shrink
(image1) <image2> <processor>
tag = 209 ==> finished obtain the outline of an image : nh.finished.outline
(image1) <image2> <processor>
tag = 210 ==> finished subtract a border from an image : nh.finished.border
(image number) <processor>
tag = 211 ==> finished perform a rank filter : nh.finished.rank.filter
(image1) <image2> <processor>
tag = 212 ==> finished perform a logical operation : nh.finished.logical
(image1) <image2> <processor>
tag = 213 ==> finished obtain statistics of an image : nh.finished.statistics
(image number) <processor>
tag = 214 ==> finished threshold an image : nh.finished.threshold
(image number) <processor>
tag = 215 ==> finished expand an image : nh.finished.expand
(image1) <image2> <processor>
tag = 216 ==> finished form the convex hull : nh.finished.convex.hull
(image1) <image2> <processor>
tag = 217 ==> finished rotate an image : nh.finished.rotate
(image1) <image2> <processor>
tag = 218 ==> request maximum and minimum pixels : nh.max.min
<processor> <image> <max> <min>
tag = 219 ==> finished fast threshold : nh.finished.fast.threshold
(image) <processor> <sum>
tag = 300 ==> acknowledge : nh.acknowledge
<source>
tag = 301 ==> log contents : nh.log.contents
<source> | link, mes, time |
tag = 302 ==> response to test : nh.test.response
<proc.id> <day> <month> <year> <displacement>
tag = 303 ==> error : nh.error
<source> <errorcode> <errorvalue>
tag = 304 ==> inactivity registers : nh.inactivity

253
Network to Network: <processor> <number> <tag>
- network-to-network tags and protocols

\[ \text{tag} = 305 \implies \text{details: nh.details} \]

\[ \text{tag} = 1000 \implies \text{request image segment: nn.request.image.segment} \]
\[ \text{<requesting processor> <source image> <row start> <column start> <row step> <column step> <border> <destination image> <destination row start> <destination column start>} \]

\[ \text{tag} = 1001 \implies \text{send image segment: nn.sending.image.segment} \]
\[ \text{<source processor> <destination image> <row start> <column start> <row step> <column step> <border> <destination row start> <destination column start> <number of pixels> | <pixels>} | \]
\[ \text{(numberofpixels+11)} \]

\[ \text{tag} = 1002 \implies \text{request row of image: nn.request.row} \]
\[ \text{<image1> <image2> <requesting processor> <row> <border>} \]

\[ \text{tag} = 1003 \implies \text{sending row of image: nn.sending.row} \]
\[ \text{<image1> <image2> <source processor> <source column position> <row number> <border> <number of pixels> | <pixels>} | \]
\[ \text{(numberofpixels+8)} \]

\[ \text{tag} = 1004 \implies \text{request image segment: nn.request.image.segment.rotate} \]
\[ \text{<requesting processor> <source image> <row start> <column start> <row step> <column step> <border> <destination image> <destination row start> <destination column start>} \]

\[ \text{Hard link message format:-} \]

destination.processor; number.of.values [:;values]; end.of.message

| | | | | |
| | | | | |

0-19 for network  \( ---------- \) not included
end.proc => no. 19 \( \text{in count} \)
host.proc => host
broadcast => sequentially 0 to 19

VALUES ... character attributes and graphics (See Page 255)
... creating boxes on screen (See Page 255)
... screen boxes (See Page 255)
... screen pages (See Page 255)
... errors (See Page 256)
... images (See Page 256)
... 'details' flags (See Page 256)
... host-to-network tags (See Page 256)
... network-to-host tags (See Page 257)
... network-to-network tags (See Page 257)
... network (See Page 257)
... tables and packets (See Page 257)
... clocks (See Page 258)
Character attributes and graphics

VAL to.low.res IS BYTE #FE :
VAL to.attr IS BYTE #F9 :
VAL to.low.mixed IS BYTE #F6 :
VAL set.cur.pos IS BYTE #F3 :
VAL to.continuous IS BYTE #F1 :
VAL to.blank IS BYTE #EF :
VAL attr.query IS BYTE 29 :
VAL attr.query.box IS BYTE 29 :
VAL attr.heading IS BYTE 62 :
VAL attr.info IS BYTE 106 :
VAL attr.choice IS BYTE 106 :
VAL attr.flash.info.box IS BYTE 234 :

Creating boxes on screen

VAL box.tl IS BYTE 218 :
VAL box.bl IS BYTE 192 :
VAL box.br IS BYTE 196 :
VAL dbox.tl IS BYTE 201 :
VAL dbox.bl IS BYTE 200 :
VAL dbox.h IS BYTE 205 :
VAL box.tr IS BYTE 191 :
VAL box.br IS BYTE 217 :
VAL box.v IS BYTE 179 :
VAL dbox.tr IS BYTE 187 :
VAL dbox.br IS BYTE 188 :
VAL dbox.v IS BYTE 186 :

Screen boxes

VAL heading IS "MULTITRANSPUTER IMAGE PROCESSING SYSTEM":
VAL size.heading.2 IS ((SIZE(heading))+2):
VAL heading.t.row IS 1 :
VAL heading.b.row IS heading.t.row :
VAL heading.l.col IS 40-(size.heading.2/2) :
VAL heading.r.col IS 40+(size.heading.2/2) :
VAL query.t.row IS heading.b.row+3 :
VAL query.b.row IS query.t.row :
VAL query.l.col IS 10 :
VAL query.length IS query.r.col-query.l.col :
VAL info.t.row IS query.t.row+3 :
VAL info.b.row IS 22 :
VAL info.l.col IS 10 :
VAL info.length IS info.r.col-info.l.col :
VAL choice.t.row IS heading.b.row+3 :
VAL choice.b.row IS 22 :
VAL choice.l.col IS 1 :

Screen pages

255
VAL no. of screens IS 4 :
VAL work page IS 1 :
VAL choice page IS 0 :
VAL help page IS 2 :

Errors

VAL test.timeout IS 1 :
VAL test.dest.not.host IS 3 :
VAL test.proc.out.of.range IS 5 :
VAL wrong.accept.tag IS 102 :
VAL wrong.value IS 104 :
VAL wrong.source IS 106 :
VAL faulty.termination IS 108 :
VAL wrong.count IS 110 :
VAL faulty.message.link.id IS 112 :
VAL no.free.tables IS 114 :
VAL wrong.image.size IS 116 :
VAL wrong.image.number.receive.row IS 118 :
VAL wrong.image.number.for.hull IS 120 :
VAL wrong.col.dividers IS 122 :
VAL wrong.dividers IS 124 :
VAL wrong.number.cols IS 126 :
VAL wrong.col.diff IS 128 :
VAL wrong.rotate.col IS 130 :
VAL test.escaping IS 2 :
VAL test.wrong.tag IS 4 :
VAL wrong.dest IS 101 :
VAL wrong.repeat IS 103 :
VAL long.wait IS 105 :
VAL faulty.message IS 107 :
VAL wrong.proc.tag IS 109 :
VAL faulty.message.contents IS 111 :
VAL wrong.table.number IS 113 :
VAL wrong.image.number IS 115 :
VAL wrong.image.number.request.row IS 117 :
VAL wrong.image.number.do.hull IS 119 :
VAL wrong.row.dividers IS 121 :
VAL wrong.source.proc.for.rotate IS 123 :
VAL wrong.number.rows IS 125 :
VAL wrong.row.diff IS 127 :
VAL wrong.rotate.row IS 129 :

Images

VAL forever IS TRUE :
VAL rows IS 100 :
VAL cols IS 100 :
VAL border IS 1 :
VAL image.buffer.size IS 100 :
VAL image.buffer.size.1 IS image.buffer.size - 1 :
VAL total.images IS 5 :
VAL total.images.1 IS total.images - 1 :
VAL max.name.length IS 20 :
VAL B0 IS BYTE 0 :
VAL B3 IS BYTE 3 :

Details flags

VAL d.num.dividers IS INT 1 :
VAL d.source IS INT 3 :
VAL d.request.send IS INT 5 :
VAL d.divider IS INT 2 :
VAL d.row.col IS INT 4 :
VAL d.request.received IS INT 6 :

Host-to-network tags

VAL hn.load.image IS INT 1 :
VAL hn.add.images IS INT 3 :
VAL hn.copy.image IS INT 5 :
VAL hn.constant IS INT 7 :
VAL hn.send.image IS INT 2 :
VAL hn.move.image IS INT 4 :
VAL hn.subtract.images IS INT 6 :
VAL hn.shrink IS INT 8 :
VAL hn.outline IS INT 9 :
VAL hn.rank.filter IS INT 11 :
VAL hn.statistics IS INT 13 :
VAL hn.expand.image IS INT 15 :
VAL hn.rotate IS INT 17 :
VAL hn.set.threshold IS INT 19 :
VAL hn.fast.constant IS INT 21 :
VAL hn.set.subtract.constant IS INT 22 :
VAL hn.set.subtract.constant IS INT 22 :
VAL hn.border IS INT 10 :
VAL hn.logical IS INT 12 :
VAL hn.threshold IS INT 14 :
VAL hn.convex.hull IS INT 16 :
VAL hn.request.max.min IS INT 18 :
VAL hn.fast.threshold IS INT 20 :
VAL hn.neighbours IS INT 41 :
VAL hn.read.inactivity IS INT 43 :
VAL hn.clear.details IS INT 45 :
VAL hn.purge IS INT 50 :
VAL hn.check IS 52 :

Network-to-host tags

VAL nh.sending.image IS INT 101 :
VAL nh.finished.load.image IS INT 201 :
VAL nh.finished.add.images IS INT 203 :
VAL nh.finished.copy.image IS INT 205 :
VAL nh.finished.constant IS INT 207 :
VAL nh.finished.outline IS INT 209 :
VAL nh.finished.rank.filter IS INT 211 :
VAL nh.finished.statistics IS INT 213 :
VAL nh.finished.expand IS INT 215 :
VAL nh.finished.rotate IS INT 217 :
VAL nh.finished.fast.threshold IS INT 219 :
VAL nh.log.contents IS 301 :
VAL nh.error IS 303 :
VAL nh.details IS INT 305 :

Network-to-network tags

VAL nn.request.image.segment IS INT 1000 :
VAL nn.sending.image.segment IS INT 1001 :
VAL nn.request.row IS INT 1002 :
VAL nn.sending.row IS INT 1003 :
VAL nn.request.image.segment.rotate IS INT 1004 :

Network

VAL full.network IS 20 :
VAL low.row IS 4 :
VAL low.col IS 1 :
VAL host.proc IS MOSTNEG INT :
VAL end.proc IS MOSTPOS INT :
VAL no.neighbour IS (MOSTNEG INT)/8 :

Tables and packets

VAL packet.size.limit IS 15 :
VAL max.tables IS 10 :
VAL max.table.size IS 4020 :
VAL table.source.id IS max.tables+2 :
VAL table.sink.id IS max.tables+3 :
VAL biggest.packet IS 1000000 :
VAL unspecified IS -1 : -- for packet sizes

Clocks

VAL tptr.l.ticks.per.second IS 15625 : -- low priority process
VAL time.allowed IS tptr.l.ticks.per.second*1 :
VAL setup.delay IS tptr.l.ticks.per.second*1 :
VAL unlimited.time IS 0 :

page:miscellaneous

VAL otherwise IS TRUE :
VAL broadcast IS (MOSTPOS INT)/4 :
VAL escape IS BYTE 223 :
VAL max.log IS 200 :

Other VALues

VAL image.buffer.size IS 100 :
VAL image.buffer.size.1 IS image.buffer.size - 1 :
VAL total.images IS 10 :
VAL short.pause IS 8 : -- seconds
VAL B0 IS BYTE 0 :
VAL B3 IS BYTE 3 :

Variables

-- channels
CHAN to.network, from.network :
PLACE to.network AT 2 :
PLACE from.network AT 6 :

-- image variables
[image.buffer.size][image.buffer.size]BYTE image.buffer :
[total.images]INT image.rows, image.cols :
[total.images][max.name.length]BYTE image.name :
[total.images]BOOL image.name.set :

-- BOOLean variables
[full.network]BOOL replied :
[full.network]BOOL proc.details :
BOOL profile, fatal, contour, display, ok, exiting, notime, report.progress :
BOOL fast.threshold.set, bell.on, single.column, fast.constant.set, details :
[no.of.screens]BOOL erased : -- screen

-- INTeger variables
E.1.2 Main program

... BOOL network.not.found :
... initialize (See Page 259)

... link.flush(from.network), initial.info(network.not.found) -- menu onto screen
... IF network.not.found SKIP
otherwise
... SEQ choose(choice) setup.workpage()
... WHILE NOT exiting
... determine choice \pageref{page:determine$choice})
... initialize
... IF NOT exiting
... link.flush(from.network) -- clean up network before next choice
select.page(choice.page)
... IF erased[choice.page] choices() -- restore menu
... choose(choice), clock ? last.time -- start timing from here
select.page(0) -- before finishing

initialize

... exiting, report.progress, proc.details[all procs.] := FALSE, bell.on := TRUE
SEQ img = 0 FOR total.images
SEQ sc.page = 0 FOR no.of.screens eraser[sc.page] := TRUE
iterations := 1

Determine choice

... IF choice = 'A' : for.add()
... similarly for other possible choices

... testing network (See Page 260)
o otherwise valid := FALSE
Testing network

```occam
... setup.workpage(), blot.info(), lay := 1, insert.info("Testing ... ", lay)
    test.network(not.ok, err.code, pps, dd, mm, yy, lay, disp)
... lay := 1, blot.info.line(lay)
... IF not.ok : insert.info("Failed to Test Network ", lay)
... OTHERWISE provide details
    pause.query()
```

E.1.3 Input Output Utilities

```occam
... slice stuff (This is part of the standard software supplied with the TDS)
... COMMENT on string conventions (See Page 260)
... PROC delay (See Page 260)
... general PROCs to write to the screen (See Page 260)
... special PROCs to write to the screen (See Page 262)
... PROCs to read from the keyboard (See Page 264)
... PROC query.and.update.info (See Page 266)
... PROC y.n.query.and.update.info (See Page 266)
... PROC input.int -- from network (See Page 266)
```

String conventions used in these procedures The standard OCCAM strings are placed in arrays starting at element zero.
e.g. `str := "Hello"` has `str[0] = 'H'` and `str[4] = 'o'`, with `SIZE str = 5`.
The PROCEDures that use these strings are prefixed with 'write'.
e.g. write("Hello").

A major drawback with this convention is that the string length must, in general, be carried through separately from the string itself.
e.g. given [20]BYTE str, if str is set to "Hello" `SIZE str` will return 20, not 5.

An alternative convention places the length in element zero.
e.g. for the string "Hello", `str[0] = 5`, `str[1] = 'H'`, etc.
The PROCedures that use these strings are prefixed with 'print'.

PROC delay

```occam
PROC delay(VAL INT seconds)
    INT now, interval :
    ... SEQ clock ? now, interval := tptr.l.ticks.per.second*seconds
    ALT
    ... clock ? AFTER (now PLUS interval) : SKIP -- timeout
    ... keyboard ? ch : SKIP -- or any key to exit
    : -- delay
```

General PROCs to write to the screen
PROC send(VAL BYTE ch)
    ... SEQ screen ! tt.out.byte; ch
    : -- send
    ... PROC sound.bell(VAL INT duration)
    ... PROC clear.screen()
    ... PROC position(VAL INT row, col)
    ... PROC write(VAL []BYTE str)
    ... PROC char.atr(VAL BYTE char, VAL BYTE attribute)
    ... PROC char.at(VAL BYTE char, VAL INT y, x, VAL BYTE attribute)
    ... PROC char.to(VAL BYTE char, VAL INT y, x)
    ... PROC write.at(VAL []BYTE str, VAL BYTE attrib, VAL INT y, x)
    ... PROC write.attr(VAL []BYTE str, VAL BYTE attrib)
    ... PROC newline(VAL INT leading)
    ... PROC print.len.string(VAL INT len, VAL []BYTE str)
    ... PROC print(VAL []BYTE str)
    ... PROC print.number(VAL INT n, field, VAL BOOL leading.zeroes)
    ... PROC print.number.attr(VAL INT n, field, VAL BYTE attr, VAL BOOL leading.zeroes)
    ... PROC print.vector.attr(VAL INT x, y, VAL BYTE attr)

    ... PROC print.decimal (See Page 261)

    ... PROC blank.line(VAL INT row)
    ... PROC select.page(VAL INT page.number)
    ... PROC draw.vertical(VAL BYTE char, VAL INT number, y, x, VAL BYTE attribute)

PROC print.decimal

PROC print.decimal(VAL INT top, the.bot, decits, VAL BYTE attribute)
    -- Output a decimal correctly rounded. The denominator (the.bot) is positive.
    ... VAL digits.limit IS 4 : VAL mult IS [1, 10, 100, 1000, 10000] :
    INT remainder, dec.digits, bot, whole, part :
    ... SEQ send(to.continuous), send(attribute) -- establish attribute
    ... IF the.bot = 0 : write(" ERROR - ZERO DIVIDER")
    otherwise
        ... ensure denominator positive
        ... dec.digits := decits, whole := top/bot -- may be rounded up: don't print yet
        ... IF dec.digits > 0 -- calculate decimal part, adjust if necessary
            remainder := top REM bot
            ... make sure not too many decimal digits requested
            ... IF remainder <> 0
                ... adjust the remainder (See Page 262)

                ... otherwise part := 0
                ... otherwise SKIP -- no decimal part
                ... IF (top < 0) AND ((whole = 0) AND (part <> 0)) : send('-')
                ... otherwise SKIP
                print.number(whole, 1, FALSE)
                ... IF dec.digits > 0 : send('.', print.number(part, dec.digits, TRUE)

261
Adjust the remainder

INT adjustment : -- adjusted remainder -- use approximations to avoid overflow

... IF remainder > 99999999 : adjusted.rem := remainder/(bot/mult[dec.digits])
   (remainder > 999999) AND (dec.digits >= 2)
   adjusted.rem := ((remainder*mult[2])+(bot/2)/mult[dec.digits-2])/((bot/mult[dec.digits-2])
... otherwise adjusted.rem := ((remainder*mult[dec.digits])+(bot/2))/bot
adjustment := (adjusted.rem/mult[dec.digits])
... IF top < 0 : whole := whole - adjustment -- perform adjustment
... otherwise whole := whole + adjustment
part := adjusted.rem REM mult[dec.digits]

Special PROCs to write to the screen )

... PROC blot.line(VAL INT row, start.col, fin.col, VAL BYTE attrib)
... PROC blot.box(VAL INT start.row, fin.row, start.col, fin.col, VAL BYTE attrib)

... PROC blot.query (See Page 262)
... PROC blot.info.line (See Page 263)
... PROC blot.info (See Page 263)

... PROC make.box(VAL INT start.row, fin.row, start.col, fin.col, VAL BYTE attrib)
... PROC make.dbox(VAL INT start.row, fin.row, start.col, fin.col, VAL BYTE attrib)
... PROC make.heading()

... PROC make.info (See Page 263)
... PROC make.choice (See Page 263)
... PROC make.query (See Page 263)
... PROC insert.query (See Page 263)
... PROC pause.query (See Page 264)
... PROC insert.info(VAL BYTE str, INT layer)
... PROC insert.mes.integer (See Page 264)
... PROC insert.mes.decimal (See Page 264)
... PROC setup.workpage (See Page 264)

PROC blot.query

PROC blot.query()
... SEQ blot.box(query.t.row, query.b.row, query.l.col, query.r.col, attr.query)
: -- blot.query
PROC blot.info.line

PROC blot.info.line(VAL INT layer)
  VAL INT row IS info.t.row+(layer-1) :
  ... SEQ blot.line(row, info.l.col, info.r.col, attr.info)
  : -- blot.info.line

PROC blot.info

PROC blot.info()
  ... SEQ blot.box(info.t.row, info.b.row, info.l.col, info.r.col, attr.info)
  : -- blot.info

PROC make.info

PROC make.info()
  -- Prepare for information.
  ... make.dbox(info.t.row, info.b.row, info.l.col, info.r.col, attr.info.box)
    blot.box(info.t.row, info.b.row, info.l.col, info.r.col, attr.info)
  : -- make.info

PROC make.choice

PROC make.choice()
  -- Prepare for information.
  ... make.dbox(choice.t.row, choice.b.row, choice.l.col, choice.r.col, attr.choice.box)
    blot.box(choice.t.row, choice.b.row, choice.l.col, choice.r.col, attr.choice)
  : -- make.choice

PROC make.query

PROC make.query()
  -- Prepare for information.
  ... make.box(query.t.row, query.b.row, query.l.col, query.r.col, attr.query.box)
    blot.box(query.t.row, query.b.row, query.l.col, query.r.col, attr.query)
  : -- make.query

PROC insert.query

PROC insert.query(VAL []BYTE str)
  -- Place a query in the query box.
  ... VAL len IS SIZE(str) : INT row, col :
  ... row := query.t.row, position(row, query.l.col+2), col := (query.l.col+len)+2
  ... send(to.continuous), send(attr.query)
  ... SEQ s.pt = 0 FOR len : send(str[s.pt])
  ... send(to.discontinuous), position(row, col)
  : -- insert.query
PROC pause.query

PROC pause.query()
   -- Wait for keyboard entry before proceeding. Clear existing keystrokes from buffer.
   ... TIMER hold : BYTE ch : BOOL cleared : INT dummy, now :
   ... hold ? now, cleared := FALSE
   WHILE NOT cleared
      ALT
         ... keyboard ? dummy : hold ? now
         ... hold ? AFTER now PLUS 1000 : cleared := TRUE
         ... insert.query(" Press any key to continue ... "), keyboard ? ch, blot.query()
      : -- pause.query

PROC insert.mes.integer

PROC insert.mes.integer(VAL []BYTE mes, VAL INT result, INT lay)
   -- Place message and integer onto given layer of info box. Update layer pointer.
   ... insert.info(mes, lay), lay := lay-1 -- back to original line
   ... print.number.attr(result, 1, attr.info, FALSE), lay := lay+1 -- update line
   : -- insert.mes.integer

PROC insert.mes.decimal

PROC insert.mes.decimal(VAL []BYTE mes, VAL INT num, den, dpoints, INT lay)
   -- Place message and decimal onto given layer of info box. Update layer pointer.
   ... insert.info(mes, lay), lay := lay-1 -- back to original line
   ... print.decimal(num, den, dpoints, attr.info), lay := lay+1 -- update line
   : -- insert.mes.decimal

PROC setup.workpage

PROC setup.workpage()
   ... select.page(work.page)
   ... IF erased[work.page]
      ... send(to.clear.page), make.heading(), make.info()
      ... make.query(), erased[work.page] := FALSE
      ... otherwise : blot.info(), blot.query()
   : -- setup.workpage

PROCs to read from the keyboard

   ... PROC echo(BYTE ch)
   ... PROC noecho(BYTE ch)
   ... PROC read.string([]BYTE str)

   ... PROC read.y (See Page 265)
   ... PROC extract.number.s(VAL []BYTE str, INT n, INT start.end, BOOL neg, ok)
PROC read.y

PROC read.y(BOOL got.y)
-- Respond with TRUE if keyboard is 'y' or 'Y', FALSE otherwise
-- Clear any existing keystrokes from the buffer

WHILE NOT cleared
    ALT
        keyboard ? dummy : hold ? now
        hold ? AFTER now PLUS 1000 : cleared := TRUE
    noecho(ch)
    IF (ch = 'y') OR (ch = 'Y') : got.y := TRUE
    OTHERWISE : got.y := FALSE

PROC num.in

PROC num.in(INT num)
    INT point : [256]BYTE str : BOOL ok, sgn :
    read.string(str), point := 1, extract.number.s(str, num, point, sgn, ok)

PROC input.integer

PROC input.integer(VAL INT mes.displace, INT return, VAL INT lowest, highest)
    VAL row is query.t.row VAL col is (query.l.col+2)+mes.displace : BOOL acceptable :
    acceptable := FALSE
    WHILE NOT acceptable
        position(row, col), num.in(return)
        IF (return >= lowest) AND (return <= highest) : acceptable := TRUE
        OTHERWISE : blot.line(row, col, query.r.col-1, attr.query)

PROC input.decimal

PROC input.decimal(VAL []BYTE str, VAL INT row, col, VAL BYTE attr, INT num, den, BOOL ok)
-- str is a standard OCCAM string
    VAL len is SIZE(str) :
        write.at(str, attr, col, row), position(row, col+len), decimal.in(num, den, ok)

PROC query.and.update.info

PROC query.and.update.info(VAL []BYTE mes, INT x, VAL INT low, high, INT lay)
-- Message to query box, return value x within range low to high, blot query box,
-- write message in info box, append x in info box, update layer number.
... insert.query(mes), input.integer(SIZE(mes), x, low, high)
... blot.query(), blot.info.line(lay), insert.info(mes, lay)
... lay := lay-1, print.number.attr(x, 1, attr.info, FALSE), lay := lay+1
: -- query.and.update.info

PROC y.n.query.and.update.info

PROC y.n.query.and.update.info(VAL []BYTE mes, BOOL return, INT lay)
-- Message to query box, return Boolean value, blot query box,
-- write message in info box, append Boolean in info box, update layer number.
... insert.query(mes), write.attr(" (Y or [N]) ? ", attr.query), read.y(return)
... blot.query(), blot.info.line(lay), insert.info(mes, lay)
... IF return : write.attr(": TRUE", attr.info)
... otherwise, write.attr(": FALSE", attr.info)
: -- y.n.query.and.update.info

PROC input.int

PROC input.int(INT return, VAL INT timeout, BOOL continuing, escaped, time.out)
-- Input from network
... BYTE ch : INT now :
... return := end.of.message
WHILE ((return = end.of.message) AND (continuing AND (NOT (escaped OR time.out))))
... IF timeout <> unlimited.time -- input with timeout
... IF (continuing AND (NOT escaped)) AND (NOT time.out)

... proceed with input with timeout (See Page 266)

... otherwise SKIP
otherwise
... IF (continuing AND (NOT escaped)) AND (NOT time.out)

... proceed with input without timeout (See Page 267)

... otherwise SKIP
: -- input.int

Proceed with input with timeout

... clock ? now
ALT

266
keyboard ? ch     -- finished if 'escape'
... IF ch = escape : continuing := FALSE, escaped := TRUE, sound.bell(1)
... otherwise : SKIP
... from.network ? return : SKIP
... clock ? AFTER now PLUS timeout : continuing := FALSE, time.out := TRUE

Proceed with input without timeout

-- finished if 'escape'
... clock ? now
ALT
keyboard ? ch
... IF ch = escape : continuing := FALSE, escaped := TRUE, sound.bell(1)
... otherwise : SKIP
from.network ? return : SKIP

E.1.4 File Handling

... PROCs to read from the stream (See Page 267)
... PROCs to write to the stream (See Page 267)
... PROCs to call the user filer (See Page 267)

PROCs to read from the stream

... PROC get.stream.result(CHAN fs, INT result)
... PROC get.stream.attr(CHAN fs, BYTE tag, [attr.size+1]INT attr, INT result)
... PROC get.stream.data(CHAN fs, BYTE tag, []BYTE buffer, INT result)

PROCs to write to the stream

... PROC put.stream.attr(CHAN ts, VAL [attr.size+1]INT attr)
... PROC put.stream.endfold(CHAN ts)
... PROC put.stream.endfiled(CHAN ts)
... PROC put.stream.endstream(CHAN ts)
... PROC put.stream.number(CHAN ts, VAL INT n)
... PROC put.stream.result(CHAN ts, VAL INT r)
... PROC put.stream.data(CHAN ts, VAL BYTE tag, VAL []BYTE buffer)
... PROC open.stream(CHAN fs, ts, VAL BYTE op, VAL INT fold.no, INT result)
... PROC close.stream(CHAN fs, ts, INT result)

PROCs to call the user filer

... PROC send.command(CHAN from.uf, to.uf, VAL BYTE op, VAL INT seq.no, INT result)
... PROC number.of.folds(CHAN from.uf, to.uf, INT n, result)
... PROC create.fold(CHAN from.uf, to.uf, INT new.fold.number,
**E.1.5 Local Procedures**

**PROC** report.network.error

PROC report.network.error(INT layer)

--- Values and variables

VAL test.timeout IS tptr.ticks.per.second*2 :
BOOL test.count.down, test.timing.out, escaping :
INT source, errorcode, errorvalue :
... input source, errorcode, errorvalue
... print source and errorcode
... identify the error
... print error value

: -- report.network.error

**PROC** test.network

PROC test.network (BOOL testing.error, INT error.code,
INT proc.num, day, month, year, layer, displace)
... BOOL testing, dummy : INT dest, numb, tag :
... testing := TRUE, testing.error := FALSE
WHILE testing
VAL test.timeout IS tptr.ticks.per.second*2 :
... INT response : BOOL test.timing.out, escaping :
... initialize variables
to.network ! end.proc; 1; hn.test; end.of.message
... input dest, numb, tag, proc.num, day, month, year, displace from network

268
proc.num := proc.num+1  -- convert from [0...] scale to [1...] scale.

... IF test.timing.out
    ... error.code := test.timeout, proc.num := -1, testing.error := TRUE
    ... escaping : error.code := test.escaping, testing.error := TRUE
    otherwise
    ... testing.error := TRUE
    ... IF dest >= 0  -- any negative destination is acceptable
        ... tag = nh.error : report.network.error(layer)
        ... tag <> nh.test.response : error.code := test.wrong.tag
        (proc.num > 100) OR (proc.num < 0)
        ... error.code := test.proc.out.of.range
    ... otherwise : testing := FALSE, testing.error := FALSE
    ... IF testing.error = y.n.query.and.update.info("Repeat", testing, layer)
    ... otherwise : SKIP

:  -- test.network

PROC choices

PROC choices()

... VAL space IS BYTE #20 : VAL centre IS 40 : INT row, offset :
PROC present.choice(VAL []BYTE message, VAL BYTE indicator)
... write.at(message, attr.info, row, offset-(SIZE(message)))
... row := row + (OFFSET+40)/80, offset := (OFFSET+40)/80
... print space, indicator, space with attribute = attr.request
:  -- present.choice
... send(to.clear.page), make.heading(), make.choice()
... row := choice.t.row, offset := 36
VAL mes IS "Add two images " :
present.choice(mes, 'A')
... other choices available
VAL mes IS " CHOICE : ":  -- make choice
... row := row + 2, write.at(mes, attr.request, row, centre-(SIZE(mes)))
... position(row, centre-2), erased[choice.page] := FALSE
:  -- choices

PROC initial.info

PROC initial.info(BOOL not.ok)

VAL space IS BYTE #20 : VAL centre IS 45 :
INT error.code, row, d, m, y, procs, info.layer :
... select.page(work.page), setup.workpage(), info.layer:=1 -- prepare for test
insert.info("Testing Network ...", info.layer)

test.network(not.ok, error.code, procs, d, m, y, info.layer, displacement)
full.minus.displacement := (full.network - displacement),
half.displacement := (displacement+1)/2
... IF not.ok : info.layer := 1 -- back to first line
insert.info("Error in testing network:", info.layer)
... IF error.code = test.timeout
    insert.info("Timeout - no network response ", info.layer)
... identify other error codes
otherwise
... BOOL proceeding : INT lay :
... provide information on network
... make.query(), y.n.query.and.update.info("Proceed ", proceeding, lay)
... IF proceeding

... testing and network details (See Page 270)

... otherwise : not.ok := TRUE
: -- initial.info

Testing and network details

... INT pp, dd, mm, yy, ct.reps : BOOL ct :
... query.and.update.info("Number of columns of transputers to be used = ",
  no.of.cols, low.col, procs, lay)
... IF no.of.cols = 1 : single.column := TRUE
... otherwise : single.column := FALSE
query.and.update.info("Number of rows of transputers to be used = ",
  no.of.rows, low.row, procs/no.of.cols, lay)
no.in.network := no.of.cols*no.of.rows
insert.mes.integer("Number of transputers used is ",no.in.network, lay)
... segment.rows := rows/no.of.rows, segment.cols := cols/no.of.cols
... [4]INT neighbours : INT this.processor : BOOL centre.of.row :
... SEQ row = 0 FOR no.of.rows SEQ
col = 0 FOR no.of.cols
... this.processor := (row*no.of.cols)+col
... IF col = 0 : neighbours[0] := no.neighbour
... otherwise : neighbours[0] := this.processor-1
... similarly right-hand side, top, and bottom
... IF row = (no.of.rows-1) : neighbours[3] := no.neighbour
... otherwise : neighbours[3] := this.processor+no.of.cols
centre.of.row := (col = ((no.of.cols-1)/2))
to.network ! this.processor; 10; hn.neighbours
to.network ! no.of.cols; row; col; INT(centre.of.row);
INT(single.column); end.of.message
... pause.query(), select.page(choice.page), choices()
PROC do.display (See Page 276)

PROC do.contour (See Page 278)

PROC plot.point

PROC plot.point(VAL INT this.row, this.col, VAL BYTE colour)
-- plot a point on the low-res graphics with windowing
BYTE rl, rh, cl, ch :
... rh := BYTE(this.row/256), rl := BYTE(this.row REM 256) -- set up bytes
... ch := BYTE(this.col/256), cl := BYTE(this.col REM 256)
... send(rl), send(rh), send(cl), send(ch), send(colour) -- send bytes
: -- plot.point

PROC plot.point.w

PROC plot.point.w(VAL INT this.row, this.col, VAL BYTE colour)
-- plot a point on the low-res graphics with windowing
BYTE rl, rh, cl, ch :
... IF (this.row>=0)AND((this.row<=199)AND((this.col>=0)AND(this.col<=319)))
... rh := BYTE(this.row/256), rl := BYTE(this.row REM 256) -- set up bytes
... ch := BYTE(this.col/256), cl := BYTE(this.col REM 256)
... send(rl), send(rh), send(cl), send(ch), send(colour)
... otherwise : SKIP
: -- plot.point.w

PROC plot.point.shade

PROC plot.point.shade(VAL INT this.row, this.col, VAL BYTE colour)
-- plot a point on the low-res graphics with windowing and shading
BYTE rl, rh, cl, ch :
... IF (this.row>=0)AND((this.row<=199)AND((this.col>=0)AND(this.col<=319)))
... IF this.row < top.shade[this.col]
... rh := BYTE(this.row/256), rl := BYTE(this.row REM 256) -- set up
... ch := BYTE(this.col/256), cl := BYTE(this.col REM 256)
... send(rl), send(rh), send(cl), send(ch), send(colour)
temp.top.shade[this.col] := this.row -- update shade
... otherwise : temp.top.shade[this.col] := top.shade[this.col]
: -- plot.point.shade

PROC join.points.w

PROC join.points.w(VAL INT rv1, cv1, rv2, cv2, VAL BYTE colour)
-- Plot in terms of ends of line, but only within valid screen
PROC abs(VAL INT a, INT absolute)
... IF a >= 0 : absolute := a
... otherwise : absolute := -a
: -- abs
PROC swap(INT a, b, c, d)
-- Return so that b > a. If a and b are swapped, then so are c and d.
INT t :
... IF a > b : t := a, a := b, b := t -- swap first pair
... t := c, c := d, d := t -- swap second pair
... otherwise : SKIP -- no need to do anything
: -- swap
... INT r1, c1, r2, c2, r, c, r.diff, c.diff : BOOL r.swap, c.swap :
... r1 := rv1, c1 := cv1, r2 := rv2, c2 := cv2 -- parameters as variables
... abs(r2 - r1, r.diff), abs(c2 - c1, c.diff)
... IF (r.diff = 0) AND (c.diff = 0) : SKIP -- coincident points
... r.diff <= c.diff : swap(c1, c2, r1, r2) -- step by column
... SEQ c = c1 FOR c.diff : plot.point.v(r1+(r.diff*(c-c1))/c.diff),c,colour)
otherwise -- plot by row
... swap(r1, r2, c1, c2)
... SEQ r = r1 FOR r.diff : plot.point.w(r,c1+((c.diff*(r-r1))/r.diff),colour)
: -- join.points.w

PROC join.points.shade
PROC join.points.shade(VAL INT rv1, cv1, rv2, cv2, VAL BYTE colour)
-- In terms of ends of line. Plot only within valid screen and apply shading.
... PROC abs(VAL INT a, INT absolute)
... PROC swap(INT a, b, c, d)
INT r1, c1, r2, c2, r, c, r.diff, c.diff :
... r1 := rv1, c1 := cv1, r2 := rv2, c2 := cv2
... abs(r2 - r1, r.diff), r.diff := r.diff + 1
... abs(c2 - c1, c.diff), c.diff := c.diff + 1
... IF r.diff <= c.diff -- plot by column
... swap(c1, c2, r1, r2) -- ensure c2 > c1
r.diff := r2 - r1 -- need signed difference
SEQ c = c1 FOR c.diff -- step by column
plot.point.shade(r1 + (r.diff*(c-c1))/c.diff), c, colour)
otherwise -- plot by row
... swap(r1, r2, c1, c2) -- ensure r2 > r1
c.diff := c2 - c1 -- need signed difference
SEQ r = r1 FOR r.diff -- step by row
plot.point.shade(r, c1 + ((c.diff*(r-r1))/r.diff), colour)
: -- join.points.shade

PROC obtain.image.number
PROC obtain.image.number(VAL []BYTE mes, INT img)
... make.query(), insert.query(mes),
... input.integer(SIZE(mes), img, 0, total.images.1), blot.query()
: -- obtain.image.number

PROC load.image
PROC load.image(CHAN from, to, INT image.number, BOOL success, INT info.layer)
... messages(mes1 to mes8)
... VAL good IS 0 : VAL bad IS -1 : VAL fileid IS INT 0 :
... (10) INT attrs : -- attributes, (20) BYTE fileid : [max.record.size] BYTE buf :
INT pt, len, location, rows, cols, outcome, fold.number, result, mes2layer :
... BYTE acknowledge : BOOL ok, open.ok, file.ok, fold.ok, close.ok, same :
insert.query(mes1) -- details in, read.string(fileid)
IF -- allow for possibility that this image already has a name
image.name.set[image.number]
... same := TRUE
SEQ pt = 1 FOR INT(fileid[0])
... IF image.name[image.number][pt] = fileid[pt] : SKIP
... otherwise : same := FALSE
... IF same : insert.info("Image renamed", info.layer)
... otherwise : SKIP
... otherwise : SKIP
... SEQ pt = 0 FOR ((INT(fileid[0])+1) : image.name[image.number][pt] := fileid[pt]
... image.name.set[image.number] :=TRUE, blot.query(), insert.info(mes1, info.layer)
... SEQ c = 1 FOR (INT fileid[0]) : char.attr(fileid[c], attr.info)
... mes2layer := info.layer, insert.info(mes2, info.layer)
open.fold(from, to, fold.number, outcome, info.layer)
... IF outcome = fi.ok -- file open, read, and close
... fold.ok:=TRUE, file.fold(from, to, uf.attach.file, fileid, fold.number, outcome)
... IF outcome <> fi.ok : file.ok := FALSE, insert.info(mes3, info.layer)
... otherwise -- ok - open and read
... file.ok := TRUE
... open.stream(from, to, uf.open.data.read, fold.number, outcome)
... IF outcome <> fi.ok : open.ok := FALSE, insert.info(mes4, info.layer)
... otherwise
... proceed to read file (See Page 273)

... otherwise : fold.ok := FALSE, insert.info(mes7, info.layer), pause.query()
... IF file.ok AND (open.ok AND close.ok) -- unfile and delete fold
send.command(from.uf[0], to.uf[0], uf.unfile, fold.number, result)
... otherwise : result := bad
... IF result = good -- delete contents
send.command(from.uf[0], to.uf[0], uf.delete.contents, fold.number, result)
... otherwise : SKIP
... IF result = good -- delete fold
send.command(from.uf[0], to.uf[0], uf.delete.fold, fold.number, result)
... otherwise : SKIP
... IF result = good : success := TRUE
otherwise -- inform of any problems
... success:=FALSE, insert.mes.integer(mes8, result, info.layer), pause.query()
blot.info.line(mes2layer)
: -- load.image

Proceed to process the file

... [max.record.size] BYTE buf : BOOL started, failure, yes :
INT image.index, file.index, buf.row, buf.col :
INT row.num, col.num, file.rows, file.cols :
273
BYTE rows.low, rows.high, cols.low, cols.high, tag :

... initialize variables

... tag := fsd.startstream, outcome := fi.ok

WHILE (tag <> fsd.endstream) AND (outcome = fi.ok) AND (NOT failure)

... proceed to read the file (See Page 274)

close.stream(from, to, outcome)

... IF outcome = fi.ok : close.ok := TRUE

... otherwise : close.ok := FALSE, insert.info(mes6, info.layer)

Proceed to read the file

... to ! fsc.read, get.stream.data(from, tag, buf, outcome)

... IF tag = fsd.record

... IF NOT started -- first record - find number of rows and columns

... started := TRUE

... find number of rows and columns in the file image

image.rows[image.number] := file.rows

image.cols[image.number] := file.cols

otherwise -- read pixel values from hex file and place in buffer

... VAL len IS (INT buf[0]) : VAL len.2 IS len/2 : INT buf.pnt :

SEQ pix.pnt = 0 FOR len.2

... IF fatal : SKIP

... otherwise

... buf.pnt := (2*pix.pnt)+1 -- hex to binary and to buffer

pix.as.binary([buf FROM buf.pnt FOR 2],

image.buffer[buf.row][buf.col])

buf.col := buf.col + 1

... IF buf.col >= file.cols : buf.col := 0, buf.row := buf.row+1

... otherwise : SKIP

... tag = fsd.endstream : SKIP

... otherwise : insert.info(mes5, info.layer)

PROC save.image

PROC save.image(CHAN from, to, VAL INT img, BOOL ok)

[10] INT attrs: -- attributes

[20] BYTE fileid: [max.record.size] BYTE buf :

... INT new.fold, len, location, rows, cols, result, lay: BYTE acknowledge :

... BOOL ok, open.ok, file.ok, fold.ok, close.ok: VAL filer IS INT 0 :

... PROC create.image.stream (See Page 275)

... provide heading and obtain file name

... attrs[0,1,2,3] := attr.size, ft.opstext, fc.source.text, 0 -- fold attributes

... result := fi.ok, create.fold(from, to, new.fold, attrs, result)

... IF result = fi.ok -- file fold for file writing

... make.filed(from, to, uf.make.filed, new.fold, fileid, result)

274
... IF result <> fi.ok  
  ... file.ok := FALSE, blot.info.line(lay)  
  ... insert.info("Unable to file fold ", lay), delay(short.pause)  
  otherwise  
  ... open.stream(from, to, uf.open.data.write, new.fold, result)  
  ... IF result <> fi.ok  
  ... file.ok := FALSE, blot.info.line(lay)  
  ... insert.info("Unable to write to file", lay), delay(short.pause)  
  otherwise  
  -- ok - proceed  
  create.image.stream(from, to, img, result, lay)  
  ... IF result = fi.ok : SKIP  
  otherwise  
  ... blot.info.line(lay)  
  ... insert.info("Unable to create stream ", lay), delay(short.pause)  
  otherwise  
  ... file.ok := FALSE, blot.info.line(lay)  
  ... insert.info("Unable to open fold", lay), delay(short.pause)  

: -- save.image

PROC create.image.stream

PROC create.image.stream(CHAN from.stream,to.stream,VAL INT img,INT result,info.layer)

... [max.record.size]BYTE data.buffer : BOOL started, finished, premature.end :
... BYTE tag : INT these.rows, these.cols, r, data.buffer.pnt :

... PROC complete.this.record (See Page 276)

... initialize
  from.stream ? tag
  WHILE tag <> fsc.close
  ... IF NOT started
  ... set up header
  ... put.stream.data(to.stream, fsd.record, data.buffer)
  ... otherlise
  -- write the body of the file
  ... IF r = these.rows
  ... put.stream.endstream(to.stream), from.stream ? tag
  ... otherlise
  ... complete.this.record(data.buffer.pnt-2,info.layer)
  otherwise
  -- write next record
  ... data.buffer.pnt := 1 -- this is for the output character stream
  ... pixels.per.record IS 64 : -- to fit into a TDS file record
  ... bytes.per.record IS 2*pixels.per.record :
  ... SEQ c = 0 FOR these.cols
  ... pix.as.hex(image.buffer[r][c],
  ... [data.buffer FROM data.buffer.pnt FOR 2])
  data.buffer.pnt := data.buffer.pnt + 2
  ... IF ((data.buffer.pnt REM bytes.per.record)=1)OR
  
  275
data.buffer.pnt:=1
  otherwise : SKIP
r := r + 1  -- next row
  IF tag = fsc.close : from.stream? result, put.stream.result(to.stream, fi.ok)
  otherwise SKIP
: -- create.image.stream

PROC complete.this.record

PROC complete.this.record(V A L INT length, INT lay)
  ... IF length > 255 : insert.mes.integer("Length too great, viz. ", length, lay)
  ... otherwise data.buffer[0] := (BYTE length)
  ... IF tag <> fsc.close
  ... put.stream.data(to.stream, fsd.record, data.buffer), from.stream? tag
  ... otherwise : premature.end := TRUE, r := these.rows
: -- complete.this.record

PROC get.one.image

PROC get.one.image(INT image.i.d, info.layer)
  ... VAL mes IS "Image number = " : VAL len IS INT(image.name[image.i.d][0]) :
  ... obtain.image.number(mes, image.i.d), insert.mes.integer(mes, image.i.d, info.layer)
  ... IF image.name.set[image.i.d]
    insert.info("Name: ", info.layer)
  ... IF len = 0 : write.attr("<null>", attr.info)
    otherwise
    SEQ c = 1 FOR len char.attr(image.name[image.i.d][c], attr.info)
  ... otherwise SKIP
: -- get.one.image

PROC get.two.images

PROC get.two.images(V A L INT destination, INT image.one, image.two, info.layer)
  ... VAL mesd IS "Destination image = " : VAL mess IS "Source image = " :
  ... IF destination = 1  -- destination first, then source
    obtain.image.number(mesd,image.one)
    insert.mes.integer(mesd,image.one,info.layer)
    ... supply a name if one exists
    obtain.image.number(mess, image.two)
    insert.mes.integer(mess, image.two, info.layer)
    ... supply a name if one exists
    otherwise
    ... source first, then destination
: -- get.two.images

PROC do.display

PROC do.display(V A L INT img, rows, cols, INT inf.lay)
-- Use of graphics
PRNG random(INT Ran)
  ... VAL A IS 1664525 : INT X :
  ... LONGPROD(X, Ran, A, Ran, 1), Ran := Ran >> 1 -- So that Ran is positive :
  ... VAL to.mixed IS BYTE #F6 : VAL B0 IS BYTE 0 : VAL B199 IS BYTE 199 :
        INT place, scaled, base.colour, counter, diff, scale.factor, offset :
  ... BOOL satisfactory, default, negative, show.segments: BYTE pix, ch; [256]BYTE str :
        insert.info("Displaying an Image", inf.lay), place := 123456, random(place)
        y.n.query.and.update.info("Use default mapping values", default, inf.lay)
  ... IF default
        ... scale.factor := 200, offset := -20 -- these give good results
        otherwise
        ... query.and.update.info("Scale factor(NORM=100) = ", scale.factor,1,1000,inf.lay)
        query.and.update.info("Offset = ", offset, -1000, 1000, inf.lay)
        y.n.query.and.update.info("Display segment boundaries", show.segments,inf.lay)
  ... send(to.mixed,B0,B0,B0,B0,B199,B0,B0,B199,B0) -- set up for mixed graphics
  ... SEQ sc.page = 0 FOR 4 erased[sc.page] := TRUE
  SEQ x = 0 FOR rows
  ... VAL cols.1 IS cols-1 : VAL rows.1 IS rows-1 : VAL cols.2 IS cols/2 :
        [2][200]INT line.buf :
  ... SEQ c = 0 FOR cols -- fill a row
        ... VAL c2 IS 2*c : VAL half IS (MOSTPOS INT)/2 : INT rr, cc :
  ... pixel := image.buffer[(rows-1)-r][c]
        ... IF show.segments AND (((r\segment.rows) = 0) OR
        ((c\segment.cols) = 0)) OR ((r = rows.1) OR (c = cols.1))
          pix := BYTE(255-(INT pix»-- bit-wise NOT of pixel
        ... otherwise SKIP
            scaled := (((INT pix)+offset) TIMES scale.factor) TIMES 12)/25600
            guard 'scaled' against overflow
            ... base.colour := scaled/4, diff := scaled - (4*base.colour)
            ... counter := 4, random(place) -- fill 2x2 box
            ... rr := place/half, random(place), cc := place/half
            WHILE counter > 0
              ... IF counter > diff : line.buf[rr][cc+c2] := base.colour
              ... otherwise line.buf[rr][cc+c2] := base.colour+1
              ... counter := counter - 1
              ... IF cc = 0 : cc := 1
              ... otherwise cc := 0
              ... IF counter = 2
              ... IF rr = 0 : rr := 1
              ... otherwise rr := 0
              ... otherwise SKIP
              ... SEQ rr = 0 FOR 2 SEQ cc = 1 FOR cols.2 -- send to screen
            INT code :
            ... -- pixels are packed into code, four per byte
            ... code := 0
            SEQ code.pt = 1 FOR 4 -- first pixel is at least-significant
            code := (code << 2) + line.buf[rr][(4*cc)-code.pt] -- part
            send(BYTE code) -- to graphics screen
  ... IF profile -- draw profile
        INT prev.r, prev.c, this.r, this.c :
        ... prev.r := 150 - (((INT(image.buffer[line][0])))/3)
        ... prev.c := 210, plot.point.w(prev.r, prev.c, B3) -- start off the profile

277
SEQ cc = 0 FOR 100  -- scale and place
... this.r := 150 - ((INT(image.buffer[1][cc][0])/3), this.c := cc + 210
join.points.w(prev.r, prev.c, this.r, this.c, B3),
... prev.r := this.r, prev.c := this.c
... otherwise SKIP
... TIMER hold: BYTE ch : BOOL cleared := INT dummy, now :
... hold ? now, cleared := FALSE  -- terminate graphics
WHILE NOT cleared  -- clear any existing keystrokes from the buffer
ALT
... keyboard ? dummy : hold ? now
... hold ? AFTER now PLUS 1000 : cleared := TRUE
... keyboard ? ch  -- wait, plot.point(999, 999, B0)
: -- do.display

PROC do.contour

PROC do.contour(VAL INT img, INT inf.lay) -- Contour drawing of object
PROC obtain(INT return)
... INT pt : BOOL satisfactory, neg :
... satisfactory := FALSE
... WHILE NOT satisfactory : read.string(str), pt := 1
extract.number.s(str, return, pt, neg, satisfactory)
: -- obtain
... BOOL default, coloured: INT colour,r,r.step,x.incr,y.incr,xO,yO,xa,ya,xb,yb:
insert.info(IIContour Display of an Image", inf.lay)
y.n.query.and.update.info("Use default contouring values ", default, inf.lay)
... IF
... default : r.step := 10  -- row step size in image
x.incr := 5  -- screen increments from one contour
y.incr := 10  -- line to next
coloured := TRUE
... otherwise : obtain values from operator and update info
send(to.low.res)
... SEQ scr = 0 FOR 320 top.shade[scr] := 199  -- set up covering shade
... initialize - global variables
WHILE r < 100
... ya := y0 - ((INT image.buffer[r][0])/4), xa := x0  -- init. this contour
... SEQ scr = 0 FOR 320 : temp.top.shade[scr] := 199  -- = bottom of screen
SEQ cc = 0 FOR 100  -- do this contour
... xb := x0 + (2*cc), yb := y0 - ((INT image.buffer[r][cc])/4)
... join.points.shade(ya, xa, yb, xb, BYTE colour), xa := xb, ya := yb
... SEQ scr = 0 FOR 320 -- update the shade INT temp :
... temp := temp.top.shade[scr]
... IF temp < 199 : top.shade[scr] := temp
... otherwise SKIP  -- it hasn't been updated
... r := r+r.step, x0 := x0+x.incr, y0 := y0-y.incr  -- ready for next contour
colour := ((colour+1) REM 4) + ((colour+1)/4)  -- = 1, 2, 3
... keyboard ? ch  -- wait, plot.point(999, 999, B0)  -- terminate graphics
: -- do.contour

278
E.1.7 Distribution of Images Over the Network

... PROC image.to.network (See Page 279)
... PROC image.from.network (See Page 279)

PROC image.to.network

PROC image.to.network(VAL INT img.num, INT lay)
    ... VAL layO IS lay : VAL total.rows IS (segment.rows+(2*border)) :
    VAL total.cols IS (segment.cols+(2*border)) :
    VAL count IS (total.rows*total.cols)+4 :
    VAL send.mes IS "Sending to processor " :
    VAL send.len IS SIZE(send.mes) :
        ... BOOL success, tto : INT proc, source.row, source.col, proc.lay :
        ... success := TRUE -- so far, proc.lay := layO, insert.info(send.mes, proc.lay)
        ... SEQ rr = 0 FOR no.of.rows SEQ cc = 0 FOR no.of.cols
        VAL start.row IS ((r*segment.rows)-border) :
        VAL start.col IS ((c*segment.cols)-border) :
        ... proc := (r*no.of.cols) + c -- target processor number
        position((info.t.row-2)+proc.lay, (info.t.col+2)+send.len)
        print.number.attr(proc, i, attr.info, FALSE)
        to.network ! proc; count; ha.load.image; img.num; total.rows; total.cols
        ... SEQ rr = 0 FOR total.rows SEQ cc = 0 FOR total.cols
        ... source.row := start.row + rr, source.col := start.col + cc
        ... IF ((source.row < 0) OR (source.row > image.buffer.size.1)) OR
            ((source.col < 0) OR (source.col > image.buffer.size.1))
        to.network ! 0 -- not on image : send 0
        ... otherwise to.network ! INT image.buffer[source.row][source.col]
    to.network ! end.of.message
    VAL ut IS unlimited.time : -- message from network
    ... INT dest.proc, num, tag, img, from, eos : BOOL escaping :
    ... escaping := FALSE, tto := FALSE, dest.proc := end.of.message
    WHILE dest.proc = end.of.message
        input.int(dest.proc, ut, success, escaping, tto)
        ... IF NOT escaping
            ... network response : input num, tag, img, from
            ... otherwise insert.info("NO RESPONSE ", lay).
        ... inform user whether success or failure
        : -- image.to.network

PROC image.from.network

PROC image.from.network(VAL INT img.num, INT lay, BOOL failure)
    ... VAL ut IS unlimited.time : VAL layO IS lay :
    VAL total.rows IS (segment.rows+(2*border)) :
    VAL total.cols IS (segment.cols+(2*border)) :
    VAL receive.mes IS "Receiving from processor ":
    VAL receive.len IS SIZE(receive.mes):
    INT proc, r, c, rr, cc, pixel, source.row, source.col, network.count :
    INT dest.proc, num, tag, img, from, pix.count, proc.lay :
    BOOL escaping, success, receiving, tto :

279
success := TRUE, proc.lay := lay0, insert.info(receive.mes, proc.lay)

SEQ r = 0 FOR no.of.rows
SEQ c = 0 FOR no.of.cols

VAL start.row IS (r*segment.rows): VAL stop.row IS (((r+1)*segment.rows)-1):

VAL start.col IS (c*segment.cols): VAL stop.col IS (((c+1)*segment.cols)-1):

IF success

proc := (r*no.of.cols) + c -- target processor number, lay := lay0
to.network ! proc; 2; hn.send.image; img.num; end.of.message
position((info.t.row-2)+proc.lay, (info.l.col+2)+receive.len)
print.number.attr(proc, 1, attr.info, FALSE)

... read message from network (See Page 280

... otherwise SKIP

: -- image.from.network

Read message from network

... escaping := FALSE, tto := FALSE -- message from network
... input.int(dest.proc, ut, success, escaping, tto) -- source identification
... any response?
... input num, tag, img, from, pix.count
... check tag : failure := TRUE if wrong tag

... IF failure : SKIP
otherwise

-- receive image segment

SEQ rr = -border FOR total.rows SEQ cc = -border FOR total.cols INT pix :

... source.row := start.row + rr, source.col := start.col + cc
input.int(pix, ut, success, escaping, tto)

... IF (((source.row < start.row) OR (source.row > stop.row)) OR
  ((source.col < start.col) OR (source.col > stop.col))) OR (NOT success)
SKIP -- this is border

... otherwise image.buffer[source.row][source.col] := BYTE pix

E.1.8 Miscellaneous Procedures

... PROC choose (See Page 281)
... PROC for.help()
... PROC for.exploration (See Page 281)
... PROC for.options (See Page 281)
... PROC for.information (See Page 283)
... PROC for.miscellaneous (See Page 284)
... PROC link.flush (See Page 284)
... PROC process.statistics (See Page 285)
... PROC display.statistics (See Page 285)
... PROC process.details (See Page 285)
... PROC test.response (See Page 286)
... PROC display.log(VAL INT from)
... PROC check.replies (See Page 286)
... PROC service.network (See Page 287)
... PROC network.response (See Page 287)

280
PROC choose

PROC choose(BYTE this.choice)
BYTE ch :
... SEQ noecho(ch)
... IF ch = ’?’ : this.choice := ch -- leave as is
... ch > ’Z’ : this.choice := BYTE((INT ch) - #20) -- convert to uppercase
... otherwise : this.choice := ch
: -- choose

PROC exploration

PROC exploration() -- Set up 'custom' instructions for network
... INT info.layer, dest, number, returned : [200]INT value :
BOOL continuing, exiting, timing.out :
... provide heading and general information
query.and.update.info("Destination = ",dest,MOSTNEG INT,MOSTPOS INT,info.layer)
... IF dest = (-1) : dest := broadcast
... dest = (-2) : dest := end.proc
... otherwise : SKIP
query.and.update.info("Number of values = ", number, 1, 199, info.layer) -- values in
query.and.update.info("Tag = ", value[1], MOSTNEG INT, MOSTPOS INT, info.layer)
... IF number > 1
SEQ numb = 2 FOR number-1
query.and.update.info("Value = ", value[numb], MOSTNEG INT, MOSTPOS INT, info.layer)
... otherwise : SKIP
y.n.query.and.update.info("Proceed ", continuing, info.layer)
... IF continuing -- optionally proceed
... insert.info("Sending instruction to network", info.layer)
to.network ! dest; number -- send to network
... SEQ numb = 1 FOR number to.network ! value[numb]
to.network ! end.of.message
insert.info("Type *’escape*’ when finished", info.layer)
... continuing := TRUE, exiting := FALSE, timing.out := FALSE
WHILE continuing
... input.int(returned, unlimited.time, continuing, exiting, timing.out)
... IF continuing
... interpret incoming message if appropriate
... otherwise SKIP
... otherwise SKIP
: -- exploration

PROC for.options

PROC for.options() -- Fix option(s)
... messages
... BOOL it.update, change : BYTE choice :
INT info.layer, bell.layer, details.layer, fast.layer, iterations.layer :
INT progress.layer, value.layer, fast.constant.layer, constant.layer :
... PROC show.details (See Page 283)
set up workpage and provide current status
change := TRUE
WHILE change -- assess choices
... insert.query("Choice : "), choose(choice)
... IF choice = 'B' -- bell
... toggle bell
choice = 'C' -- fast constant value
... info.layer := constant.layer
... IF fast.constant.set
... query.and.update.info(mes.fast.constant, fast.constant, 0, 255, info.layer)
to.network ! broadcast; 2 :hn.set.subtract.constant;
fast.constant; end.of.message
... otherwise SKIP
choice = 'D'
... toggle details flag
... details -- set detail flags
... SEU pro = 0 FOR no.in.network
... IF proc.detailes[pro]
to.network ! pro; 1; hn.set.details; end.of.message
... otherwise SKIP
... insert.info(mes.details.set, info.layer)
show.details(details.layer+1)
info.layer := info.layer+2
insert.info(mes.details.unset, info.layer)
otherwise -- clear detail flags
... to.network ! broadcast; 1; hn.clear.details; end.of.message
insert.info(mes.details.clear, info.layer)
... blot.info.line(details.layer+1, +2, +3)
choice = 'E' -- obtain proc ID for setting
... VAL mes IS "Processor Number = ": INT pro :
... insert.query(mes), input.integer(SIZE(mes), pro, 0, no.in.network-1)
... blot.query(), proc.details[pro] := TRUE
to.network ! pro; 1; hn.set.details; end.of.message
show.details(details.layer+1)
choice = 'U' -- obtain proc ID for clearing
... VAL mes IS "Processor Number = ": INT pro :
... insert.query(mes), input.integer(SIZE(mes), pro, 0, no.in.network-1)
... blot.query(), proc.details[pro] := FALSE
to.network ! pro; 1; hn.clear.details; end.of.message
show.details(details.layer+1)
choice = 'F' -- fast threshold
... fast.threshold.set := NOT fast.threshold.set
... blot.info.line(fast.layer), blot.info.line(fast.layer+1)
info.layer := fast.layer
... IF fast.threshold.set
... insert.info(mes.fast.on, info.layer)
fast.threshold.value := 0
insert.mes.integer(mes.update.thres, fast.threshold.value, info.layer)
otherwise insert.info(mes.fast.off, info.layer)
choice = 'I' -- iterations
... info.layer := iterations.layer
query.and.update.info(mes.iterations, iterations, 1, MOSTPOS INT, info.layer)
to.network ! broadcast; 2; hn.iterations; iterations; end.of.message

282
choice = 'P'  -- progress reports
  ... report.progress := NOT report.progress, blot.info.line(progress.layer)
  info.layer := progress.layer
  ... IF report.progress : insert.info(mes.progress.on, info.layer)
  ... otherwise insert.info(mes.progress.off, info.layer)
choice = 'S'  -- fast constant
  ... fast.constant.set := NOT fast.constant.set
  ... blot.info.line(constant.layer), blot.info.line(fast.constant.layer)
  info.layer := fast.constant.layer
  ... IF fast.constant.set
  ... fast.constant := 0
  ... insert.info(mes.fast.constant.on, info.layer)
  ... insert.mes.integer(mes.fast.constant.fast.constant, info.layer)
  ... otherwise insert.info(mes.fast.constant.off, info.layer)
choice = 'H'  -- fast threshold value
  ... info.layer := value.layer
  ... IF fast.threshold.set
  ... query.and.update.info(mes.update.thres, fast.threshold.value,
    0, 255, info.layer)
  to.network ! broadcast; 2; hn.set.threshold; fast.threshold.value;
  ... end.of.message
  ... otherwise SKIP
  ... choice = 'X' change := FALSE
  ... otherwise SKIP

: -- for.options

PROC show.details

PROC show.details(VAL INT lay)
  ... VAL no.in.network.2 IS no.in.network/2 : INT this.layer :
  ... this.layer := lay, insert.info(mes.details.for, this.layer)
  SEQ line = 0 FOR 2
  ... SEQ p = 0 FOR no.in.network.2, VAL pr IS (no.in.network.2*p+line)+p :
  ... IF proc.details[pr] : print.number.attr(pr, 2, attr.info, TRUE)
  ... otherwise write.attr("****", attr.info)
  write.attr("", attr.info)
  ... IF line = 0 : insert.info("", this.layer)
  ... otherwise SKIP

: -- show.details

PROC for.information

PROC for.information() -- Provide image information
  ... INT info.layer, named, image.number : BOOL change, same :
  ... change := TRUE
  WHILE change
  ... setup.workpage(), info.layer := 1
  ... insert.info(" TIPS image information", info.layer), named := 0
  SEQ im = 0 FOR total.images -- find number of named images
  ... IF image.name.set[im] : named := named+1
  ... otherwise SKIP

283
... IF named = 0
   insert.info("There are no named images on the system", info.layer)
   otherwise
   name them
   ... insert.mes.integer("Number of named images = ", named, info.layer)
   SEQ im = 0 FOR total.images
   ... IF image.name.set[im]
   ... provide name plus number of rows and columns
   ... otherwise SKIP
   ... y.n.query.and.update.info("Provide or change an image name", change, info.layer)
   ... IF change
   -- provide an image name
   ... VAL mesname IS ", Image name = ": [200]BYTE nameid :
   ... query.and.update.info("Image number = ", image.number, 0, total.images.1, info.layer)
   ... insert.query(mesname), read.string(nameid)
   -- allow for possibility that this image already has a name
   ... IF image.name.set[image.number]
   ... same := TRUE
   SEQ pt = 1 FOR INT(nameid[0])
   ... IF image.name[image.number][pt] = nameid[pt] : SKIP
   ... otherwise same := FALSE
   ... IF same : insert.info("Image renamed", info.layer)
   ... otherwise SKIP
   ... otherwise SKIP
   SEQ pt = 0 FOR ((INT(nameid[0]))+1)
   ... IF pt < max.name.length : image.name[image.number][pt] := nameid[pt]
   ... pt = max.name.length : insert.info("Name too long", info.layer)
   ... otherwise SKIP
   image.name.set[image.number] := TRUE
   ... otherwise SKIP

: -- for.information

PROC for.miscellaneous

PROC for.miscellaneous() -- Provide miscellaneous functions

   ... INT info.layer : BOOL change, inact :
   ... change := TRUE
   WHILE change
   ... set up workspace and list choices
   ... insert.query("Choice = ", choose(choice)
   ... IF choice = 'Z'
   ... zero the inactivity registers
   ... choice = 'R'
   ... read inactivity registers (here, count, tag, from, lov, high)
   ... choice = 'X' : change := FALSE
   ... otherwise SKIP

: -- for.miscellaneous

PROC link.flush

PROC link.flush(CHAN link) -- Remove any extraneous garbage from the input link

284
... TIMER local.clock: VAL tptr.l.ticks.per.second IS 15625: -- low priority
VAL ten.milliseconds IS tptr.l.ticks.per.second/100:
... BOOL proceeding: INT start: BYTE garbage:
... local.clock ? start, proceeding := TRUE
WHILE proceeding
ALT
... link ? garbage : SKIP
... local.clock ? AFTER start PLUS ten.milliseconds: proceeding := FALSE
: -- link.flush

PROC process.statistics

PROC process.statistics(INT source.proc)
... INT area, total, sum, max, min, rmax, cmx, rmin, cmn: BOOL cont, timing.out:
... cont := TRUE, timing.out := FALSE
... input values for total.image, area, total, source.proc, sum
-- IF cont AND (NOT (exiting OR timing.out))
... find total statistics
... find overall max and min
... otherwise null return
: -- process.statistics

PROC display.statistics

PROC display.statistics()
INT lay:
... blot.info(), lay := 1
... insert info on image number, area, total pixels, sum, mean, max, and min
-- (including locations)
pause.query()
: -- display.statistics

PROC process.details

PROC process.details(VAL INT ct, INT layer)
... INT num, d.type, value, from: BOOL cont, exiting, timing.out:
... cont := TRUE, exiting := FALSE, timing.out := FALSE
input.int(from, unlimited.time, cont, exiting, timing.out)
insert.msg.integer(“Network details from processor “, from, layer)
input.int(d.type, unlimited.time, cont, exiting, timing.out)
... IF d.type = d.num.dividers -- identify the detail type
insert.info(“Number of image dividers for rotation”, layer)
... d.type = d.divider : insert.info(“Divider location”, layer)
... d.type = d.source : insert.info(“Source processor”, layer)
d.type = d.row.col
insert.info(“Values of row, row.diff, col, col.diff”, layer)
d.type = d.request.send
insert.info(“Rotation request sent to processor”, layer)
... d.type = d.request.received : insert.info(“Rotation request received”, layer)

285
PROC test.response

PROC test.response()

PROC check.replies(VAL INT from, INT tag, inf.lay, BOOL finished, VAL BOOL disp)

Provide information

-- finished : write time to screen if displaying
INT whole, part :

... finished := TRUE, clock ? this.time, elapsed := this.time - last.time
... IF disp -- inform
... IF (iterations > 1)
  IF -- no iterations allowed
    ((tag = nh.finished.convex.hull) AND (NOT single.column)) OR
    ((tag = nh.finished.move.image) OR ((tag = nh.finished.rotate)))
  insert.mes.decimal("Time for function ", elapsed, tptr.l.ticks.per.second, 4, inf.lay)
  write.attr(" seconds ", attr.info)
otherwise
  ... insert.mes.decimal("Total time for function = ", elapsed, tptr.l.ticks.per.second, 4, inf.lay)
  write.attr(" seconds ", attr.info)
  elapsed := elapsed/iterations
  insert.mes.decimal("Net time for one iteration = ", elapsed, tptr.l.ticks.per.second, 4, inf.lay)
  write.attr(" seconds ", attr.info)
 otherwise -- no iterations
  ... insert.mes.decimal("Time for function ", elapsed, tptr.l.ticks.per.second, 4, inf.lay)
  write.attr(" seconds ", attr.info)
 pause.query()
 ... otherwise SKIP

PROC service.network

PROC service.network(INT source, tag, BOOL cont, exiting, timing.out, INT info.layer)
  ... BOOL an.error : INT count, mess, dest, im, im1, im2 :
  ... an.error := TRUE
  WHILE an.error
    ... an.error := FALSE
    dest := end.of.message -- may need to flush out e.o.m. -- obtain destination
    WHILE (dest = end.of.message) AND cont
      input.int(dest, unlimited.time, cont, exiting, timing.out)
      input.int(count, unlimited.time, cont, exiting, timing.out)
      ... IF cont
      ... input.int(tag, unlimited.time, cont, exiting, timing.out)
      ... IF cont -- process the message
      ... IF tag = nh.details -- network details
        process.details(count, info.layer)
      tag = nh.sending.statistics -- sending statistics
      process.statistics(source)
      ... process is determined by tag
      ... tag = nh.error : report.netvork.error(info.layer), an.error:=TRUE
      ... otherwise : an.error := TRUE
      ... otherwise SKIP
      ... otherwise SKIP
  : -- service.network

PROC network.response

PROC network.response(INT lay, BOOL successful, VAL BOOL display) -- Obtain full network response
  ... BOOL ex, wait.for.response, timeout : INT net.proc, tag :
  ... ex := FALSE, successful := FALSE, wait.for.response := TRUE, timeout := FALSE
  ... SEQ n.proc = 0 FOR full.network : replied[n.proc] := FALSE
  WHILE wait.for.response -- get response from all network processors
    ... service.network(net.proc, tag, wait.for.response, ex, timeout, lay)
    ... IF wait.for.response
    ... check.replies(net.proc, tag, lay, successful, display)
    ... IF successful : wait.for.response := FALSE
E.1.9 Images for the Network

```
... otherwise SKIP
... otherwise
... insert.info("Unable to get response from network", lay), pause.query()
: -- network.response

PROC for.add

PROC for.add()
  VAL mes IS "Truncate addition " :
  ... INT imagel, image2, sum, info.layer : BOOL truncate, ok :
  ... setup.workpage(),info.layer:=1,insert.info("Add Two Images ",info.layer)
  get.two.images(1,imagel,image2,info.layer)
  y.n.query.and.update.info(mes,truncate,info.layer)
  clock ? last.time -- set up clock
  to.network ! broadcast; 4; hn.add.images; imagel; image2;
  (INT truncate); end.of.message
  network.response(info.layer, ok, TRUE)
: -- for.add

PROC for.exit

PROC for.exit()
  ... BYTE ch : INT lay :
  ... setup.workpage(),notime := TRUE, lay := 1
  ... insert.info("EXIT requested", lay), insert.query("CONFIRM EXIT with 'Y' ")
```
noecho(ch)
... IF (ch <> 'Y') AND (ch <> 'y') : blot.query()
... otherwise exiting := TRUE  -- exit confirmed

PROC for.expand

PROC for.expand()
... BOOL ok : INT image1, image2, pixel, min, max, diff, info.layer :
... setup.workpage(), info.layer := 1, insert.info("Expand an Image ", info.layer)
get.two.images(2, image1, image2, info.layer)
image.name.set[image2] := TRUE  -- deal with image name
image.name[image2][0] := B0
image.rows[image2] := image.rows[image1]  -- destination image size
image.cols[image2] := image.cols[image1]
clock ? last.time  -- set up clock
... expand.max := 0, expand.min := 255  -- find max and min from network
to.network ! broadcast; 3; hn.request.max.min; image1; border; end.of.message
network.response(info.layer, ok, FALSE)
... IF ok  -- expand if max > min
... IF expand.max > expand.min  -- ok to proceed
to.network ! broadcast; 6; hn.expand.image; image1; image2; expand.max;
expand.min; border; end.of.message
network.response(info.layer, ok, TRUE)
otherwise
... insert.info("Uniform image - cannot be expanded", info.layer)
pause.query()
otherwise
... insert.info("Failed to obtain maximum and minimum values", info.layer)
pause.query()
: -- for.expand

PROC for.load

PROC for.load()
INT image.num, info.layer :
... setup.workpage(), info.layer := 1, insert.info("Loading an Image ", info.layer)
get.one.image(image.num, info.layer)
load.image(from.uf[0], to.uf[0], image.num, ok, info.layer)
... IF ok : image.to.network(image.num, info.layer)  -- load into host
... otherwise SKIP
: -- for.load

PROC for.logical

PROC for.logical()
VAL image.buffer.size.4 IS (image.buffer.size/4) :
... BYTE logic : BOOL ok : INT image1, image2, info.layer :
... setup.workpage(), info.layer := 1
insert.info("Logical Operations on an Image ", info.layer)
insert.query("Choose from: AND (A), OR (O), XOR (X), NOT (N) ")
echo(logic) -- find operator and convert to uppercase
... IF (INT logic) > (INT 'Z') : logic := BYTE((INT logic) - #20)
... otherwise SKIP
... IF ((logic = 'A') OR (logic = '0')) OR (logic = 'X')) OR (logic = 'N')
... insert.info("Logic function: ", info.layer) -- inform
... IF logic = 'A' : write.attr("AND", attr.info)
... logic = '0' : write.attr("OR", attr.info)
... logic = 'X' : write.attr("XOR", attr.info)
... logic = 'N' : write.attr("NOT", attr.info)
... IF logic = 'N' : get.one.image(image1, info.layer), image2 := 0 -- dummy
... otherwise get.two.images(2, image1, image2, info.layer)
clock ? last.time -- set up clock
to.network ! broadcast; 4; hn.logical; image1; image2; INT(logic); end.of.message
network.response(info.layer, ok, TRUE)
... otherwise insert.info("Invalid operator", info.layer)
:
-- for.logical

PROC for.move

PROC for.move()

... VAL mes1 IS "Row shift = " : VAL mes2 IS "Column shift = ": BOOL ok :
INT image1, image2, image3, r.shift, c.shift, info.layer :
INT repeats, segment.rows, segment.cols, abs.r.shift, abs.c.shift :
INT multiple.rows, multiple.cols, multiple, shift.rows, shift.cols :
... setup.workpage(), info.layer := 1, insert.info("Move an Image ", info.layer)
get.two.images(2, image1, image2, info.layer)
query.and.update.info(mes1, r.shift, -100, 100, info.layer)
query.and.update.info(mes2, c.shift, -100, 100, info.layer)
... image.name.set[image2] := TRUE, image.name[image2][0] := BO
image.rows[image2] := image.rows[image1]
image.cols[image2] := image.cols[image1]
-- allow for shifts broken into a multiplicity of shifts
segment.rows := image.rows[image1]/no.of.rows
segment.cols := image.cols[image1]/no.of.cols
... abs.r.shift := r.shift, abs(abs.r.shift) -- unsigned shift
... abs.c.shift := c.shift, abs(abs.c.shift) -- unsigned shift
multiple.rows := abs.r.shift/segment.rows
multiple.cols := abs.c.shift/segment.cols
... IF multiple.rows > multiple.cols : multiple := multiple.rows
... otherwise : multiple := multiple.cols
... IF multiple > 0
... shift.rows := r.shift/(multiple+1), shift.cols := c.shift/(multiple+1)
... otherwise SKIP
... IF multiple > 0 -- use copies and multiple shifts
... insert.info("Multiple moves involved - spare image required", info.layer)
get.one.image(image3, info.layer)
... IF image.name.set[image3] : SKIP
... otherwise : image.name.set[image3] := TRUE, image.name[image3][0] := BO
... set up destination image size
clock ? last.time -- set up clock
to.network ! broadcast; 3; hn.copy.image; image1; image2; INT(logic); end.of.message
network.response(info.layer, ok, FALSE)
SEQ mul = 1 FOR multiple
...
  to.network ! broadcast; 6; hn.move.image; image3; image2; shift.xrows; shift.xcols;
border; end.of.message -- send move instruction to network
network.response(info.layer, ok, FALSE)
-- form copy ready for next move
to.network ! broadcast; 3; hn.copy.image; image2; image3; end.of.message
network.response(info.layer, ok, FALSE)
  r.shift := r.shift - (multiple * shift.rows) -- calculate remainders
c.shift := c.shift - (multiple * shift.cols)
to.network ! broadcast; 6; hn.move.image; image3; image2; r.shift; c.shift;
border; end.of.message -- send final move to network
network.response(info.layer, ok, TRUE)
otherwise    -- single-move instruction
  clock ? last.time                                      -- set up clock
  to.network ! broadcast; 6; hn.move.image; image1; image2; r.shift; c.shift;
  border; end.of.message
network.response(info.layer, ok, TRUE)
: -- for.move

PROC for.purge

PROC for.purge()
...
  VAL mes TS "Time (seconds) = ": INT secs, info.layer :
...
  setup.workpage(), info.layer := 1
  insert.info("Purge the network using a timeout ", info.layer)
  query.and.update.info(mes, secs, 0, 10000, info.layer)
  to.network ! broadcast; 2; hn.purge; secs; end.of.message
: -- for.purge

PROC for.save

PROC for.save()
...
  INT image.num, info.layer : BOOL failed :
...
  setup.workpage(), info.layer := 1, insert.info("Saving an Image ", info.layer)
...
  get.one.image(image.num, info.layer), notime := TRUE
  image.from.netvork(image.num, info.layer, failed)
...
  IF failed : SKIP
...
  otherwise : save.image(from.uf[0], to.uf[0], image.num, ok)
: -- for.save

PROC for.statistics

PROC for.statistics()
...
  INT image, info.layer : BOOL ok :
...
  setup.workpage(), info.layer := 1
...
  insert.info("Image Statistics ", info.layer), get.one.image(image, info.layer)
clock ? last.time                                      -- set up clock
...
  initialise counters
to.network ! broadcast; 3; hn.statistics; image; border; end.of.message
...
  network.response(info.layer, ok, TRUE), display.statistics()
: -- for.statistics
PROC for.threshold

PROC for.threshold()

... VAL full IS BYTE 255 : VAL empty IS BYTE 0 : VAL mes IS "Threshold = " : 
VAL mes2 IS "Perform fast thresholding" :
... BOOL ok, fast : INT image, thres, info.layer :
... setup.workpage(), info.layer := 1 
insert.info("Thresholding an Image", info.layer), get.one.image(image, info.layer)
... IF fast.threshold.set : y.n.query.and.update.info(mes2, fast, info.layer)
... otherwise fast := FALSE 
... IF fast : fast.threshold.sum := 0 
... otherwise : query.and.update.info(mes, thres, 0, 255, info.layer)
clock ? last.time -- set up clock
... IF fast 
to.network ! broadcast; 3; hn.fast.threshold; image; border; end.of.message 
otherwise

to.network ! broadcast; 3; hn.threshold; image; thres; end.of.message
network.response(info.layer, ok, TRUE)
... IF fast 
... insert.mes.integer("Sum for network = ", fast.threshold.sum, info.layer)
pause.query()
... otherwise SKIP

-- for.threshold

E.2 Network Software

E.2.1 Program for the Network

PROC chordal(CHAN from.the.host, toward.host, toward.network, from.network, 
f.chord.in, f.chord.out, r.chord.in, r.chord.out, 
VAL proc, displacement, location.in.network)
VAL version.date IS [15, 11, 1988] : -- identify version for driver (day, month, year)
... VALUES (See Page 293) 
... variables (See Page 293) 
... general -- general (local) PROCs (See Page 297) 
... fast -- fast image-processing PROCs (See Page 298) 
... imageproc -- image processing PROCs (See Page 298) 
... specials -- PROCs employing special techniques (See Page 302) 
... internet -- internetwork PROCs (See Page 304) 
... hulling -- convex hull PROCs (See Page 305) 
... load/send -- PROCs for loading and sending images (See Page 309) 
... linkshell -- PROCs concerned with the link shell (See Page 310) 
... PROC input/output -- link shell (See Page 294) 
- invoke the shell appropriate to this processor's position in network

... IF first.transputer -- link equivalences for transputer adjacent to host
... net.out IS toward.network : net.in IS from.network :
... CHAN host.in, host.out : PLACE host.in AT 4 : PLACE host.out AT 0 :
input.output(host.in, net.in, f.chord.in, r.chord.out, host.out, 
net.out, f.chord.in, r.chord.out)

292
otherwise -- link equivalences for transputers not adjacent to host
... host.in IS from.the.host : host.out IS toward.host :
... net.out IS toward.network : net.in IS from.network :
input.output(host.in, net.in, f.chord.in, r.chord.in, host.out, 
  net.out, f.chord.out, r.chord.out)

: -- chordal

VALues

-- Errors, details flags, and tags are as defined for host software.
VAL to.down IS 0 : VAL to.host IS 0 : -- links
VAL to.up IS 1 : VAL to.net IS 1 :
VAL to.forward IS 2 : VAL to.reverse IS 3 :
VAL from.down IS 0 : VAL from.host IS 0 :
VAL from.up IS 1 : VAL from.net IS 1 :
VAL from.forward IS 2 : VAL from.reverse IS 3 :
VAL forever IS TRUE : VAL total.images IS 5 :
VAL image.buffer.rows IS 27 : VAL image.buffer.cols IS 102 :
VAL image.buffer.rows.1 IS image.buffer.rows - 1 :
VAL image.buffer.cols.1 IS image.buffer.cols - 1 :
VAL full.network IS 20 : VAL full.network.1 IS full.network - 1 :
VAL full.minus.displacement IS (full.network - displacement) :
VAL half.displacement IS (displacement+1)/2 :
VAL proc.ge.displacement IS BOOL (proc >= displacement) :
VAL last.transputer IS BOOL (location.in.network = 2) :
VAL first.transputer IS BOOL (location.in.network = 0) :
VAL host.proc IS MOSTNEG INT : VAL end.proc IS MOSTPOS INT :
VAL no.neighbour IS (MOSTNEG INT)/8 :
VAL LHS IS 0 : VAL RHS IS 1 : -- neighbours
VAL above IS 2 : VAL below IS 3 :
VAL packet.size.limit IS 15 : VAL max.tables IS 5 :
VAL max.table.size IS 2800 : VAL biggest.packet IS 1000000 :
VAL unspecified IS -1 : -- for packet sizes
VAL in.buf.size IS 1 : VAL merged.buf.size IS 1 : -- buffer sizes
VAL out.buf.size IS 40 :
VAL tptr.l.ticks.per.second IS 15625 : -- low priority process -- clocks
VAL tptr.h.ticks.per.second IS 1000000 : -- high priority process
VAL time.allowed IS tptr.h.ticks.per.second*1 :
VAL setup.delay IS tptr.h.ticks.per.second*1 :
VAL otherwise IS TRUE : VAL null IS 0 : -- miscellaneous
VAL end.of.message IS (MOSTPOS INT)/2 :
VAL broadcast IS (MOSTPOS INT)/4 : VAL unlimited.time IS 0 :
VAL row.buf.size IS 256 : VAL max.log IS 200 :
VAL processor IS proc : -- just another name
VAL slope.scale IS 65536 : -- for convex hull

Variables
VAL image.buf IS -3:
        VAL send.out.buf IS -4:
[4] [in.buf.size+1]CHAN OF INT in.buffer:
[4] [merged.buf.size+1]CHAN OF INT merged.buffer:
[4] [out.buf.size+1]CHAN OF INT out.buffer:
in.host.buffer IS in.buffer[0][0]: in.net.buffer IS in.buffer[1][0]:
into.merged.buffer IS merged.buffer[0]:
        -- Use of into.merged.buffer:
        -- either into.merged.buffer ! channel.id; tag [:values]
        -- or into.merged.buffer ! send.out.buf; channel.id; count [:values]
from.merged.buffer IS merged.buffer[merged.buf.size]:
down.buffer IS out.buffer[0][0]: up.buffer IS out.buffer[1][0]:
forward.buffer IS out.buffer[2][0]:
reverse.buffer IS out.buffer[3][0]:
        -- out.buffer[0,1,2,3][x] is on host, network, forward chordal, and reverse sides
[4] BOOL free.table:
        BOOL received:
[4] BOOL packet.waiting:
BOOL setup, single.column, purging, centre, timing.out, ok, erased, details:
[4] INT current.tag.in, count.this.tag.in:
[4] INT row.buffer.size INT row.buf:
        [full.network] INT packets.to.send:
[4] INT packets.source, packets.dest:
[4] INT neighbours:
        [image.buffer.cols] INT slope.scale.div.x:
[4] [max.tables+1] [max.table.size] INT tables:
        -- for tabled I/O
[4] [max.log] INT log.link, log.mes, log.time:
        -- input only
[256] INT pack:
        [full.network] INT route:
INT expected, host.now, net.now, next.in.queue, purging.now, purge.timeout:
INT number.in.network, repeat, seconds, timeout, top.of.queue:
INT next.packet, num.of.tables, iterations, iteration.count:
INT position.row, position.col, total.columns:
INT log.point, log.start.time, log.total, log, setup.now:
INT pt, line, image.num, last.time, this.time, elapsed, fold.num, result:
INT inactivity.low, inactivity.high, value:
[256] INT subc:
        [256] INT blank:
[256] INT threshold.settings:
        TIMER clock, delay.clock, purge.clock, log.clock:

E.2.2 Input-Output for the Link Shell

PROC input.output(CHAN h.in, n.in, f.in, r.in, h.out, n.out, f.out, r.out)
        ... PROC initialize (See Page 295)

initialize(h.in, n.in)

PRI PAR
PAR
        -- communications
        -- inputs from the network links
        ... WHILE TRUE: h.in ? value, in.buffer[0][0] ! value -- host side
        ... WHILE TRUE: n.in ? value, in.buffer[1][0] ! value -- network side
        ... WHILE TRUE: f.in ? value, in.buffer[2][0] ! value -- forward chordal
        ... WHILE TRUE: r.in ? value, in.buffer[3][0] ! value -- reverse chordal
        -- input buffers
        ... WHILE TRUE: input.buffer(0)
        ... WHILE TRUE: input.buffer(1)
        ... WHILE TRUE: input.buffer(2)

294
WHILE TRUE: input.buffer(3)

WHILE TRUE
    -- Merge all buffered inputs into a single (buffered) stream
    -- (subsequent processing is done by 'handle.input')
    INT input.value, dest.proc, numb:
    ALT
        in.buffer[0][in.buf.size] ? input.value  -- messages from host side of network
            logit(from.down, input.value)
            ... IF input.value <> end.of.message
            ... dest.proc := input.value, in.buffer[0][in.buf.size] ? numb
                    logit(from.down, numb)
                    accept.input(from.down, dest.proc, numb)
            ... otherwise : SKIP
        in.buffer[1][in.buf.size] ? input.value  -- from outer side of network
            ... as for link 0
        in.buffer[2][in.buf.size] ? input.value  -- from forward chords of network
            ... as for link 0
        in.buffer[3][in.buf.size] ? input.value  -- from reverse chords of network
            ... as for link 0
    -- output buffers
    ... WHILE TRUE : output.buffer(0, h.out)
    ... WHILE TRUE : output.buffer(1, n.out)
    ... WHILE TRUE : output.buffer(2, f.out)
    ... WHILE TRUE : output.buffer(3, r.out)
    -- merged buffer  -- processing occurs here
    ... WHILE TRUE : merge.buffer()
    ... measure inactive time

: -- input.output

Initialize PROC initialize(CHAN h.in, n.in)

... PROC set.up.route (See Page 295)

... details := FALSE, setup := TRUE, purging := FALSE, timing.out := FALSE
... SEQ link = 0 FOR 4 : packet.waiting[link] := FALSE
... SEQ tbl = 0 FOR max.tables : free.table[tbl] := TRUE
... num.of.tables := max.tables, iterations := 1, timeout := MOSTPOS INT
... expected := MOSTPOS INT, inactivity.low := 0, inactivity.high := 0
... SEQ blank.pt = 0 FOR 256 : blank[blank.pt] := 0
SEQ im = 0 FOR total.images
... image.rows[im] := image.buffer.rows, image.cols[im] := image.buffer.cols
SEQ x = 1 FOR image.buffer.cols-1  -- for convex hull
slope.scale.div.x[x] := slope.scale/x
... flush garbage out of input links
... link.flush(h.in), link.flush(n.in), link.flush(f.in), link.flush(r.in)
... log.point := 0, set.up.route(), log.start.time := MOSTPOS INT
    -- ensure that net time negative before log-clock is zeroed
: -- initialize

Set up route

295
PROC set.up.route()

... INT diff, result, ndiff, min, bestmove, compare, result : BOOL negative :
... [4]INT chordmoves, ringmoves :
SEQ destination = 0 FOR full.network
... diff := destination - proc, negative := (diff < 0) -- initialize
... IF negative : diff := -diff
... otherwise : SKIP
... min := MOSTPOS INT, ndiff := full.network - diff
-- set up options
... chordmoves[0] := diff/displacement, ringmoves[0] := diff/displacement
... chordmoves[1] := chordmoves[0] + 1, ringmoves[1] := displacement - ringmoves[0]
... otherwise : SKIP -- adjust for break between nodes 0 and (n-1)
... SEQ index = 0 FOR 4 INT sum : -- find minimum choice
... sum := chordmoves[index] + ringmoves[index]
... IF sum < min : min := sum, bestmove := index
... otherwise : SKIP
... IF bestmove >= 2 : negative := NOT negative -- adjust sign
... otherwise : SKIP
... IF (bestmove < 0) OR (bestmove > 3) : route[destination] := 0 -- to host
... diff := 0 : SKIP
chordmoves[bestmove] <> 0
... IF (ringmoves[bestmove] > 0) AND (chordmoves[bestmove] = 1)
... IF negative : result := proc - displacement -- find result
... otherwise : result := proc + displacement
... IF result < 0 : result := full.network + result
... result >= full.network, result := result - full.network
... otherwise : SKIP
compare := destination - result -- obtain absolute difference
... IF compare < 0 : compare := -compare
... otherwise : SKIP
... IF compare >= displacement -- ring
... IF (bestmove = 1) OR (bestmove = 3)
... negative := NOT negative -- reverse of chordal motion
... otherwise : SKIP
... IF negative : route[destination] := 0 -- to host
... otherwise : route[destination] := 1 -- to network
otherwise -- chordal
... IF negative : route[destination] := 3 -- reverse
... otherwise : route[destination] := 2 -- forward
otherwise -- chordal move
... IF negative : route[destination] := 3 -- reverse
... otherwise : route[destination] := 2 -- forward
otherwise -- ring move
... IF negative : route[destination] := 0 -- to host
... otherwise : route[destination] := 1 -- to network
: -- set.up.route

296
E.2.3 General (Local) Procedures

... PROC abs(INT signed) -- Return the absolute value of the parameter.
... PROC delay(VAL INT seconds)
... PROC logit(VAL INT this.link, this.value)
... PROC link.flush(CRAN link) -- Remove any extraneous garbage from the input link
... PROC clean.out(VAL INT input.id, this.tag, VAL BOOL send.message) -- clean up links
... PROC find.first.free.table(INT first.free.table, BOOL success)
... PROC handle.first.free.table(See Page 297)
... PROC handle.output.2(VAL INT destin, VAL INT v0, v1) - Interface to PROC handle.output.
... PROC handle.output.3(VAL INT destin, VAL INT v0, v1, v2)
... PROC handle.output.4(VAL INT destin, VAL INT v0, v1, v2, v3)
... other handle.output PROCs up to 15 individual inputs
... PROC check.image.number(INT image) -- check for validity

Handle output

PROC handle.output(VAL INT destin, numb, VAL []INT value)

-- This uses the best route to the specified destination.
-- The number of integers in value is given by numb.
-- To conform to the message protocol, value[0] should = total count for message.
INT dest :
...
IF first.transputer AND (destin < 0) -- to host
down.buffer ! host.proc
    ... SEQ num = 0 FOR numb : down.buffer ! value[num]
down.buffer ! end.of.message; end.buf
destin = broadcast -- send to next in this row and next in this column
...
IF (neighbours[above] = no.neighbour) AND (neighbours[RHS] <> no.neighbour)
-- send to processor to right of this one
VAL link IS route[neighbours[RHS]] :
out.link.buffer IS out.buffer[link][0] :
...
out.link.buffer ! destin
...
SEQ num = 0 FOR numb : out.link.buffer ! value[num]
out.link.buffer ! end.of.message; end.buf
... otherwise : SKIP
...
IF neighbours[below] <> no.neighbour -- send to proc. next row down
VAL link IS route[neighbours[below]] :
out.link.buffer IS out.buffer[link][0] :
...
out.link.buffer ! destin
...
SEQ num = 0 FOR numb : out.link.buffer ! value[num]
out.link.buffer ! end.of.message; end.buf
... otherwise : SKIP
otherwise -- to destination via best route -- adjust if end or host processor
...
IF destin = end.proc : dest := processor+1
...
destin < 0 : destin := 0
...
otherwise : destin := destin
...
VAL link IS route[dest] : out.link.buffer IS out.buffer[link][0] :
...
out.link.buffer ! destin
...
SEQ num = 0 FOR numb : out.link.buffer ! value[num]
out.link.buffer ! end.of.message; end.buf
: -- handle.output

297
E.2.4 Fast Image-Processing Procedures

... PROC for.fast.constant (See Page 298)
... PROC for.fast.threshold (See Page 298)
... PROC set.subtract.constant (VAL INT proc.num) -- set up constant for subtraction
... PROC set.threshold (VAL INT proc.num) -- set up threshold value

Fast Constant Subtraction

PROC for.fast.constant (VAL INT proc.num)
INT image, rows, cols :
... from merged.buffer ? image
... pass it on
check.image.number (image)
... rows := image.rows[image], cols := image.cols[image]
SEQ r = 0 FOR rows
[image.buffer.cols]INT im RETYPES image.buffer[image][r] :
... SEQ c = 0 FOR cols : im[c] := subc[im[c]]
handle.output.4 (host.proc, 3, nh.finished.constant, image, processor) -- finished
: -- for.fast.constant

Fast Threshold

PROC for.fast.threshold (VAL INT proc.num)
... VAL full IS INT 255 : VAL empty IS INT 0 :
INT image, pixel, sum, thres, rows, cols, rows.2b, cols.2b, border :
... from merged.buffer ? image; border
... pass it on
check.image.number (image)
... rows := image.rows[image], cols := image.cols[image]
... rows.2b := rows-(2*border), cols.2b := cols-(2*border), sum := 0
SEQ r = border FOR rows.2b
[image.buffer.cols]INT im RETYPES image.buffer[image][r] :
SEQ c = border FOR cols.2b
... pixel := im[c], thres := threshold.settings[pixel]
... im[c] := thres, sum := sum+thres
handle.output.5 (host.proc, 4, nh.finished.fast.threshold, image, processor, sum)
: -- for.fast.threshold

E.2.5 Image Processing Procedures

... PROC for.threshold (VAL INT proc.num)
... PROC for.expand (See Page 299)
... PROC send.max.min (See Page 299)
... PROC for.rotate (VAL INT proc.num)
... PROC add.images (VAL INT proc.num)
... PROC move.image (See Page 300)
... PROC copy.image (VAL INT proc.num)

298
Expand the Intensity Range of an Image

PROC for.expand(VAL INT proc.num)
   -- Establish output image size.
   INT image1, image2, pixel, min, max, pix, rows, cols, max.min, border, pixel:
   ... from.merged.buffer ? image1; image2; max; min; border
   ... pass it on
   max.min := max-min
   ... IF max.min < 1 : max.min := 1
   ... otherwise : SKIP
   ... check.image.number(image1), check.image.number(image2)
   ... rows := image.rows[image1], image.rows[image2] := rows
   ... cols := image.cols[image1], image.cols[image2] := cols
   VAL i.b.c IS image.buffer.cols :
   SEQ r = 0 FOR rows
      [i.b.c]INT im1 RETYPES [image.buffer[image1][r] FROM 0 FOR i.b.c] :
      [i.b.c]INT im2 RETYPES [image.buffer[image2][r] FROM 0 FOR i.b.c] :
      SEQ c = 0 FOR cols
         ... pix := im1[c], pixel := ((255*(pix-min))/max.min)
         ... IF pixel > 255 : im2[c] := 255 -- ensure against overflow
         ... pixel < 0 : im2[c] := 0
         ... otherwise : im2[c] := pixel
         handle.output.5(host.proc, 4, nh.finished.expand, image1, image2, processor)
   : -- for.expand

Send maximum and minimum pixel values

PROC send.max.min(VAL INT proc.num)
   INT image, pixel, min, max, rows, cols, border, rows.2b, cols.2b :
   ... from.merged.buffer ? image; border
   ... pass it on
   check.image.number(image)
      -- perform the search - only inside the borders
   ... rows := image.rows[image], cols := image.cols[image]
   ... rows.2b := rows-(2*border), cols.2b := cols-(2*border)
   ... max := 0, min := 255
   VAL i.b.c IS image.buffer.cols :
   SEQ r = border FOR rows.2b
      [i.b.c]INT im RETYPES [image.buffer[image][r] FROM 0 FOR i.b.c] :
      SEQ c = border FOR cols.2b
         ... pixel := im[c]
         ... IF pixel > max : max := pixel
         ... otherwise : SKIP
         ... IF pixel < min : min := pixel

299
... otherwise : SKIP
handle.output.6(host.proc, 5, nh.max.min, processor, image, max, min)
: -- send.max.min

Move an image

PROC move.image(VAL INT proc.num)
VAL null.source IS -1:
INT image1, image2, r.shift, c.shift, rows, cols, border, rows.1, cols.1 :
INT rb, rs, cb, cs, rb.rs, cb.cs, source.1, source.2, source.3 :
INT fr1, fr2, tr1, tr2, fr3, tr3, fc1, tc1, fc2, tc2, fc3, tc3, abs.r.shift, abs.c.shift :
... from.merged.buffer ? image1; image2; r.shift; c.shift; border
... IF (proc.num = broadcast) AND (NOT last.transputer) -- pass it on
handle.output.7(broadcast, 6, ha.move.image, image1, image2, r.shift, c.shift, border)
... otherwise : SKIP
... check.image.number(image1), check.image.number(image2)
... rows := image.rows[image1], image.rows[image2] := rows, rows.1 := rows - 1
... cols := image.cols[image1], image.cols[image2] := cols, cols.1 := cols - 1
... abs.r.shift := r.shift, abs(abs.r.shift), abs.c.shift := c.shift, abs(abs.c.shift)
... SEQ r = 0 FOR rows SEQ c = 0 FOR cols -- do home part first
... IF ((r < r.shift) OR (c < c.shift)) OR ((r > (rows.1 + r.shift)) OR
(c > (cols.1 + c.shift)))
image.buffer[image2][r][c] := 0
otherwise
... image.buffer[image2][r][c] := image.buffer[image1][r - r.shift][c - c.shift]
... rb := rows - border, cb := cols - border, rb.rs := rb - abs.r.shift
... rs.b := abs.r.shift + border, cb.cs := cb - abs.c.shift, cb.cs := abs.c.shift + border
... IF r.shift >= 0 -- set row sources and destinations
... fr1 := rb.rs, tr1 := border, fr2 := rb.rs, tr2 := border, fr3 := border, tr3 := rs.b
otherwise
... fr1 := border, tr1 := rb.rs, fr2 := border, tr2 := rb.rs, fr3 := border, tr3 := border
... IF c.shift >= 0 -- set column sources and destinations
... fc1 := border, tc1 := cs.b, fc2 := cb.cs, tc2 := border, fc3 := cb.cs, tc3 := border
otherwise
... fc1 := cs.b, tc1 := border, fc2 := border, tc2 := cb.cs, fc3 := border, tc3 := cb.cs
... set processor sources for the three image parts (See Page 301)

... IF r.shift = 0 -- column shift (possibly)
... IF c.shift = 0 : SKIP -- no inter-processor shift required
otherwise -- part 3 only
... IF source.3 = null.source : SKIP
otherwise
handle.output.12(source.3, 11, nn.request.image.segment, processor,
image1, fr3, fc3, rb.rs - border, abs.c.shift, border, image2, tr3, tc3)
otherwise
... row shift and possibly column shift
... IF c.shift = 0 -- part 1 only -- possibly obtain one part
... IF source.1 = null.source : SKIP
otherwise
handle.output.12(source.1, 11, nn.request.image.segment, processor,
image1, fr1, fc1, abs.r.shift, cb.cs - border, border, image2, tr1, tc1)
300
otherwise -- all 3 parts may be involved
otherwise
... IF source.1 = null.source : SKIP
otherwise
handle.output.12(source.1,11,nn.request.image.segment,processor, image1,fr1,fc1,abs.r.shift,cb.cs-border,border,image2,fr1,tc1)
... IF source.2 = null.source : SKIP
otherwise
handle.output.12(source.2,11,nn.request.image.segment,processor, image1,fr2,fc2,abs.r.shift,abs.c.shift,border,image2,fr2,tc2)
... IF source.3 = null.source : SKIP
otherwise
handle.output.12(source.3,11,nn.request.image.segment,processor, image1,fr3,fc3,rb.rs-border,abs.c.shift,border,image2,fr3,tc3)
handle.output.5(host.proc,4,nh.finished.move.image,image1,image2,processor) -- finished

-- move.image

Set processor sources for the three image parts

... IF (r.shift < 0) AND (c.shift >= 0) -- establish source processors
... IF neighbours[below] <> no.neighbour : source.1 := proc+total.columns
... otherwise : source.1 := null.source
... IF (neighbours[LHS] <> no.neighbour) AND (neighbours[below] <> no.neighbour)
source.2 := (proc-1)+total.columns
... otherwise : source.2 := null.source
... IF neighbours[LHS] <> no.neighbour : source.3 := proc-1
... otherwise : source.3 := null.source
(r.shift < 0) AND (c.shift < 0)
... IF neighbours[below] <> no.neighbour : source.1 := proc+total.columns
... otherwise : source.1 := null.source
... IF (neighbours[RHS] <> no.neighbour) AND (neighbours[below] <> no.neighbour)
source.2 := (proc+1)+total.columns
... otherwise : source.2 := null.source
... IF neighbours[RHS] <> no.neighbour : source.3 := proc+1
... otherwise : source.3 := null.source
(r.shift >= 0) AND (c.shift >= 0)
... IF neighbours[above] <> no.neighbour : source.1 := proc-total.columns
... otherwise : source.1 := null.source
... IF (neighbours[LHS] <> no.neighbour) AND (neighbours[above] <> no.neighbour)
source.2 := (proc-1)-total.columns
... otherwise : source.2 := null.source
... IF neighbours[LHS] <> no.neighbour : source.3 := proc-1
... otherwise : source.3 := null.source
otherwise -- (r.shift >= 0) AND (c.shift < 0)
... IF neighbours[above] <> no.neighbour : source.1 := proc-total.columns
... otherwise : source.1 := null.source
... IF (neighbours[RHS] <> no.neighbour) AND (neighbours[above] <> no.neighbour)
source.2 := (proc+1)-total.columns
... otherwise : source.2 := null.source
... IF neighbours[RHS] <> no.neighbour : source.3 := proc+1
... otherwise : source.3 := null.source
otherwise -- (r.shift > 0) AND (c.shift < 0)
... IF neighbours[above] <> no.neighbour : source.1 := proc-total.columns
... otherwise : source.1 := null.source
... IF (neighbours[RHS] <> no.neighbour) AND (neighbours[above] <> no.neighbour)
source.2 := (proc+1)-total.columns
... otherwise : source.2 := null.source
... IF neighbours[RHS] <> no.neighbour : source.3 := proc+1
... otherwise : source.3 := null.source
otherwise -- (r.shift > 0) AND (c.shift > 0)
... IF neighbours[above] <> no.neighbour : source.1 := proc-total.columns
... otherwise : source.1 := null.source
... IF (neighbours[RHS] <> no.neighbour) AND (neighbours[above] <> no.neighbour)
source.2 := (proc+1)-total.columns
... otherwise : source.2 := null.source
... IF neighbours[RHS] <> no.neighbour : source.3 := proc+1
... otherwise : source.3 := null.source
otherwise
Perform logical operations

301
PROC logical(VAL INT proc.num)
... BYTE logic : INT image1, image2, rows, cols, cols.4, log :
... from.merged.buffer ? image1; image2; log
... IF (proc.num = broadcast) AND (NOT last.transputer) -- pass on
handle.output.5(broadcast, 4, hn.logical, image1, image2, log)
... otherwise : SKIP
... check.image.number(image1), check.image.number(image2), logic := BYTE log
... IF logic = 'A' -- AND

... logical AND (See Page 302)

logic = 'O' -- OR
... logical OR
logic = 'X' -- XOR
... logical XOR
logic = 'N' -- NOT
... logical NOT
handle.output.5(host.proc, 4, nh.finished.logical, image1, image2, processor) :
-- logical

Logical AND

rows := image.rows[image1] -- set rows
... IF image.rows[image2] < rows : rows := image.rows[image2]
... otherwise : SKIP
cols := image.cols[image1] -- set cols
... IF image.cols[image2] < cols : cols := image.cols[image2]
... otherwise : SKIP
cols.4 := (cols+3)/4
VAL i.b.c IS image.buffer.cols :
SEQ r = 0 FOR rows
[i.b.c]INT im1 RETYPES [image.buffer[image1][r] FROM 0 FOR i.b.c] :
[i.b.c]INT im2 RETYPES [image.buffer[image2][r] FROM 0 FOR i.b.c] :
SEQ c = 0 FOR cols
im1[c] := im1[c]/im2[c]

E.2.6 Special Image-Processing Procedures

... PROC sub.image(VAL INT proc.num)
... PROC sub.constant(VAL INT proc.num)
... PROC rank.filter (See Page 302)

Rank filter

PROC rank.filter(VAL INT proc.num)
-- Adapted from Don Bailey's VIPS version.
... [266]INT hist : INT y, pos, left, image1, image2, rank, rows, rows.2, cols, cols.2 :
... PROC rank.adjust (See Page 303)

... from.merged.buffer ? image1; image2; rank
... pass on
... check.image.number(image1), check.image.number(image2)
... rows := image.rows[image1], image.rows[image2] := rows, rows.2 := rows - 2
... cols := image.cols[image1], image.cols[image2] := cols, cols.2 := cols - 2
... IF cols = 27

... handle 27 columns (See Page 303)

... cols = 22 : handle 22 columns
... cols = 12 : handle 12 columns
... cols = 7 : handle 7 columns
... otherwise : handle any number of columns (less efficient)
handle.output.5(host.proc, 4, nh.finished.rank.filter, image1, image2, processor)
: -- rank.filter

Rank adjust

PROC rank.adjust(INT position, left.of, rank)
... WHILE left.of < rank : left.of := left.of+hist[position], position := position+1
... WHILE left.of > rank : position := position-1, left.of := left.of - hist[position]
: -- rank.adjust

Handle 27 columns  This technique is used to avoid inefficiencies with the retyping.

... VAL i.b.c IS 27 :  VAL i.b.c.1 IS i.b.c-1 :  VAL i.b.c.2 IS i.b.c-2 :
SEQ i = 1 FOR rows.2 -- for each row
[i.b.c]INT im10 RETYPES [image.buffer[image1][i-1] FROM 0 FOR i.b.c] :
[i.b.c]INT im11 RETYPES [image.buffer[image1][i] FROM 0 FOR i.b.c] :
[i.b.c]INT im12 RETYPES [image.buffer[image1][i+1] FROM 0 FOR i.b.c] :
[i.b.c]INT im2 RETYPES [image.buffer[image2][i] FROM 0 FOR i.b.c] :
... SEQ x = 0 FOR 256 : hist[x] := 0 -- initialise the histogram
... pos := 128, left := 0, y := 1
... VAL this IS INT im10[y] :  -- y=1  -- initial loading of the histogram
... hist[this] := hist[this]+1
... IF this < pos : left := left+1; otherwise : SKIP
VAL this IS INT im11[y] :
... hist[this] := hist[this]+1
... IF this < pos : left := left+1; otherwise : SKIP
VAL this IS INT im12[y] :
... hist[this] := hist[this]+1
... IF this < pos : left := left+1; otherwise : SKIP
y := 0
... VAL this IS INT im10[y] :
... hist[this] := hist[this]+1
... IF this < pos : left := left +1; otherwise : SKIP
VAL this IS INT im11[y] :
... hist[this] := hist[this]+1
... IF this < pos : left := left+1; otherwise : SKIP
VAL this IS INT im12[y] :
... hist[this] := hist[this]+1
... IF this < pos : left := left+1; otherwise : SKIP
SEQ j = 1 FOR i.b.c.2 -- for each column
... VAL this IS INT im10[j+1] :
... hist[this] := hist[this]+1
... IF this < pos : left := left+1; otherwise : SKIP
VAL this IS INT im11[j+1] :
... hist[this] := hist[this]+1
... IF this < pos : left := left+1; otherwise : SKIP
VAL this IS INT im12[j+1] :
... hist[this] := hist[this]+1
... IF this < pos : left := left+1; otherwise : SKIP
rank.adjust(pos,left,rank)
im2[j] := pos
... VAL this IS INT im10[j-1] :
... hist[this] := hist[this]-1
... IF this < pos : left := left-1; otherwise : SKIP
VAL this IS INT im11[j-1] :
... hist[this] := hist[this]-1
... IF this < pos : left := left-1; otherwise : SKIP
VAL this IS INT im12[j-1] :
... hist[this] := hist[this]-1
... IF this < pos : left := left-1; otherwise : SKIP
... im2[0] := 0, im2[i.b.c.1] := 0

E.2.7 Internetwork Procedures

PROC send.image.segment (See Page 304)
PROC send.image.segment.rotate (VAL INT proc.num) -- Send image segment with rotation
PROC receive.image.segment (See Page 305)

Send image segment

PROC send.image.segment (VAL INT proc.num)
-- Send image segment, without waiting for acknowledgement
BOOL succ :
INT from, image1, image2, row0, col0, drow, dcol, torow, tocol, bord, first.free, point :
... from.merged.buffer ? from;image1;row0;col0;drow;dcol;bord;image2;torow;tocol
find.first.free.table(first.free, succ)
... IF succ -- place segment in table and send
... free.table[first.free] := FALSE, point := 0
... SEQ row = row0 FOR drow SEQ col = col0 FOR dcol
... [tables[first.free] FROM point FOR dcol] :=
[ [image.buffer[image1][row] FROM col0 FOR dcol]
point := point+dcol

304
handle.output.15(from, point+11, an.sending.image.segment, processor, image2, row0, col0, drow, dcol, bord, torow, tocol, point, table.buf, first.free, point)
otherwise -- no free tables
handle.output.5(host.proc, 4, nh.error, proc, no.free.tables, nn.request.image.segment)

: -- send.image.segment

Receive image segment

PROC receive.image.segment(VA INT proc.num)
-- Receive image segment without acknowledgement
INT source, image.out, r0, c0, drow, dcol, bord, row0, col0, numb, pix :
... from.merged.buffer ? source; image.out; r0; c0; drow; dcol; bord; row0; col0; numb
... SEQ row = row0 FOR drow  SEQ col = col0 FOR dcol
... from.merged.buffer ? pix, image.buffer[image.out][row][col] := pix

: -- receive.image.segment

E.2.8 Convex Hull Procedures

... PROC requesting.row (See Page 305)
... PROC do.hull (See Page 305)
... PROC receive.row (See Page 306)
... PROC for.hull (See Page 308)

Requesting.row

PROC requesting.row(VA INT proc.num)
INT image1, image2, row, col, pt, border, from, rows, cols, cols.2b, first.free :
BOOL succ :
... from.merged.buffer ? image1; image2; from; row; border
... rows := image.rows[image1], cols := image.cols[image1], cols.2b := cols-border
-- place in first free table
find.first.free.table(first.free, succ)
... IF succ
... free.table[first.free] := FALSE
[tables[first.free] FROM 0 FOR cols.2b] :=
[image.buffer[image1][row] FROM border FOR cols.2b]
otherwise -- no free tables
handle.output.5(host.proc, 4, nh.error, proc, no.free.tables, nn.request.row)
... IF succ
handle.output.12(from, 8+pt, mn.sending.row, image1, image2, processor,
position.col, row, border, pt, table.buf, first.free, pt)
... otherwise : SKIP -- can't proceed

: -- requesting.row

Do hull

305
PROC do.hull(VAL INT input.image, output.image, total.points, hull.row, border, VAL BOOL final.iteration)
  -- Perform hull on this row and place in output image for this segment, and onto links for other segments.
  ... VAL slope.scale IS 16 : VAL check.scale IS 255<<slope.scale :
  INT pix, pixel, col, end.col, x.diff, max.slope, y.diff, x, cols.2b, n.start :
  INT c.count, slope, int.diff, int.start, hull.pixel, points, first.free :
  ... [row.buffer.size]INT row.buffer.out : BOOL finished, succ :
  ... col := 1, cols.2b := image.cols[output.image]-(2*border)
  WHILE col <= total.points
    ... end.col := col, x.diff := 0, max.slope := MOSTNEG INT, y.diff := 0, x := col
      ... IF row.buffer[col] = 0 : finished := TRUE -- don’t proceed if zero pixel
        ... otherwise : finished := FALSE
      WHILE NOT finished -- determine max slope
        ... int.diff := row.buffer[x] - row.buffer[x-1]
          ... x.diff := x.diff + 1, y.diff := y.diff + int.diff
          ... slope := (y.diff<<slope.scale)/x.diff
          ... IF slope >= max.slope : max.slope := slope, end.col := x -- max slope
            ... otherwise : SKIP
          x := x + 1
          ... IF x = total.points : finished := TRUE -- check for finished
            ... row.buffer[x] = 0 : finished := TRUE
            ... otherwise : SKIP
        ... c.count := (end.col + 1) - col, int.start := row.buffer[col-1]
        SEQ c = col FOR c.count -- write hull
          ... hull.pixel := (int.start + (((c - col)*max.slope)>>slope.scale))
            ... IF (hull.pixel < 0) OR (row.buffer[c-1] = 0) : row.buffer.out[c-1] := 0
              ... hull.pixel > 255 : row.buffer.out[c-1] := 255
                ... otherwise : row.buffer.out[c-1] := hull.pixel
        col := end.col + 1
  points := cols.2b
  SEQ seg = 0 FOR total.columns -- send out results of hull
    VAL seg.proc IS (total.columns*position.col) : -- beginning of this row
      ... IF seg = position.col -- restore own segment
        ... n.start := position.col*points
          [image.buffer[output.image][hull.row] FROM border FOR points] :=
            [row.buffer.out FROM n.start FOR points]
        ... otherwise -- send row
          ... VAL seg.pt IS seg*points :
            ... find.first.free.table(first.free, succ)
              ... IF succ -- send
                ... free.table[first.free] := FALSE
                  ... SEQ pt = 0 FOR points
                    tables[first.free][pt] := row.buffer.out[pt+seg.pt]
                    handle.output.12(seg.proc+seg,8+points,nn.sending.row,input.image,
                      points,table.buffer,output.image,processor,seg,
                      hull.row,border,first.free,points)
              ... otherwise -- no free tables
                handle.output.5(host.proc,4,nh.error,proc,
                  no.free.tables,hn.convex.hull)
  : -- do.hull

Receive row

306
PROC receive.row(VAL INT proc.num)
   -- From nn.sending.row
BOOL finished, final.iteration :
INT pix, pixel, image1, image2, next.hull.row, col, pt, numb, cols, cols.2b, rows :
INT points, n.start, source.position.col, from, border, rows.1, original.hull.row :
... from.merged.buffer ? image1; image2; from; source.position.col;
   original.hull.row; border; numb
... rows:=image.rows[image1],image.rows[image2]:=rows,rows.1:=rows-1
... cols:=image.cols[image1],image.cols[image2]:=cols,cols.2b:=cols-(2*border)
... points := cols.2b, n.start := source.position.col*points
... SEQ n = n.start FOR numb : from.merged.buffer ? row.buff[n]
... IF centre -- host for this row of processors
   ... received[source.position.col] := TRUE, finished := TRUE
... SEQ n = 0 FOR total.columns : finished := finished AND received[n]
... IF finished -- perform hull on this row
   ... do.hull(image1,image2,total.columns*points,original.hull.row,
      border,final.iteration)
next.hull.row := original.hull.row+1
... IF next.hull.row >= rows -- finished
   ... iteration.count := iteration.count+1
... IF iteration.count >= iterations -- only centre responds
   handle.output.5(host.proc,4,nh.finished.convex.hull,
      image1,image2,processor)
otherwise

... set up for repeat (See Page 307)

otherwise -- set up for next row

... do next row (See Page 308)

... otherwise : SKIP
otherwise -- place hull
... col := border
   [image.buffer[image2][original.hull.row] FROM border FOR numb] :=
      [row.buff FROM n.start FOR numb]

: -- receive.row

Set up for repeat

... [row.buff FROM 0 FOR row.buff.size] := [blank FROM 0 FOR row.buff.size]
... SEQ seg = 0 FOR total.columns : received[seg] := FALSE
... points := cols.2b, n.start := position.col*points
SEQ seg = 0 FOR total.columns
VAL seg.proc IS (total.columns*position.row) : -- beginning of this row
... IF seg = position.col -- set up own segment
   ... col := border
      [row.buff FROM n.start FOR points] :=
         [image.buffer[image1][0] FROM border FOR points]
307
received[position.col] := TRUE
otherwise
handle.output.7(seg.proc+seg,6,nn.request.row,image1,image2,
processor,0,border)

Do next row

... SEQ row = 0 FOR rows : received[row] := FALSE
SEQ seg = 0 FOR total.columns
VAL seg.proc is (total.columns+position.row) : -- start of row
VAL this.start is position.col*points :
... IF seg = position.col -- set up own segment
... [row.buff FROM this.start FOR points] :=
[image.buffer[image1][next.hull.row] FROM border FOR points]
received[position.col] := TRUE
seg <> position.col
handle.output.7(seg.proc+seg,6,nn.request.row,imagel,
image2,processor,next.hull.row,border)
... otherwise : SKIP

For hull

PROC for.hull(VAR INT proc.num) -- Initiate row-wise convex hull.
... INT image1,image2,cols.2b,hull.border,rows,cols,rows.1,cols.1 : BOOL finished :
... from.merged.buffer ? imagel; image2; hull.border
... pass on
... check.image.number(image1), check.image.number(image2)
... rows := image.rows[image1], rows.1 := rows-1, image.rows[image2] := rows
... cols := image.cols[image1], image.cols[image2] := cols
... cols.1 := cols-1, cols.2b := cols-(2*hull.border)
... IF total.columns = 1 -- perform hull for single column
... perform hull for single column (See Page 308)
otherwise
... multi-column hull (See Page 309)
: -- for hull

Perform hull for single column

SEQ hull.row = 0 FOR rows
INT col, end.col, x.diff, max.slope, y.diff, x, big.hull.pixel :
... INT c.count, slope, int.diff, int.start, hull.pixel : BOOL finished :
... col := 1
WHILE col <= cols.2b
... end.col := col, x.diff := 0, max.slope := MOSTNEG INT, y.diff := 0, x := col
finished := (image.buffer[image1][hull.row][col] = 0) -- zero pixel?
WHILE NOT finished
-- determine max slope
... int.diff := image.buffer[image1][hull.row][x] -
image.buffer[image1][hull.row][x-1]

308
x.diff := x.diff + 1, y.diff := y.diff + int.diff
slope := y.diff*slope.scale.div.x[x.diff]
... IF slope>max.slope : max.slope:=slope, end.col:=x -- max slope
... otherwise : SKIP
x := x + 1
finished:=(x>=cols.2b)OR(image.buffer[image1][hull.row][x]=0)
c.count := (end.col + 1) - col
int.start := image.buffer[image1][hull.row][col-1]
big.hull.pixel := int.start<<4
VAL max.slope.16.to.slope.scale IS (max.slope<<4)/slope.scale :
SEQ c = col FOR c.count -- write hull
VAL c.1 IS c-1 :
... hull.pixel := big.hull.pixel>>4
image.buffer[image2][hull.row][c.1] := hull.pixel\#FF
big.hull.pixel := big.hull.pixel+max.slope.16.to.slope.scale
col := end.col + 1
handle.output.5(host.proc.4.nh.finished.convex.hull,image1,image2,processor)

Multi-column hull

... iteration.count := 0
... SEQ pt = 0 FOR cols : row.buff[pt] := 0
... IF centre
INT points, col, n.start : -- set up hull
... SEQ seg = 0 FOR total.columns : received[seg] := FALSE
... points := cols.2b, n.start := position.col*points
SEQ seg = 0 FOR total.columns
VAL seg.proc IS (total.columns*position.row) : -- beginning of this row
... IF seg = position.col -- set up own segment
... col := hull.border
[row.buff FROM n.start FOR points] :=
[image.buffer[image1][0] FROM hull.border FOR points]
received[position.col] := TRUE
otherwise
handle.output.7(seg.proc+seg.6,nn.request.row,image1,image2,
processor,0,hull.border)

E.2.9 Procedures for Loading and Sending Images

... PROC load.image (See Page 309)
... PROC send.image (See Page 310)

Load image

PROC load.image(VAL INT link.num) -- Format is <image number> <rows> <cols> |<pixels>|
INT image, rows, cols :
... from.merged.buffer ? image; rows; cols
... IF (image < 0) OR (image > total.images)
... send error message

309
(rows < 0) OR (rows > image.buffer.rows)
... send error message
(cols < 0) OR (cols > image.buffer.cols)
... send error message
... otherwise : SKIP
... image.rows[image] := rows, image.cols[image] := cols
... SEQ row = 0 FOR rows  SEQ col = 0 FOR cols -- read in and store image
... INT img : from.merged.buffer ? img, image.buffer[image][row][col] := img
... INT dummy: from.merged.buffer ? dummy -- end of message
handle.output.4(host.proc, 3, nh.finished.load.image, image, proc)
: -- load.image

Send image

PROC send.image(VAL INT link) -- Format is <image number>
INT image, rows, cols :
... from.merged.buffer ? image
... check for valid image number
... rows := image.rows[image], cols := image.cols[image]
... VAL total.pixels IS rows*cols : INT first.free : BOOL succ :
... find.first.free.table(first.free, succ)
... IF succ -- set up table
... free.table[first.free] := FALSE, pt := 0
SEQ pt.row = 0 FOR rows
... [tables[first.free] FROM pt FOR cola] :=
[image.buffer[image][pt.row] FROM 0 FOR cols]
 pt := pt+cols
handle.output.8(host.proc,total.pixels+4,nh.sending.image,image,proc,total.pixels,table.buf,first.free,total.pixels)
otherwise
handle.output.5(host.proc,4,nh.error,processor,no.free.tables,image)
: -- send.image

E.2.10 Procedures for the Link Shell

... PROC handle.input (See Page 310)
... PROC input.buffer (See Page 312)
... PROC merge.buffer (See Page 312)
... PROC output.buffer (See Page 313)
... PROC take.in (See Page 314)
... PROC accept.input (See Page 314)

Handle input

PROC handle.input(VAL INT link.id)
... from.merged.buffer IS merged.buffer[merged.buf.size] : INT count,destination,tag;
... from.merged.buffer ? destination; count
... IF ((destination = proc) OR ((destination = end.proc) AND last.transputer)) OR
(destination = broadcast) -- process the message

... from.merged.buffer ? tag
... IF tag = hn.add.images -- add images
    add.images(destination)
tag = hn.check
... IF NOT last.transputer : handle.output.2(processor+1, 1, hn.check)
... otherwise : SKIP
tag = hn.clear.details -- clear details flag
... pass it on
details := FALSE
tag = hn.iterations -- iterations
... from.merged.buffer ? iterations
... IF (destination = broadcast) AND (NOT last.transputer)
    up.buffer ! broadcast; 2 ;hn.iterations; iterations;
    end.of.message; end.buf
... otherwise : SKIP	tag = hn.neighbours -- neighbours
INT cen, sing :
... SEQ edge = 0 FOR 4 : from.merged.buffer ? neighbours[edge]
from.merged.buffer ? total.columns; position.row; position.col
from.merged.buffer ? cen
centre := BOOL cen -- is this processor the centre of this row?
from.merged.buffer ? sing
single.column := BOOL sing
tag = hn.test -- testing
... IF last.transputer
... handle.output.7(host.proc,6,nh.test.response,proc,version.date[0],
    version.date[1],version.date[2],displacement)
otherwise
    handle.output.5(host.proc,4,nh.error,proc,wrong.dest,proc)
... similarly all other tags
(destination < 0) AND first.transputer

... send to host (See Page 311)

destination = end.proc
... send up circumferentially (See Page 312)

destination < 0
... send down
otherwise
... send on
: -- handle.input

Send to host

INT hold, tag, source :
... down.buffer! host.proc; count
... IF count > packet.size.limit -- place in first free table
...  INT first.free : BOOL succ :
  find.first.free.table(first.free, succ)  
  ...  IF succ
      ...  free.table[first.free] := FALSE
      SEQ pt = 0 FOR count : from.merged.buffer ? tables[first.free][pt]
        down.buffer ! table.buf; first.free; count; end.of.message; end.buf
      ...  otherwise : SKIP
  otherwise : normal buffering  -- transfer main part of message
      SEQ scan.message = 0 FOR count
      ...  from.merged.buffer ? hold, down.buffer ! hold
      ...  IF hold <> end.of.message : down.buffer ! end.of.message; end.buf
  ...  otherwise : down.buffer ! end.buf

Send up circumferentially

  INT hold, tag, source :
  ...  IF count > packet.size.limit
        -- place in first free table
  ...  INT first.free : BOOL succ :
      ...  find.first.free.table(first.free, succ)  
      ...  IF succ
          ...  free.table[first.free] := FALSE
          SEQ pt = 0 FOR count : from.merged.buffer ? tables[first.free][pt]
            handle.output.4(destination,count,table.buf,first.free,count)
          otherwise
            handle.output.5(host.proc,4,nh.error,proc,no.free.tables,tag)
          otherwise : normal buffering
      ...  pack[0] := count
      SEQ scan.message = 1 FOR count -- transfer main part of message
      ...  from.merged.buffer ? hold, pack[scan.message] := hold
      handle.output(end.proc, count + 1, pack)

Input buffer

PROC input.buffer(VAL INT input.buf.num)
  PAR i = 0 FOR in.buf.size
    input IS in.buffer[input.buf.num][i] :
    output IS in.buffer[input.buf.num][i+1] :
    WHILE TRUE
      ...  INT next : input ? next, output ! next -- pass it on
    : -- input.buffer

Merge buffer  PROC merge.buffer()
  ...  PROC transfer.along (See Page 313)
  ...  PROC transfer.out (See Page 313)
  PAR
    transfer.along()
    transfer.out()
  : -- merge.buffer

312
Transfer along merged buffer

PROC transfer.along()
    PAR i = 0 FOR merged.buf.size
    ... input IS merged.buffer[i] : output IS merged.buffer[i+1] :
    WHILE TRUE
        ... INT next : input ? next, output ! next -- pass it on
    : -- transfer.along

Transfer out of merged buffer

PROC transfer.out()
    WHILE TRUE
        ... INT this.link : merged.buffer[merged.buf.size] ? this.link
        ... IF this.link = end.buf : SKIP -- purging
        ... (this.link >= 0) AND (this.link <= 3) : handle.input(this.link) -- link id
        ... otherwise : SKIP
    : -- transfer.out

Output buffer

PROC output.buffer(VAL INT output.buf.num, CHAN output)
    ... PROC transfer.along (See Page 313)
    ... PROC transfer.out (See Page 313)

PAR
    transfer.along()
    transfer.out()
: -- output.buffer

Transfer along output buffer

PROC transfer.along()
    PAR i = 0 FOR out.buf.size
    input IS out.buffer[output.buf.num][i] :
    output IS out.buffer[output.buf.num][i+1] :
    WHILE TRUE
        ... INT next : input ? next, output ! next -- pass it on
    : -- transfer.along

Transfer out of output buffer

PROC transfer.out()
    WHILE TRUE
        ... INT next : out.buffer[output.buf.num][out.buf.size] ? next
        ... IF next = end.buf : SKIP -- purging
        next = table.buf -- send out values from a specified table
        INT table.id, number :
out.buffer[out.buffer.num][out.buf.size] ? table.id; number
...
IF (table.id >= 0) AND (table.id < num.of.tables)
...
SEQ pt = 0 FOR number : output ! tables[table.id][pt]

free.table[table.id] := TRUE
otherwise
down.buffer ! host.proc;4;nh.error;proc;wrong.table.number;
table.id;end.of.message;end.buf

next = image.buf
-- send out values from a specified image

INT image.id, number.rows, number.cols :
...
out.buffer[out.buffer.num][out.buf.size] ?
image.id;number.rows,number.cols
...
IF (image.id >= 0) AND (image.id < total.images)
...
SEQ pt.row = 0 FOR number.rows SEQ pt.col = 0 FOR number.cols
output ! image.buffer[image.id][pt.row][pt.col]
otherwise
down.buffer ! host.proc;4;nh.error;proc;wrong.image.number;image.id;
end.of.message;end.buf
...
otherwise : output ! next
:
-- transfer.out

Take in

PROC take.in(VAL INT from.id, to, tag, number.expected, number.provided, time.allowance)
-- includes possibility of no values being passed 'number.provided' comes from message
-- header and includes 1 for tag and 'number.expected' is based on tag.
-- IF number.expected < 0 then no check is made.

... from IS in.buffer[from.id][in.buf.size] :
	TIMER take.in.clock : BOOL repeating : INT cyc, now, hold :
...
IF (number.expected <> number.provided) AND (number.expected >= 0)
clean.out(from.id, tag, TRUE) -- discard to end-of-message
otherwise : into.merged.buffer ! from.id; to; number.provided; tag
...
IF number.provided > 1 -- first is tag
-- read in from channel and place in buffer -- unlimited time
... cyc := 1 -- first value (i.e. tag) has already been read
... repeating := TRUE : WHILE repeating
...
from ? hold, logit(from.id, hold), cyc := cyc+1
... IF cyc=number.provided : repeating:=FALSE; otherwise : SKIP
into.merged.buffer ! hold
...
otherwise : SKIP
into.merged.buffer ! end.of.message; end.buf -- end-of-message marker
...
from ? hold, logit(from.id, hold)
...
IF hold <> end.of.message
into.merged.buffer ! send.out.buf;host.proc;5;nh.error;proc;
faulty.termination;cyc;end.of.message;end.buf
...
otherwise : SKIP
:
-- take.in

Accept input

PROC accept.input(VAL INT link.id, destination, number)

INT tag :

314
... in.buffer[link.id][in.buf.size] ? tag, logit(link.id, tag)
-- Note that count of integers to be taken in is full message count,
-- including tag, which is sent separately.
IF -- tag provides next step
    (number <= 0) OR (number > biggest.packet) -- wrong count
    ... into.merged.buffer ! send.out.buf;host.proc;5;nh.error;proc;
        wrong.count;number;end.of.message;end.buf
    clean.out(link.id, number-1, FALSE)
tag = nh.acknowledge -- acknowledge
    take.in(link.id, destination, nh.acknowledge, 2, number, unlimited.time)
... similarly other tags
otherwise -- wrong tag
down.buffer ! host.proc;4;nh.error;proc;wrong.accept.tag;tag;
        end.of.message;end.buf
: -- accept.input