Parallel Computer Architectures and Algorithms for Medical Image Analysis

by

Granville Vincent Moore, B.Sc..

Volume 1

A thesis submitted to
the Victoria University of Manchester,
for the degree of Doctor of Philosophy
in the Faculty of Medicine.

Department of Medical Biophysics,
Faculty of Medicine,
University of Manchester.

October, 1987.

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

Abstract

Computer manipulation of digitised images is a rapidly expanding field of computer science, which has many biomedical applications. This project examines some medical applications of computerised image analysis, and studies, in detail, two applications in clinical ophthalmology. The first of these is the automatic evaluation of optic disc cup volume, for early diagnosis of glaucoma. A partially implemented model-based, iterative approach is described for this. The second ophthalmic application is the automatic detection and measurement of corneal endothelial cells, and a program is described which performs this task. From the measured performance of these programs, a need for faster image processing hardware is apparent, and a number of machine structures are studied for this purpose. To aid comparison of machines, a machine taxonomy is developed. This includes a matrix-based representation scheme for interconnection networks, from which quantitative characteristics of networks are derived. A parallel machine structure, suitable for image analysis applications, is proposed. Two machine simulators which operate at a unit level and register level are described, and simulation results are presented. A high-level language, developed for use with the low-level simulator, is described, and the implementation of a compiler for this language is also described.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this, or any other, university or institution of learning.

Acknowledgements

The author would like to thank all those who assisted in the completion of the work described in this thesis, and in the production of the thesis itself. In particular:

All of the members of the Wolfson Image Analysis Unit and the staff of Visual Machines Limited; in particular, Dr. R. N. Dixon, Dr. J. Graham and Mr. D. Pycock, for their advice on image analysis problems and assistance in de-bugging programs on the Magiscan 2;

Mr. A. Ridgeway, of the Manchester Royal Eye Hospital, for assistance in relation to the ophthalmic aspects of the project;

The Department of Computer Science, at the University of Manchester, for the use of the MU6 research computer, to provide some of the results presented in the thesis, and for the use of typesetting facilities to produce the thesis;

Mr. M. I. Wolczko of the Department of Computer Science for assistance in typesetting the thesis.

The author would also like to thank his family and friends, for the support without which this project could not have been completed. Finally, the author is greatly indebted to the project supervisor, Dr. C. J. Taylor, for his invaluable help throughout the project, in the writing of the thesis, and, in particular, for reading the manuscript.

The White Rabbit put on his spectacles.
"Where shall I begin, please your Majesty?", he asked.
"Begin at the beginning," the King said, very gravely,
"and go on till you come to the end: then stop."

'Alice's Adventures In Wonderland' — Lewis Caroll.

Contents: Volume 1

Contents: Volume 2

List of Illustrations

Chapter 13
Test Programs for Simulation

Chapter 14
Results of Low-Level Simulation

List of Plates

Chapter 11
A Low-Level Simulator

Chapter 1:

Introduction

1.1 Digital Image Manipulation

Image processing, image analysis and image reconstruction are rapidly expanding fields of computer science, and have many biomedical applications. The purpose of this thesis is to investigate some of these applications, and to study some parallel machine structures which may be of use in such applications. This chapter reviews a number of medical applications in which image manipulation techniques are used.

1.2 Medical Applications of Digital Image Manipulation

A number of differing image manipulation techniques are used in biomedical applications. Image processing involves modifying existing images, usually to enhance some particular aspect of the image, for a human observer. It is used in many imaging applications, and also as part of image analysis systems.

Image analysis is the automatic examination of images, and aims to produce some quantitative or qualitative information. The applications of image analysis are mainly in areas where large numbers of images need to be routinely examined. In these situations, human operators become prone to mistakes, and appreciable variation may be found between the performance of different operators, or of the same operator, at different times. The use of computer image analysis, in these cases, assures repeatability and reliability, and may decrease processing time.

Image reconstruction is the process of building a two-dimensional image from data obtained in some other format, such as projections obtained from computed tomography or nuclear magnetic resonance imaging. The use of computers in these areas is vital, so that the process may be performed in a reasonable time.

Specific medical applications of image processing, image analysis, and image reconstruction are discussed in the remainder of this chapter.

1.2.1 Medical Imaging

Image processing is frequently used in medical imaging, in such fields as ultrasound scanning [2], where unprocessed images are difficult to interpret, due to high noise levels. Image processing has also been applied to scanning transmission electron microscope images, to give Z-contrast images [3], where the image intensity relates only to the depth of the viewed object, and not to its density. These processed images are more easily interpreted by the operator.

Image reconstruction techniques have been used to construct three dimensional images from projections, in computerised tomography (CT) [4, 5] and in nuclear magnetic resonance (NMR) imaging [6]. These techniques are of high importance, as they are non-invasive, and yield a large amount of clinically useful information.

1.2.2 Radiology

Clinical X-ray images are often of poor quality, and simple image processing operations, such as pseudo-colouring and edge enhancement, have been widely used to aid clinical assessment of such images [7, 8]. Image processing may enable detail to be seen in low-contrast areas of images, and so these may be more easily interpreted by clinicians. Low-exposure radiographs may be digitised by automatic counting of individual photographic grains. This gives a very high resolution image, and allows lower X-ray dosage to be used. This technique also allows shorter exposure times to be used in tritium autoradiography, where emission levels are very low [9].

The automatic detection of pulmonary nodules in chest X-rays, for the early diagnosis of pneumoconiosis, has also been investigated. Jagoe [11] used patterns of gradient directions around nodules to detect them, whereas Sklansky et al. [13] combined Sobel edge detection with boundary following, to detect the circular nodules. Gonçalvez [14] used spatial grey-level dependency matrices to detect nodules by their texture.

Image analysis has also been used in cardiology, to quantify the contraction of the left ventricle [16], and to monitor arterial blood flow rates [17]. Analysis of X-ray images of coronary arteries may be used

to estimate their cross-sectional area, and by modelling the vessels as generalised cylinders, three-dimensional reconstructions of the arteries may be obtained [19].

1.2.3 Digital Subtraction Angiography

Angiography is an X-ray technique used to examine blood vessels, and involves the injection of contrast agents into the observed vessels. In digital subtraction angiography, the resulting angiogram is processed by the subtraction of an X-ray image taken without contrast medium [6]. The images must be placed in correct registration before subtraction, to minimise the remaining background. This may be done automatically, after digitisation. Thus, an image of the contrasted blood vessels may be obtained, without the complications of surrounding tissues and bone. Digital subtraction angiography may be used to process multiple-image sequences which show the pumping action of the heart, and this may be used as the basis for three-dimensional reconstruction of the motion of the ventricle walls [20].

1.2.4 Haematology and Cytology

The large numbers of red blood cell counts, and differential white blood cell counts, which are required for routine medical tests have promoted the development of a large number of systems to perform these tasks. Simple image analysis operations, such as region counting, have been used to count red blood cells [5], and more complex systems may be used to produce differential cell counts [21, 22, 24].

For similar reasons, a number of systems have been developed to automatically process cervical smear slides, in screening for uterine cancer. Some of these systems locate suitable cells for manual assessment [26], whereas others classify cells on the basis of nuclear size, shape, optical density, and cytoplasm area [27, 28, 29, 30].

1.2.5 Two-Dimensional Gel Electrophoresis

Two-dimensional gel electrophoresis is a technique, applicable to a large number of problems in biochemistry, pathology, genetics and clinical research [31, 32], to which image analysis has been applied.

Two-dimensional gel electrophoresis is a method by which mixtures of proteins, or other complex organic molecules, may be separated, according to different characteristics in each dimension, on a polyacrylamide gel [33, 34]. Many separation techniques exist, and any two may be combined to make a two-dimensional gel. Separation may, typically, be performed according to molecular charge, molecular weight or molecular size. The proteins may be stained for examination, or radio-labelled proteins may be used to to produce autoradiographs. In each case, the optical density of each spot obtained is proportional to the amount of a particular protein in the original sample. Gels prepared in this way may be compared with other gels, for identification purposes, or with a database of gel representations [35], to identify specific proteins.

Many systems have been developed which use computer image analysis techniques in the analysis and matching of electrophoretic gels [31, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46]. In these systems, processing is usually divided into a spot-detection phase, and a gel-matching phase. Spot detection may be performed in many different ways, but this usually involves background subtraction and streak removal, followed by some form of local peak detection. In some systems, the spots are modelled as two-dimensional Gaussian functions [36, 42, 46]. This may be done as part of the matching process, or to reduce the storage requirements of the large database required for comparison purposes. The matching of gel images is not, however, a straightforward task, since, during gel preparation, the rate of propogation of proteins in each of the two dimensions is not necessarily constant, and this causes spatial non-linearity in the resulting gel. Most methods of gel matching require some form of manual intervention to indicate common points in pairs of images, and, between these, interpolation is used to match other protein spots.

1.2.6 Clinical Chromosome Analysis

Chromosome analysis is a clinical technique in which chromosomes are classified, sorted, and examined to detect genetic abnormalities [48]. Often, the chromosomes are labelled and presented in an ordered array, known as a karyogram. An automatic karyogramming system which uses the Magiscan 2 image analyser to examine human chromosomes has been developed by the Wolfson Image Analysis Unit, at the University of Manchester, in conjunction with the Rigshospitalet, Copenhagen [49, 50, 51, 52, 53]. The

system operates in two phases; an unsupervised section, which is usually run overnight, and a supervised section. In the first phase, slides are scanned to detect cells which are in the process of division (in metaphase), and are thus suitable for analysis. In the second, interactive, phase, these metaphases are re-located and automatically paired and sorted. A karyogram is thus assembled, and may be recorded on a photographic output device. Other similar systems have been developed elsewhere [54, 55].

1.2.7 Clinical Ophthalmology

Digital image manipulation has a number of applications in clinical ophthalmology. Baudoin et al. [56] describe a computerised system for the detection, counting and measurement of microaneurisms from fluorescein angiograms. Microaneurisms are small saccular dilations of the retinal blood vessels which are often observed in cases of diabetes mellitus [57, 59]. These appear as small circular spots on the fluorescein angiogram and are detected, by this system, using an algorithm based on morphological image transforms.

The level of retinal intravascular oxygen saturation is a parameter of interest to clinicians, and this has been measured by the digital imaging of the retinal blood vessels, using two different wavelengths of light [61]. Using the known optical absorption properties of of oxygenated and de-oxygenated blood, the percentage of oxygenated heme groups was estimated from the difference between these images.

Image analysis has also been used in the field of retinal hemodynamics [61], to measure blood flow velocity and flow volume. To do this, pulse-injected fluorescein angiograms were digitised in real-time, and the variation with time of the optical density of fixed points on blood vessels was observed. From the resulting graphs, the flow time between these points was calculated, and, by estimating the cross-sectional area of the vessel, the flow volume was derived. Fram et al. [62] describe a study in which the amount and distribution of photolabile visual pigment in the retina was assessed. The fundus was imaged in a dark-adapted state, when the maximum amount of retinal pigment is present, and the retina was then bleached by the application of a strong light. A second image was obtained, and the two were subtracted digitally. This difference between the dark-adapted retina and the bleached retina is indicative of the amount of visual pigment present.

Two other applications of digital image manipulation in clinical ophthalmology are discussed in chapters 2 and 3.

1.3 Hardware for Medical Image Manipulation

A large number of differing machines have been used, by different workers, for medical image manipulation tasks. In most cases, these have been general-purpose computers, occasionally with additional, custom-built hardware for image processing. Some of these machines make use of parallelism in the image manipulation task to increase processing speed. Later in the thesis, the hardware requirements for image analysis are examined, and parallel hardware structures are considered.

1.4 Summary

A number of medical applications of digital image manipulation have been discussed. All of these areas are the subject of continuing research, and many other applications exist. Chapters 2 and 3 discuss in more detail two applications of image analysis in clinical ophthalmology, and describe two programs which were written in connection with these. Chapter 4 examines the hardware requirements of image analysis programs, in relation to the performance of these two application programs, and discusses the use of parallel machines for image analysis.

The remainder of the thesis examines parallel machine architectures in detail, and a taxonomy for parallel machines is defined. A number of classes of parallel machines are described, and their applicability to image analysis is discussed. A parallel machine structure, designed for image analysis applications, is proposed, and the implementation of critical sections of this machine is discussed. Results of simulations of this machine structure are presented.

The last thing one discovers
in writing a book
is what to put first.

'Pensées'— Blaise Pascal

Chapter 2:

A Feasibility Study of the Use of A Priori Knowledge in the Analysis of Stereo Pairs

2.1 Introduction

This chapter is concerned with the feasibility of using the Magiscan 2 image analyser to provide an automatic system for the evaluation of optic disc cup volume from stereo pairs of photographs. An increase in the cupping of the optic disc is considered to be an important early symptom of glaucoma, and an automatic system for cup volume evaluation would provide a useful clinical tool for glaucoma screening. In this chapter, an overview of the problems of glaucoma is presented, and a partially implemented model-based scheme for cup volume measurement is described. The Magiscan 2 is also briefly described.

2.1.1 Ocular Hypertension and Glaucoma

Ocular hypertension is a general term for a group of conditions in which intraocular pressure rises above the value of approximately 20mm. Hg [63]. Normal intraocular pressure is approximately 15mm. Hg. Glaucoma is a condition, associated with ocular hypertension, in which visual field loss occurs as a result of atrophy of the optic disc. Not all ocular hypertensives necessarily develop glaucoma. The transition between ocular hypertension and glaucoma is not well defined, but an enlargement of the optic disc cup is usually considered to be an important symptom. The initial rise in intraocular pressure may occur for many reasons, but the most common causes are associated with aqueous humor flow blockages in the anterior part of the eye.

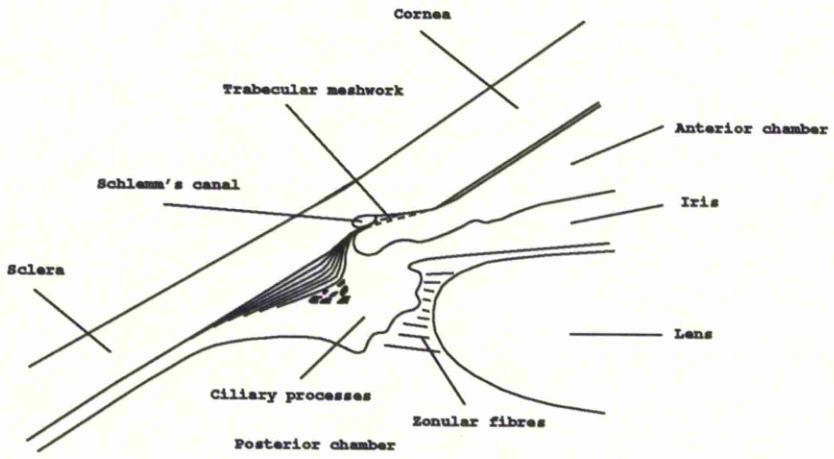


Fig. 2.1: The anterior part of the human eye.

2.1.2 Anatomy of the Eye

The basic anatomy of the anterior part of the normal eye is shown in Fig. 2.1. The anterior and posterior chambers are filled with aqueous humor. This is secreted by the ciliary body in the posterior chamber, and flows through the iris to leave the eye via a specialised drainage apparatus, known as the trabecular meshwork, at the edge of the anterior chamber. Intraocular pressure is thereby maintained constant [57, 64]. The normal optic disc is shown in Plate 2.1. A shallow cupping occurs towards the centre of the disc, but this does not reach the neuroretinal rim. The healthy disc is an even, pinkish colour.

2.1.3 Classification of Glaucomas

Glaucomas may be divided into primary glaucoma and secondary glaucoma [57, 65, 66, 64]. Primary glaucoma occurs independently of any other ocular condition, whereas secondary glaucoma is a consequence of some separately identifiable condition, such as an intraocular tumor or vascular disorder.

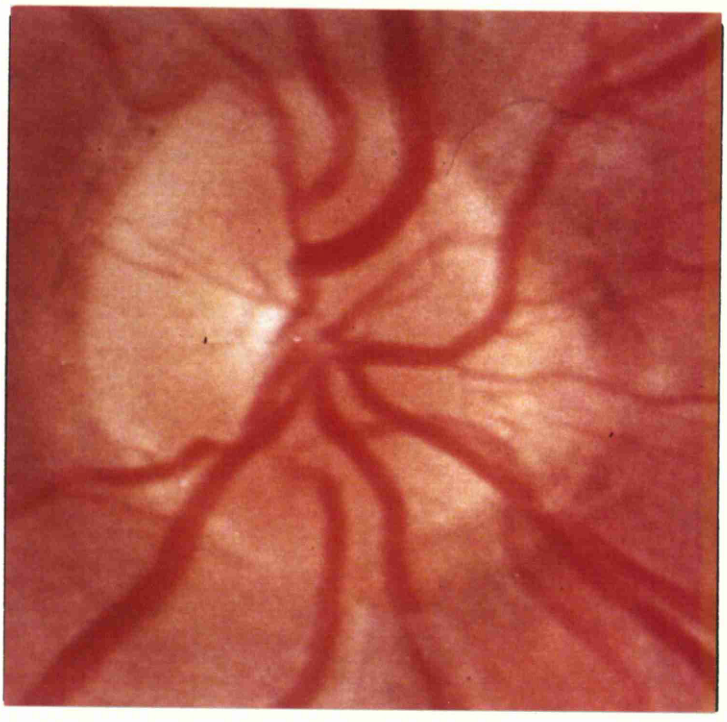


Plate 2.1: The normal optic disc.

2.1.3.1 Primary Glaucoma

In the primary glaucomas, intraocular pressure increases as a result of some form of blockage of the aqueous humor drainage mechanism. Primary glaucoma may be sub-divided into closed-angle glaucoma, open-angle glaucoma and congenital glaucoma.

i) Closed-angle glaucoma

Closed-angle glaucoma occurs when the anterior chamber is very narrow and the iris comes into contact with the trabeculum, blocking the aqueous flow. The drainage angle is then said to be closed. Diagnosis is difficult in the early stages, since there are few or no external symptoms. In later stages, ocular hypertension, optic disc cupping and visual field loss develop. The normal treatment is surgical, in the form of a peripheral iridectomy [65]. This procedure involves the removal of a small sector of the iris, to allow the flow of aqueous humor to be restored.

ii) Open-angle glaucoma

Here the drainage angle is open, but an obstruction is present at the trabeculum, in the canal of Schlemm, or in the aqueous vein. Symptoms are similar to closed-angle glaucoma, and the two are distinguished by gonioscopy. This is a clinical procedure which involves the use of a special contact lens to allow the drainage angle to be examined. Treatment is usually medical, and surgery is only performed in extreme cases. This is the most common form of glaucoma, and accounts for approximately 90% of cases.

iii) Congenital glaucoma

This is a recessively-inherited ocular defect in which the iris is connected to the trabeculum, preventing aqueous flow. Symptoms are similar to open-angle and closed-angle glaucomas, but usually occur in early infancy. Gonioscopy is used to distinguish congenital glaucoma from open-angle or closed-angle glaucoma, and a surgical procedure known as a goniotomy is the usual treatment. Goniotomy involves making a circular incision to separate the iris from the trabeculum, and is performed through a small incision at the edge of the corona.

No definite causes have been established for primary glaucomas, but its occurrence has been positively correlated with several factors, including a history of acute blood loss [67], diabetes [67], abnormal con-

trast sensitivity [68], and negatively correlated with episcleral venous pressure [69]. Ocular hypertension without subsequent development of glaucoma has been positively correlated with cigarette smoking [67].

2.1.3.2 Secondary Glaucoma

In secondary glaucoma, the condition arises as a result of some other (possibly non-ocular) condition, and treatments are varied, according to the primary cause.

2.1.4 Diagnostic Techniques

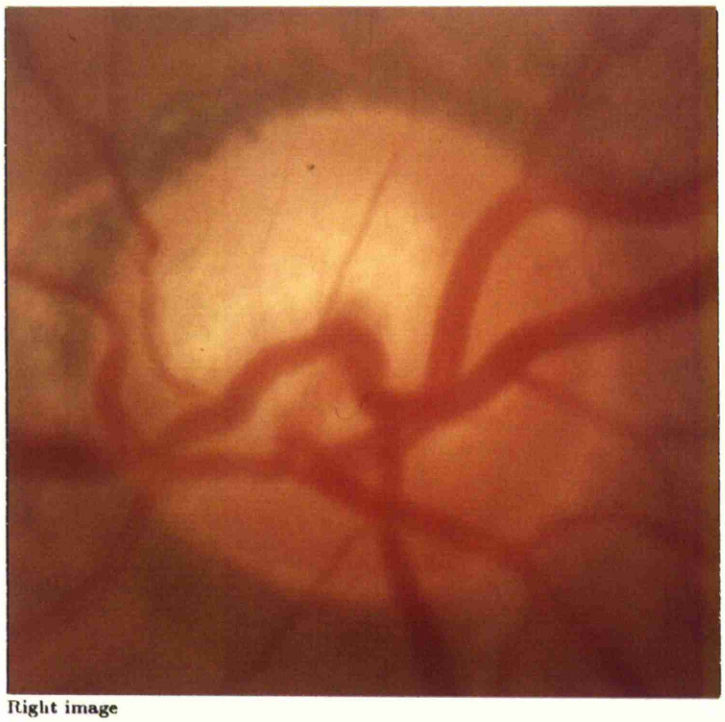
Visual field loss may be measured by perimetry [71, 72], and ocular pressure may be measured by tonometry [57, 64], but neither of these allows early diagnosis. Enlargement of the optic disc cup is an early sign, but is only detectable in vivo by examination of the optic disc, through the dilated pupil. This is a highly subjective method, and results vary considerably between examiners. Stereo pairs of photographs (Plate 2.2) are often used in diagnosis [74], although in live examination, a monocular ophthalmoscope is sufficient, since the examiner may make use of parallax [71]. Disc evaluation is based on a number of factors:

i) The ratio of the cup area to the optic disc area is a commonly-used measure of cupping [75, 76, 77, 78, 79, 80]. The mean value for normal discs is approximately 0.4.

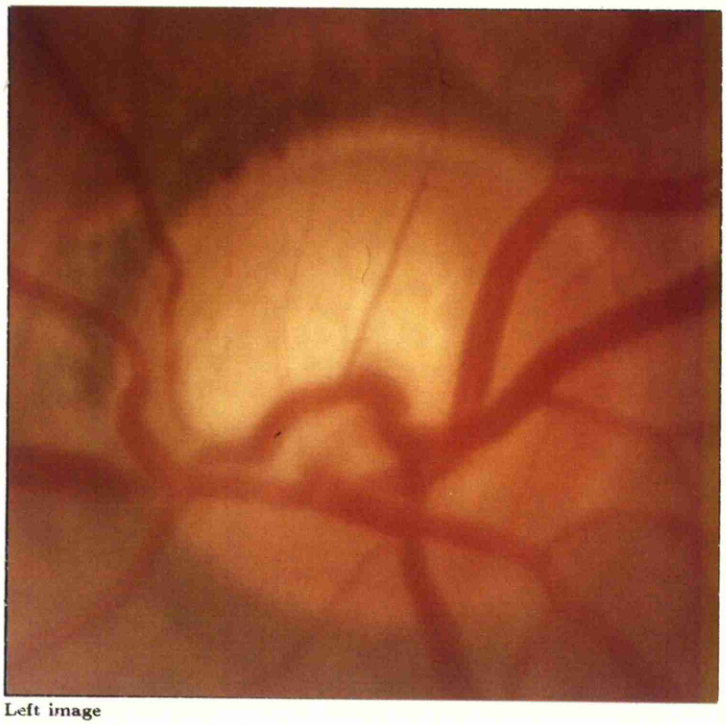
ii) The eccentricity (horizontal to vertical diameter ratio) of the cup is often used in diagnosis [78, 81, 82, 83]. A horizontal cup ovality is often found in glaucoma.

iii) Cup volume is an important parameter, but difficult to measure directly. Krakau [84] projected vertical lines onto the disc, and by measuring the shadowed area with a planimeter, estimated the cup volume. This is, however, a time-consuming manual procedure.

iv) The degree of pallor of the disc is considered to be independent of cupping [85, 81], but also to be a sign of glaucoma. Pallor is difficult to measure quantitatively, but the area of filling defects in fluorescein angiography is associated with the degree of pallor [86, 80]. Pallor has been found to increase with age, in normal eyes [87].



Right image



Left image

Plate 2.2: The glaucomatous optic disc: a stereo pair.

v) The state of the neuroretinal rim is another important factor, sometimes considered to be the most important. It is thought that visual field loss only commences when cupping or pallor reaches the neuroretinal rim [81, 88].

All of these parameters vary considerably between individuals, so that without a long term study of a disc, it is often impossible to tell whether any change from normal has taken place. Comparison of the affected eye with its fellow eye is of great value in resolving these cases, since fellow normal eyes match each other closely [75, 76, 77, 78, 85, 82, 89].

Attempts have been made to predict visual field from disc appearance [81, 88, 90]. These have not, however, proved accurate enough to replace perimetry. Hitchings [89] defined six qualitative classes of optic disc appearance, based on stereo pairs of optic disc photographs. These are normal, overpass cupping, cupping with pallor of rim, cupping without pallor of rim, focal notching of the rim, and bean-pot cupping.

2.1.5 Automated Diagnostic Techniques

The lack of external symptoms in early glaucoma is a problem which may result in the disease reaching an advanced state before detection [81, 91]. Glaucoma screening clinics have been established, to check relatives of known glaucoma sufferers [81, 92, 68]. These are considered to be at a high risk because of the apparent dependency of glaucoma on inherited ocular characteristics. These clinics require large numbers of examinations to be performed and it is, therefore, desirable to speed up the process by automation.

i) Perimetry

Perimetry may be automated by using a computer to operate a matrix of lights to evaluate the visual field [93, 94, 95, 96, 97, 98], and also to display the results in a convenient form [99, 100, 101]. This method has been shown to produce more consistent results than manual perimetry [93]. This is attributed to the faster procedure causing less patient fatigue, and the lack of distractions caused by the actions of the ophthalmologist in the manual procedure.

ii) Tonometry

Tonometry is a quick and simple procedure, and automation is not really worthwhile.

iii) Cupping evaluation

Several moves have been made towards automatic cupping evaluation. Pe'er and Zajicek [102] digitised images of the optic disc and, from these, generated images of the grey-level contours of the disc, to aid manual assessment. Portney [103, 104] used manual photogrammetry [105] on stereo photographs to obtain the three-dimensional contours of the optic disc. These images were digitised and, from them, the cup volume was estimated. Photogrammetry is, however, a time-consuming manual process. Kottler et al. [106, 107, 108] used digital photogrammetric techniques [109, 110] on stereo photographs of a Zeiss model eye, to obtain estimates of the optic disc cup volume. This technique requires manual intervention to identify corresponding points in the two images and is, again, time-consuming.

The remainder of this chapter is concerned with the feasibility of automatically measuring the volume of the optic disc cup, from stereo photographs.

2.2 Approaches to Automated Cup Volume Evaluation

A number of possible approaches to automatic cup volume evaluation are now discussed, with a view to implementation on the Magiscan 2 image analyser. Processing may take appreciable time, and so it is best done using a batch processing system, to reduce the time that a patient must spend in the clinic. For this reason, all of these methods use stereo photographs of the optic disc, similar to those shown in Plate 2.2.

2.2.1 Use of Retinal Vessels

This approach uses the branching points of the retinal vessels as reference points. A given branch point is identified in each image, and its position in three-dimensional space may be calculated. A surface may then be interpolated between these points. This approach suffers from a number of problems:

i) The retinal vessels cannot be taken to accurately indicate the surface contours, since in overpass cupping [89] the vessels do not lie on the surface of the optic disc.

ii) Identification of corresponding points in the two images is not straightforward. Large disparities between the images may be required, to give sufficient depth resolution.

iii) There are often large gaps between the visible retinal vessels, and this would necessitate interpolation across large areas of the disc.

This method appears, therefore, to be unsuitable.

2.2.2 Grid Projection Systems

This approach involves the projection of a regular grid pattern onto the optic disc, to provide a set of reference points. The positions of these points may be calculated in three dimensions, from stereo photographs. Grid projection onto the optic disc is, however, a difficult process. Gloster [111] used a modified fundus camera to project a rectangular grid onto the disc, but encountered problems concerned with focussing. In cases where cupping of the disc was present, it was not possible to focus the grid on both the floor of the cup and the surrounding retina simultaneously, due to their differing depths. In addition, the tissue of the disc itself was found to be a strong diffuser of light, and the grid shadows obtained on the disc were, therefore, weak and diffuse.

2.2.3 The Shape-From-Shading Approach

The shape-from-shading approach [112, 113] may be used to find the orientation of a surface, if a simple analytic expression is known for the reflection at the observed surface. There are a number of different ways in which this may be done. One relaxation-based method [113] employs a Gauss-Seidel iterative algorithm, to find the gradient of the surface, using the constraints that the surface gradient and reflectance function must match the observed intensity, and that the gradient function must be continuous. This approach relies on the use of a simple expression for the surface reflectance. The optic disc has, however, been shown to be a strong diffuser of light. It is, therefore, unlikely that a mathematically tractable solution could be found.

2.2.4 A Model-Based Approach

Due to the lack of natural reference points, and the difficulty involved in creating artificial ones, it seems that insufficient data is directly available, in stereo photographs, to produce unique three-dimensional

interpretations. The model-based approach attempts to make use of a priori knowledge of the disc structure and optical properties, to allow an appropriate three-dimensional interpretation to be found.

In this scheme, an approximate three-dimensional model of the optic disc surface is created, using the a priori knowledge that the disc is a flat surface with a shallow cupping. A graphics system is used to generate a stereo pair of images from this three-dimensional model, viewed so as to correspond with the geometry of the optic disc photographs. The images of the model and the observed disc images are compared, and the model is modified appropriately. A new pair of images is generated from the model, and the process is repeated until a sufficiently close match is obtained. Such a system would be unlikely to be able to deal with detailed structures such as retinal vessels, and these would have to be removed from the disc images, before matching with the model. This could be done using a linear, or non-linear, filter.

2.3 A Graphics Package for Surface Modelling

A graphics system devised for use in a model-matching system is now described. This application requires some features not normally encountered in other surface modelling applications.

i) The surface of the optic disc may not be modelled by a simple equation, or by interpolation between a small number of data points. The model must be capable of representing an arbitrary surface, and must, therefore, contain a large amount of detailed surface description data.

ii) Conventional graphics systems are designed to produce images which are acceptable to the human eye. This does not necessarily require the image to be strictly accurate in terms of geometry or shading. Deliberate inaccuracies are often introduced in order to aid human recognition of computer generated images. An example of this is exaggerated perspective [114], which aids depth perception. A system which matches models to observed images requires generated images which are accurate in geometry and shading.

The optical theory and the implementation of the graphics system for model-matching are now described.

2.3.1 Optical Theory and Mathematical Modelling

For most graphics applications, the colour or light intensity of an object viewed by an observer may be regarded as the sum of of the effects of four phenomena [115, 114]. These are:

i) Diffuse illumination

This is a low-level illumination which models diffuse background lighting originating uniformly from all directions. This prevents part of an object which is not in direct light from being totally black and thus invisible. Diffuse lighting is constant for all surfaces across an entire scene.

$$E_d = RI_d,$$

where E_d is the energy radiated due to diffuse illumination, R is the reflection coefficient for the material and I_d is the intensity of the diffuse light.

ii) Diffuse reflection

From Lamberts law, the amount of energy impinging on a surface, from any light source, is proportional to the cosine of the angle between the incident ray and a normal to the surface. The energy radiated from the surface by diffuse reflection is proportional to the energy received, so that

$$E_r = R\cos(i)I_s,$$

where E_r is the energy radiated due to diffuse reflection, i is the angle of incidence of the impinging light and I_s is the intensity of the incident light.

iii) Specular reflection

The surface properties of most materials cause light which falls at some constant incident angle to the surface normal to be reflected over a range of angles. This is known as specular reflection, and results in a non-linear cone of light intensity which is narrow for shiny surfaces, and wide for dull surfaces. This may be modelled by the equation

$$E_s = W(i)I_s\cos^n(s),$$

where E_s is the radiated energy due to specular reflection, W is the specular reflection coefficient of the material, which is a function of i, and s is the angle between the directly reflected ray and

the observed ray. The constant, n, controls the 'shine' of the surface, and ranges from unity, for a dull surface, to about ten for a shiny surface.

iv) Transparency effects

The energy transmitted due to transparency may be assumed to be proportional to the amount of light falling on the back of a surface, giving

$$E_t = T_p E_b,$$

where E_t is the radiated energy due to transparency effects, E_b is the energy received by the rear of the surface, and T_p is the coefficient of transparency, which is between zero and one. A similar method may be used to model light transmitted from adjacent facets, by diffusion through the surface material.

These four components may be independently calculated, and summed to give a modelled illumination value for each surface in a scene.

2.3.2 Methods of Surface Modelling

To generate an image, a grey-level value must be calculated for each point on the surface of the model. This may be done by independent calculation of the value for each pixel, as in ray tracing [116], but this is extremely time-consuming. The amount of calculation required to generate an image is usually reduced by using surface models composed of a number of plane facets, each of which has a uniform grey-level across its surface. If these facets are very small, the increase in speed is not significant; if they are too large, then the image will not have the required accuracy. Since the modelled surface requires high resolution only in regions of high curvature, it would be advantageous to vary the sizes of the facets to give a smooth appearance with minimum workload.

A graphics system was developed for the Magiscan 2, with the intention of modelling the optic disc cup and evaluating its volume by the method described earlier in this chapter. Before this program is discussed in detail, the Magiscan 2 is briefly described.

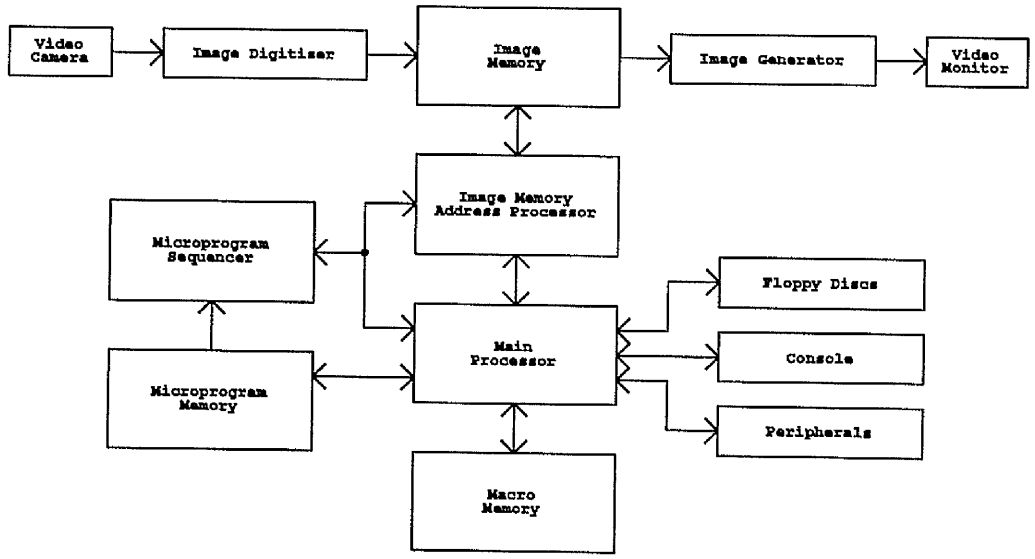


Fig. 2.2: The Magiscan 2 image analyser.

2.4 A Brief Description of the Magiscan 2

The Magiscan 2 is an image analysis machine, designed and constructed by R. N. Dixon and C. J. Taylor [118] in the Department of Medical Biophysics, at the University of Manchester. Its structure is shown in Fig. 2.2. Up to three Bosch TYK 9A vidicon cameras provide images, which are digitised to a spatial resolution of 512×512 pixels and grey-scale resolution of 64 grey levels. Images may be viewed on a built-in monochrome monitor, or on a separate colour monitor, which allows pseudo-coloured images to be displayed. An image store of up to two megabytes is provided, composed of eight or sixteen planes of $1024 \times 1024 \times 1$-bit semiconductor storage. These are normally used to store a mixture of $512 \times 512 \times 1$-bit and $512 \times 512 \times 6$-bit images.

The main processor has a 16-bit word length and a cycle time of 150ns. The purpose of the memory address processor is to calculate the memory addresses of pixels in the image memory, and to perform operand fetches from it, in parallel with main processor activity. The main processor and the memory address processor are microcoded, and are based on bit-slice devices. Each microcode instruction has three fields, containing a main processor instruction, a memory address processor instruction, and a jump field giving the address of the next microcode instruction. Four kilobytes of microcode store is

available, and this is partially occupied by the p-code interpreter, which supports the UCSD p-system operating system [119].

The operating system and user programs are compiled into p-code, which is a pseudo-machine code, interpreted at run-time by the p-code interpreter. All p-code programs and data reside in the 64 kilowords of 16-bit macro memory. The UCSD p-system provides editing, compiling and floppy disc file handling facilities. Microcode written by the user may be dynamically loaded from the macro memory, by the main processor, to allow time-consuming sections of programs to be efficiently encoded. Peripherals include a motorised microscope, a 35mm film transport, dot-matrix printers, and a Ramtek 4500C photographic hard-copy output device.

2.5 GRAFIX — A Graphics Package

UCSD Pascal allows programs to be split into separately compilable modules known as units, which must be linked together before running. This feature allows procedure libraries to be constructed, and a graphics module, known as the GRAFIX unit, was implemented to generate stereo pairs of images. The procedures in the GRAFIX unit allow three-dimensional objects to be constructed from triangular facets. Triangles are used because three vertices are the minimum required to define the plane of the facet, and the use of more than three vertices can allow rounding errors to create data inconsistencies. The vertices are stored as a list of points, using a three-dimensional homogeneous coordinate system, and the facets are represented by triples of pointers to the vertex list. These structures are implemented as dynamic arrays, which are not a feature of standard Pascal, but are implemented in the UCSD Pascal system on the Magiscan 2. Resolution may be locally increased by introducing a new vertex at the centre of the triangle, and adjusting the facet list to give three smaller triangles, as shown in Fig. 2.3. This, together with the ability to reposition individual points without altering the facet vertex pointers, allows the model to be adjusted locally without undesirable global changes.

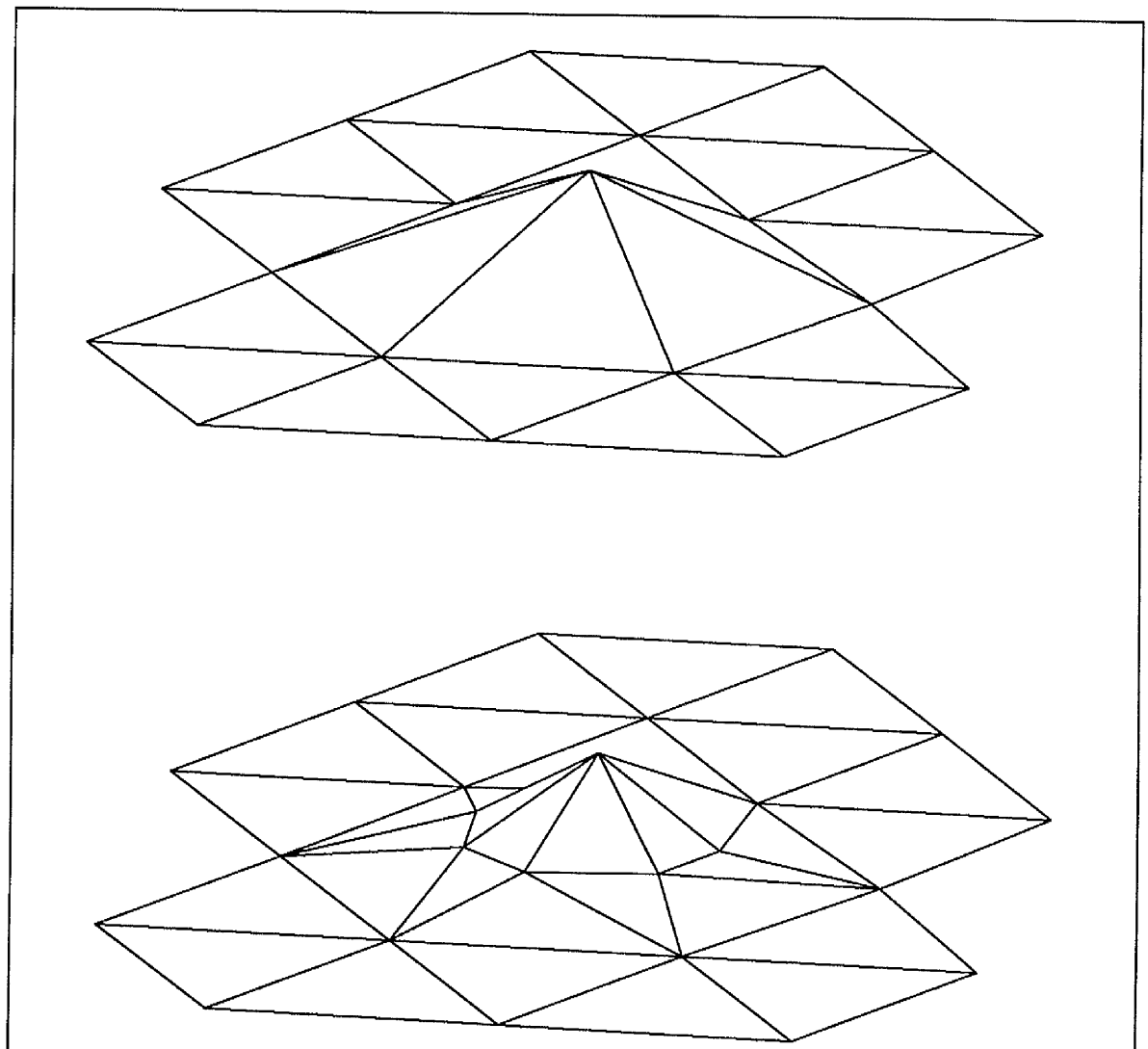


Fig. 2.3: Facet splitting to increase resolution.

2.5.1 Image Generation

Stereo pairs of images may be generated as follows:

i) A matrix of vertices are defined in three-dimensional space, at a suitable viewing position for the first image. Triples of pointers are then generated to form triangular facets. These triples have a clockwise sense, which defines the front and the rear of the facet.

ii) The equation of the plane of each facet, and, thus, the normal to the surface, is calculated. This is done by solving the set of simultaneous equations obtained by substituting the coordinates of

the vertices into the equation of a general plane, $ax + by + cz + 1 = 0$. A determinant method is used for this, and it may be shown that, since the directional sense of the three points is known to be clockwise, the sign of c indicates whether the facet is forward or backward facing.

iii) The facets are examined to see if any one shades another from the light source, which is assumed to be at infinite distance. This may be done by finding the direction cosines of a line from the light source to each facet, and comparing all possible pairs of sets of direction cosines. Any pairs in which the direction cosines are equal, within a certain tolerance, must be co-linear with the light source. The further of the pair from the light source must, therefore, be shaded by the nearer.

iv) For each unshaded forward-facing facet, the angle of incidence of light, i, and the angle of specular reflection, s, are calculated. From these, the diffuse illumination component of illumination, E_r, and the specular component, E_s, may be calculated.

v) For each forward-facing facet, including those which are shaded from direct light, the component of light falling on the reverse side which is transmitted by transparency, and the component transmitted by diffusion from adjacent facets, are calculated. The grey level of each facet may be calculated by summing the individual components.

vi) Steps ii) to v) are repeated for each light source, and the grey-level value for each facet is accumulated.

vii) Each facet is projected, using a perspective transformation, onto the plane of the viewscreen, and is drawn using the calculated grey value.

viii) The model and light sources are rotated to a new position and used to generate the second image of the stereo pair. This rotation is performed using a three-dimensional linear transformation.

Transparency and shadowing were not included in the implemented version. Specular reflections were partially implemented, but their completion was not felt to be worthwhile, for the reasons stated in the summary of this chapter.

2.5.2 Performance of the GRAFIX System

A simple solid modelling program was written to evaluate the GRAFIX system. Since this is merely a demonstration of the modelling technique, the solids and surfaces which were used were limited to easily-generated mathematical functions although, in principle, any arbitrary surface could be used. Plates 2.3 and 2.4 are examples of the output of the program. These models of spheres are viewed from an elevation of 30° under direct illumination from the top right foreground. Plate 2.5 shows a cupped surface, generated using the function $z = \cos r$, where $r = \sqrt{x^2 + y^2}$. This is not meant to model the exact shape of the optic disc cup, but is included to illustrate the use of the GRAFIX package in generating images similar to those required for an optic disc modelling system. Plate 2.6 shows a wire-mesh version of this image, generated by drawing facet boundaries only. Wire-mesh images may be of use for human viewing, as surface orientation is more easily discerned from such images. Plate 2.7 shows a stereo pair of images, generated using the same function. In this stereo pair the object is seen from two viewpoints, separated by an angle of 20° and is lit from 45° to the right of the centre-line. Plate 2.8 shows a wire-mesh version of these images. In this pair, the differences between the left and right images may be clearly seen.

Each of these images took about 30 seconds to generate, and each is composed of about 1600 facets. This size limit is imposed by the size of the Magiscan 2 macro memory. The unused image memory could be used to increase the size of the model, but only at a considerable time penalty.

2.6 Summary

The images generated by the GRAFIX system are of relatively low accuracy, since specular reflections, shadows and translucency were not implemented. In this form, the program takes approximately 30 seconds to generate each image. A microcode version of the program would be extremely complex, and would not be expected to produce a large speed increase, since, in the compiled Pascal version, most time is spent in the microcoded arithmetic functions.

An iterative modelling technique would require many images to be generated, and these would have to be of high accuracy, including shadows and diffusion of light within the modelled surface. An

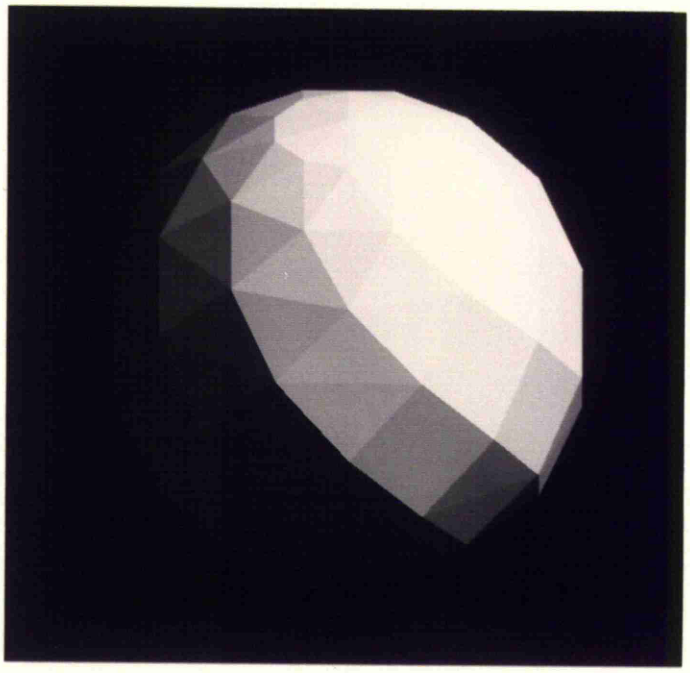


Plate 2.3: A low-resolution sphere, generated using the GRAFIX package.

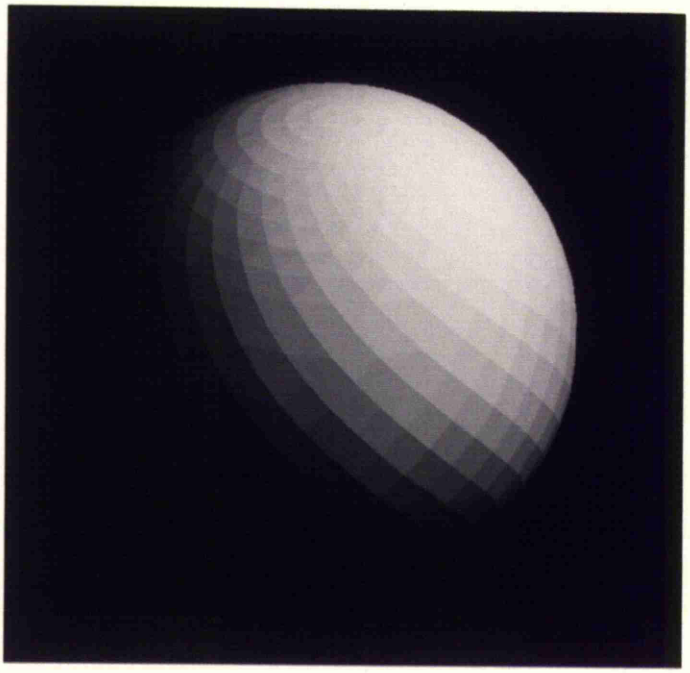


Plate 2.4: A high-resolution sphere, generated using the GRAFIX package.

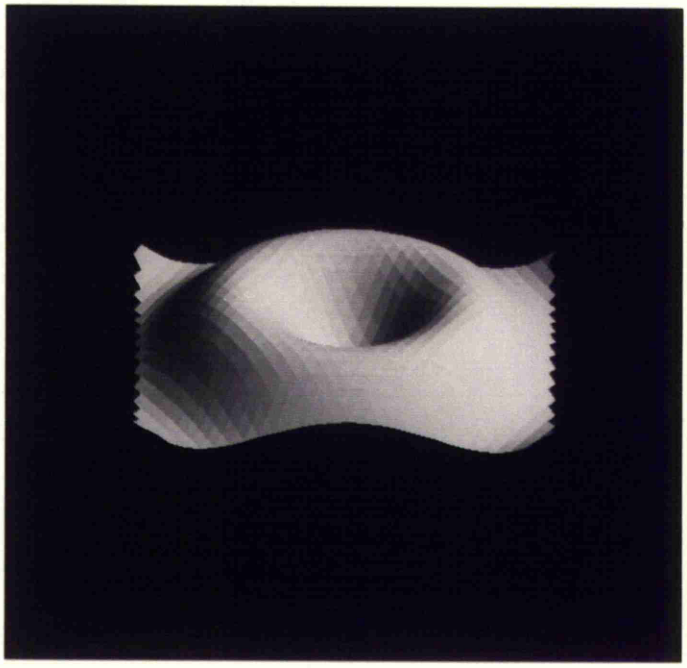


Plate 2.5: The surface $z = \cos r$, generated using the GRAFIX package.

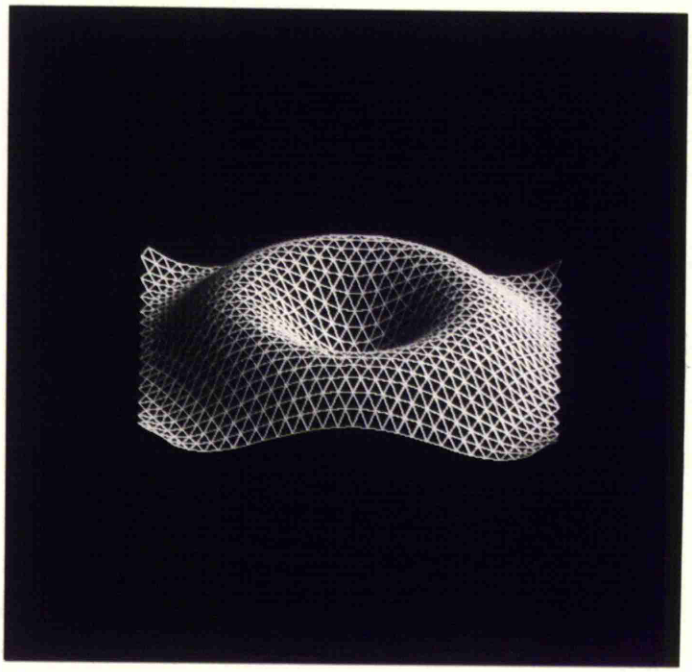
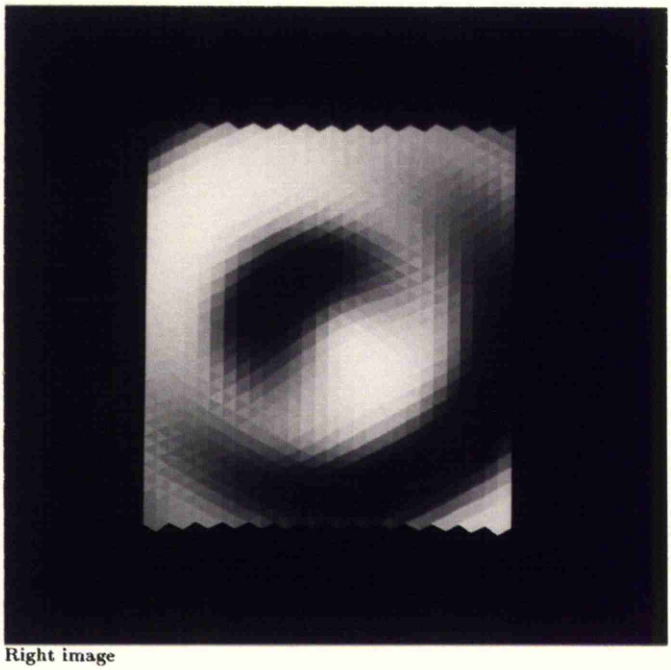
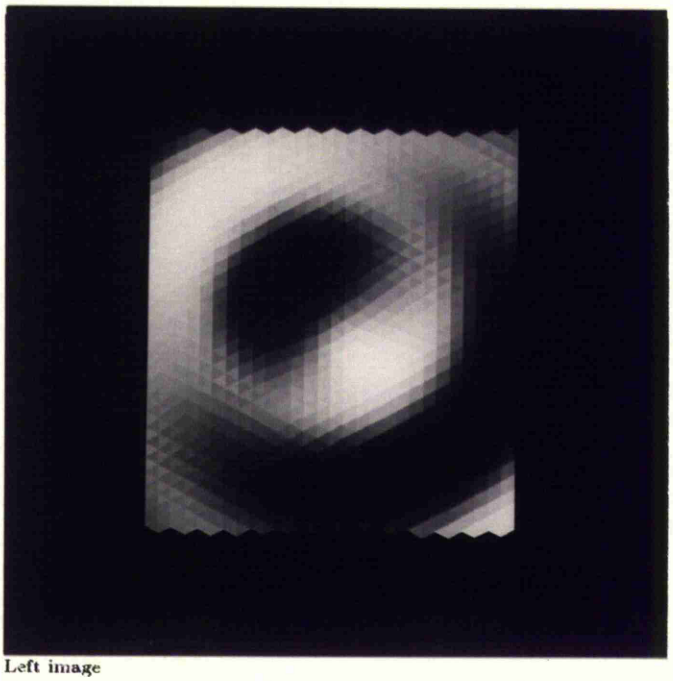


Plate 2.6: A wire-mesh version of Plate 2.5.

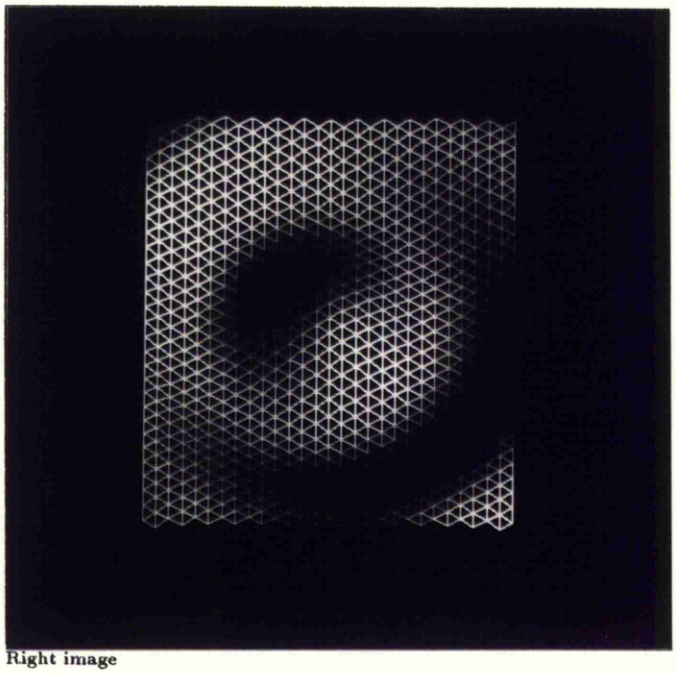


Right image

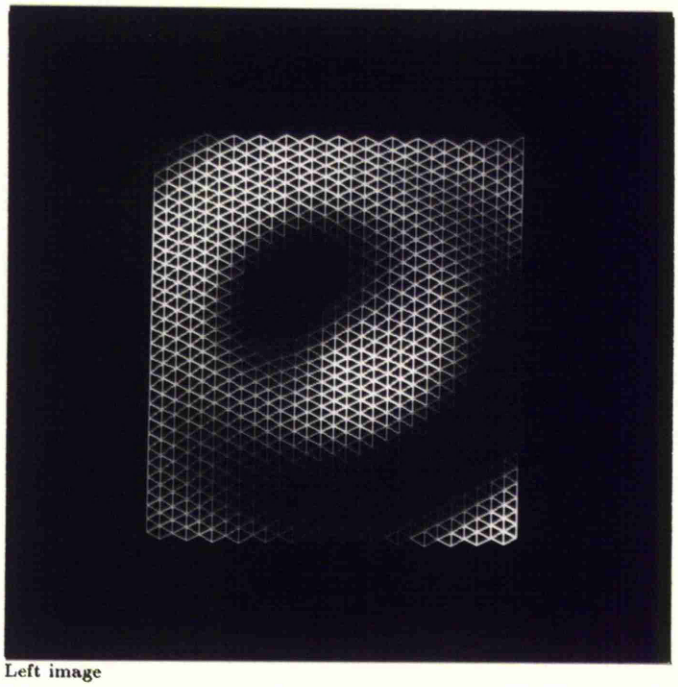


Left image

Plate 2.7: A stereo pair of images, generated using the GRAFIX package.



Right image



Left image

Plate 2.8: A wire-mesh version of Plate 2.7.

increase in spatial resolution may also be necessary. It is expected that the addition of these features would increase the time required to generate each image by a considerable amount. The combination of these factors would appear to make an iterative approximation system for surface modelling impractical, since the computational power required is greatly in excess of that provided by the Magiscan 2.

"Look here upon this picture, and on this;
The counterfeit presentment of two brothers."

Hamlet, in 'Hamlet, Prince of Denmark'
— William Shakespeare

Chapter 3:

A Program for Endothelial Cell Measurement

3.1 Introduction

This chapter is concerned with the use of the Magiscan 2 image analyser for the automatic counting of photomicrographed human corneal endothelial cells. These are low-contrast images, with high noise levels, and require more complex techniques than conventional methods of cell detection [21, 120, 122, 123].

The cornea of the eye consists of six layers: the epithelium (outermost), the basement membrane, Bowman's layer, the stroma, Descemet's membrane, and the endothelium [124, 57, 64]. The endothelium consists of a single layer of cells, typically 300 000 to 500 000 cells per cornea [125]. Its purpose is to maintain the transparency of the cornea, by pumping fluid so as to maintain the cornea in a deturgescent (fluid pressurised) state. These cells do not regenerate, and their total number declines during the lifetime of the eye [126, 127, 128, 129, 130]. The cell density does not normally vary significantly across the cornea [128, 129, 130]. Cell density varies considerably between individuals, but pairs of eyes have similar densities [129].

In the course of eye surgery, extensive endothelial cell loss may occur. This is not only due to corneal incisions, but also to the manipulation of the cornea, which is necessary in the course of surgery. The endothelial cells are very sensitive to mechanical damage and, when this occurs, cells surrounding the damaged portion migrate to repopulate the damaged area [131, 130]. This results in an area of wider, flatter cells. This process appears to continue only until an unbroken layer of cells is formed, and this results in varying cell densities across the cornea, in post-operative eyes [130]. The loss of cells may be up to 70% in some cases [127, 132]. This means that any gain in vision quality from the surgical operation may be counteracted by a loss due to endothelial cell sparsity. The cell density at which visual loss commences varies greatly between individuals, since this depends on cell efficiency. Clearly, however, such a high loss is not desirable. In order to change the surgical procedure to minimise this loss, it is necessary to have some method of accurate evaluation of the endothelial cell damage caused by an operation.

Endothelial cells may be viewed using a contact specular microscope [133, 134, 126, 135, 132] or a reticule and slitlamp [136]. Although the latter is cheaper and easier to use, it is necessary to use photomicrographs for cell measurement, and these are more easily taken from a specular microscope. Cell counting and measuring by hand is an extremely time-consuming process. It is, therefore, desirable to automate the process. The purpose of this part of the project was to use the Magiscan 2 image analyser to detect cells automatically with minimum user intervention, produce statistics, and maintain a disc-based filing system.

Photomicrographs were obtained from the Manchester Royal Eye Hospital (MREH), where a Konan-Pocklington wide-field spectral microscope camera is used to photomicrograph the endothelial cells before, and at regular intervals after, surgery [132]. Part of a photomicrograph, which was selected for processing, is shown in Plate 3.1. This represents about one tenth of the area of the entire photomicrograph. The number of cells per unit area, and other statistics, were already being measured from similar frames, using an analysis program [137] which required an operator to delineate cells manually. The system was based on the Magiscan 1 image analyser, but similar programs have been developed elsewhere. Shaw et al. [138, 139] used a Bausch and Lomb Omnicon image analyser to automatically count and measure acetate tracings of cell photomicrographs, and Jacobi and Strobel [140], used a Leitz ASM digitising tablet to enter cell vertices into a computer, which then calculated cell areas, circumferences, cell densities and circumference to area ratios.

Lester et al. [141] have developed a program which successfully detects and measures cells from photomicrographs of corneas of enucleated rabbit eyes. However, these photomicrographs are of much better quality than the MREH photomicrographs since:

i) The subject eye was a healthy juvenile (rabbit), and not an eye which, we may assume by its very prescence at MREH, has some form of defect, which may adversely affect the quality of the photomicrograph.

ii) The MREH photomicrographs are taken in vivo, whereas enucleated eyes may be relatively easily photomicrographed.

Lester's photomicrographs are of a much smaller area of the cornea; of the order of 20 cells per frame,

Plate 3.1: The unprocessed endothelial cell image.

as opposed to typically 200 cells per frame in the digitised MREH photomicrographs. The program uses a heuristic boundary tracing algorithm [142, 143] to trace the cell boundaries.

3.2 An Overview of the ENDO Cell Analysis Program

The ENDO program was developed on the Magiscan 2 to automatically detect and measure endothelial cells from the MREH photomicrographs. The analysis may be divided into four phases.

i) Cell boundary detection

The object of this phase is to find cells, by detecting the dark boundaries between them, and involves pre-processing and segmentation. Pre-processing is a general term for the manipulation of the grey-level image, to enhance some desired feature before it may be extracted. In this case, the cell walls must be enhanced. Segmentation is the derivation of a binary image from this processed grey-level image.

ii) Cell extraction

The segmented binary image may include many regions which do not represent cells. These must be identified, and discarded.

iii) Manual editing

To provide a clinically usable system, the operator must be able to correct any errors made in the automatic detection and classification processes. This may be performed by manually editing the images of extracted cells.

iv) Clinical measurement

The operator-approved cells may be measured, and clinical statistics derived from these measurements.

Algorithms to implement these four sections are now discussed, and the detailed structure of the ENDO program is described.

3.3 Cell Boundary Detection

A number of methods were considered for cell boundary detection, and some of these are now discussed.

3.3.1 Boundary Following

One method of boundary detection is that described by Lester et al. [143, 22, 141], where cell boundaries are followed from some starting point, branching where necessary, until all boundary points have been located. This requires that the program be capable of locating suitable start points, or that these be provided manually by the operator.

This approach was not adopted in the ENDO program, as the results given by Lester et al., taken for a small number of cells (typically fifteen), required run times of five minutes. Since the MREH photomicrographs contain many more cells, the execution time is likely to be longer, even though the Magiscan 2 may be expected to show better performance than the Data General Nova 840 used by Lester. In addition, the poor quality of the MREH photomicrographs could seriously affect the heuristic boundary following process, as such methods are prone to failure when confronted with indistinct boundaries.

3.3.2 Edge Gradient Methods

An alternative to boundary following is to perform a series of local operations to attempt to extract all boundary points without the need to locate suitable start points. Several edge gradient operators including Roberts, Sobel, Kirsch, and the Manchester line operator [144] were applied to the MREH photomicrograph images. In all cases, the amount of noise present in the images was sufficient to totally obscure any meaningful edges. Smoothing applied before the edge operators reduced this effect, but also caused considerable loss of true cell boundaries.

3.3.3 The Difference-of-Gaussians Operator

The Marr-Hildreth difference-of-Gaussians operator, or DOG filter, followed by a zero crossing detector [145, 112], is another possible method of detecting edges in images. This operator discards the

low-frequency and high-frequency components of an image, leaving only the information in a relatively narrow pass-band. Since background variations are low-frequency phenomena, and camera noise and film grain are predominantly high-frequency, convolution with this type of filter is useful for many applications. The filter itself is the difference of two functions, a low-pass Gaussian filter with a low cutoff frequency, g_l, and a low-pass Gaussian filter with a higher cutoff frequency, g_h, where

$$g_l(x,y) = \mathrm{e}^{-(x^2+y^2)/\sigma_l^2}$$

$$g_h(x,y) = \mathrm{e}^{-(x^2+y^2)/\sigma_h^2}$$

To successfully detect edges, the size of the filters must be matched to the scale of the edges concerned, by selection of appropriate values of σ_l and σ_h. Marr and Hildreth [145] recommend a ratio of $1:1.6$ for these values, for good rejection of frequencies outside the pass-band.

The DOG filter may be implemented in the spatial domain in a number of ways. One straightforward method is to form a single weight matrix composed of the difference of the two Gaussian functions, g_h and g_l, and to convolve this with the image, f, to give $f \otimes (g_h - g_l)$.

The most computationally efficient implementation appears to be that suggested by Torre and Poggio [146], where the image is convolved with a horizontal vector h, and then with a vertical vector v, where

$$h \otimes v = g_h - g_l,$$

and so

$$\begin{aligned}(f \otimes h) \otimes v &= f \otimes (h \otimes v) \\ &= f \otimes (g_h - g_l).\end{aligned}$$

The disadvantage of this method is that the intermediate image, $f \otimes h$, must be stored at a high grey-level resolution, otherwise appreciable rounding errors will result.

A third method is to convolve the source image separately with the two Gaussian functions, and then to subtract the two resulting images, since

$$f \otimes g_h - f \otimes g_l = f \otimes (g_h - g_l).$$

This involves two image convolution operations, and requires temporary storage of the intermediate result image.

A fourth alternative is to filter the source image using a high-cutoff low-pass Gaussian filter, g_l, followed by a low-cutoff high-pass filter, $\delta - g_l'$, where δ is the unit impulse function and g_l' is selected such that $g_l' \otimes g_h = g_l$. In this case

$$f \otimes (\delta - g_l') = f \otimes (\delta \otimes g_h - g_l' \otimes g_h)$$

$$= f \otimes (g_h - g_l).$$

Again, this requires two image convolutions, but does not require additional storage.

The third of these methods was used to implement the DOG filter on the Magiscan 2, since this allows the intermediate filtered images to be observed, and the sequence of operators to be varied more easily. The results of filtering the endothelial cell image (Plate 3.1) with Gaussian low-pass filters, for which $\sigma = 3.2$pixels and $\sigma_h = 5.2$pixels, are shown in Plates 3.2 and 3.3. These values are in a ratio of approximately $1:1.6$, as recommended by Marr and Hildreth [145]. Plate 3.4 shows the difference between these images (contrast expanded), and Plate 3.5 shows the zero-crossings. Although some cells may be distinguished in this image, too many unwanted edges are detected for this image to be usable. The effect of using smaller filter sizes may be seen in Plate 3.6, where $\sigma_h = 2.6$ and $\sigma_l = 4.1$. In this image, the high noise level obscures almost all of the cell details. Plate 3.7 show the effect of using pairs of larger of σ values. Here, $\sigma_h = 4.1$ and $\sigma_l = 5.2$. It may be seen that the detected edges are displaced appreciably from the true edge positions, and many cells boundaries are lost entirely. Smaller ratios between the filter sizes were tried, and Plates 3.8 and 3.9 show the zero crossings for $\sigma_l = 3.2$ and $\sigma_h = 4.1$, and $\sigma_l = 4.1$ and $\sigma_h = 5.2$; a ratio of about $1:\sqrt{1.6}$. It may be seen that the smaller of these (Plate 3.8) still shows excessive noise levels, and in the larger (Plate 3.9), many boundaries are incomplete.

From these results, it seems that any DOG filter which removes a sufficient amount of noise from the image will also destroy parts of the cell boundaries. The use of the frequency spectrum to distinguish between noise and feature appears to be insufficient. A non-linear technique was therefore investigated.

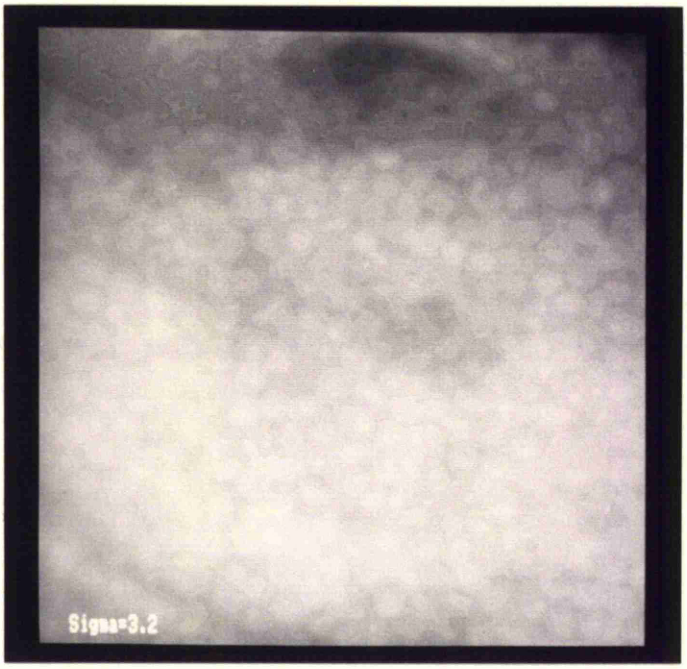


Plate 3.2: The endothelial cell image, after Gaussian filtering with $\sigma = 3.2$.

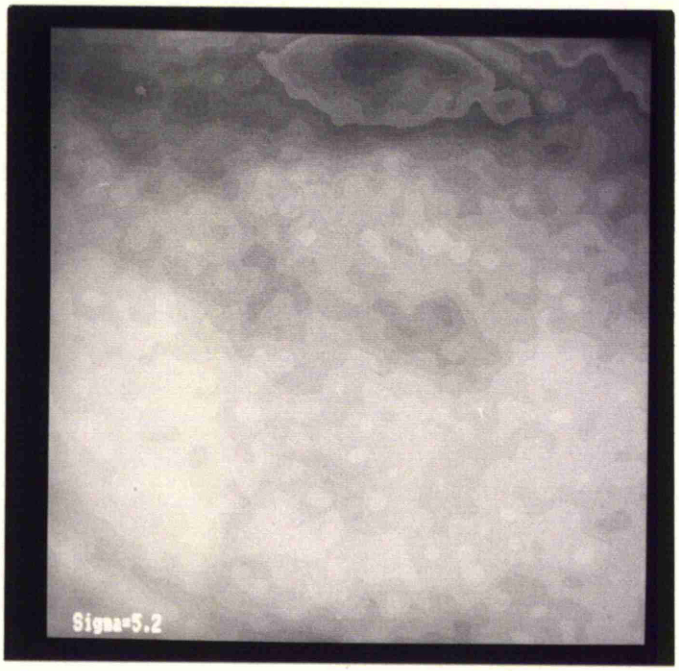


Plate 3.3: The endothelial cell image, after Gaussian filtering with $\sigma = 5.2$.

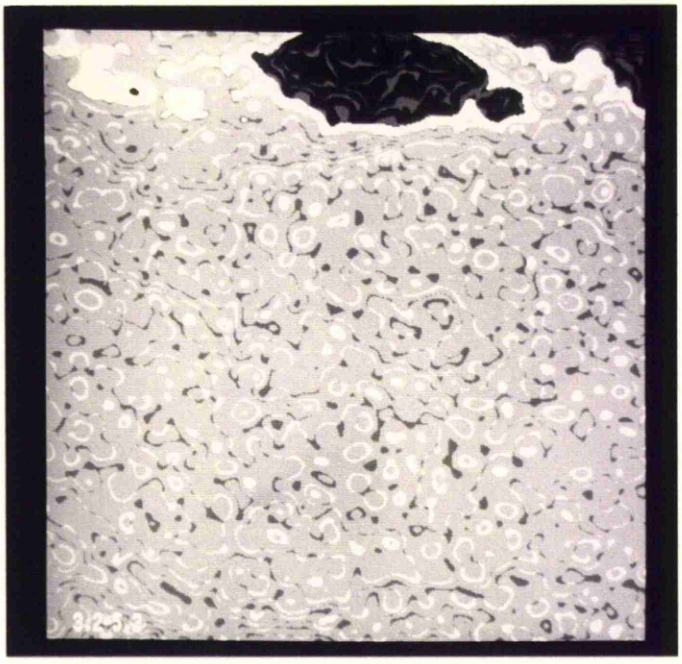


Plate 3.4: Difference-of-Gaussians for $\sigma_h = 3.2$ and $\sigma_l = 5.2$ (contrast expanded).

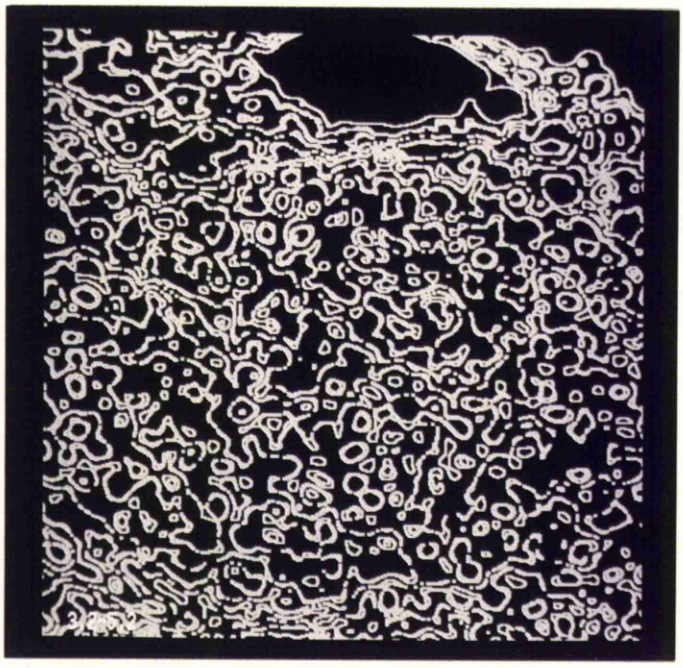


Plate 3.5: Zero crossings of Plate 3.4.

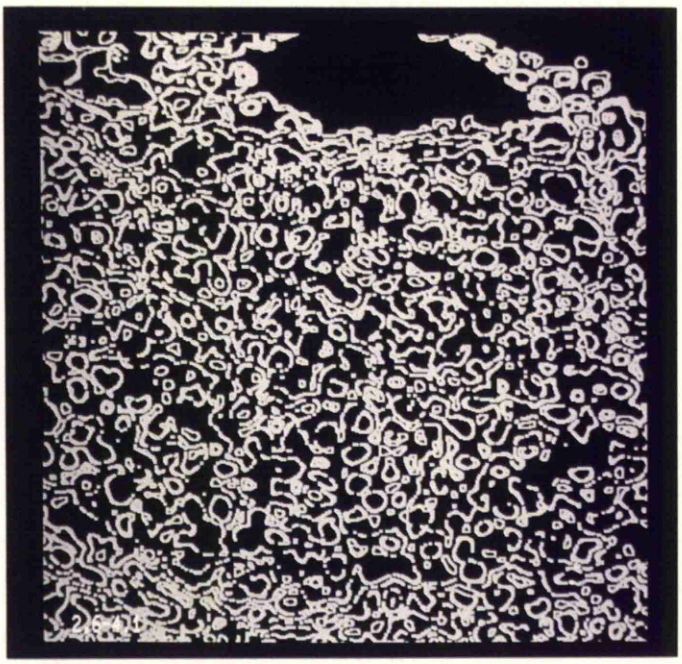


Plate 3.6: Zero crossings from difference-of-Gaussians with $\sigma_h = 2.6$ and $\sigma_l = 4.1$.

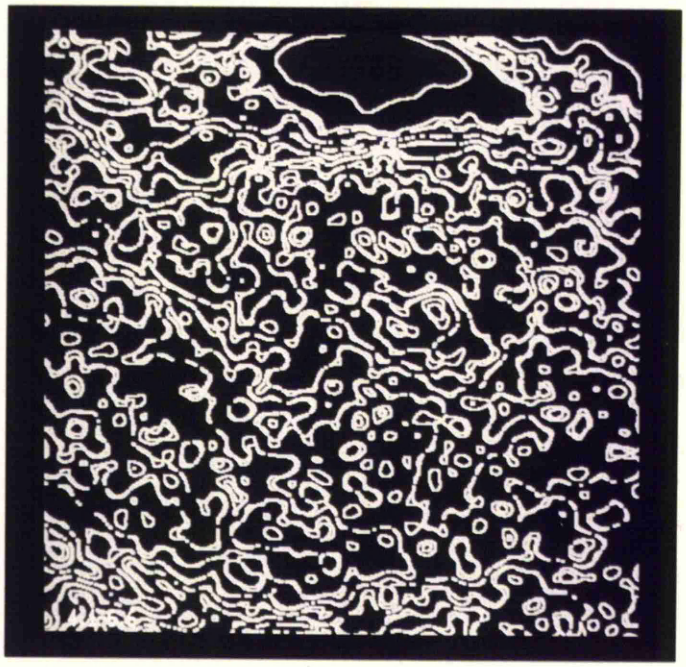


Plate 3.7: Zero crossings from difference-of-Gaussians with $\sigma_h = 4.1$ and $\sigma_l = 6.6$.

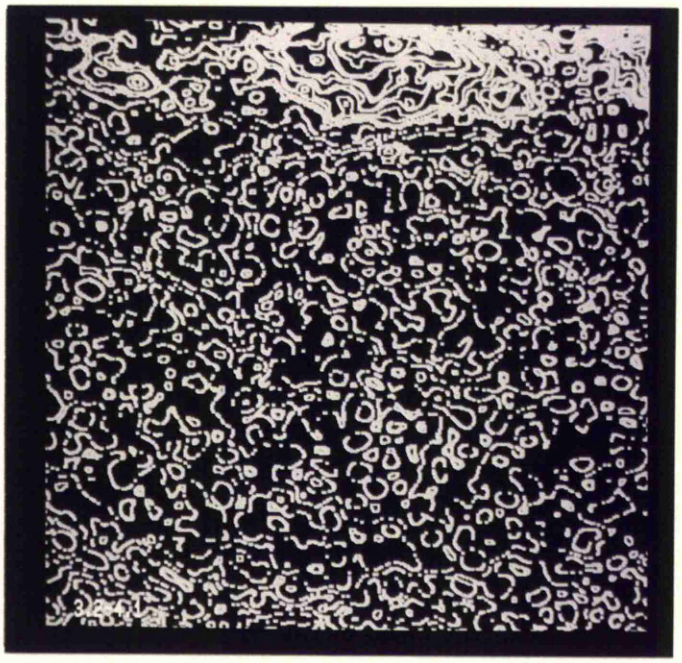


Plate 3.8: Zero crossings from difference-of-Gaussians with $\sigma_h = 3.2$ and $\sigma_l = 4.1$.

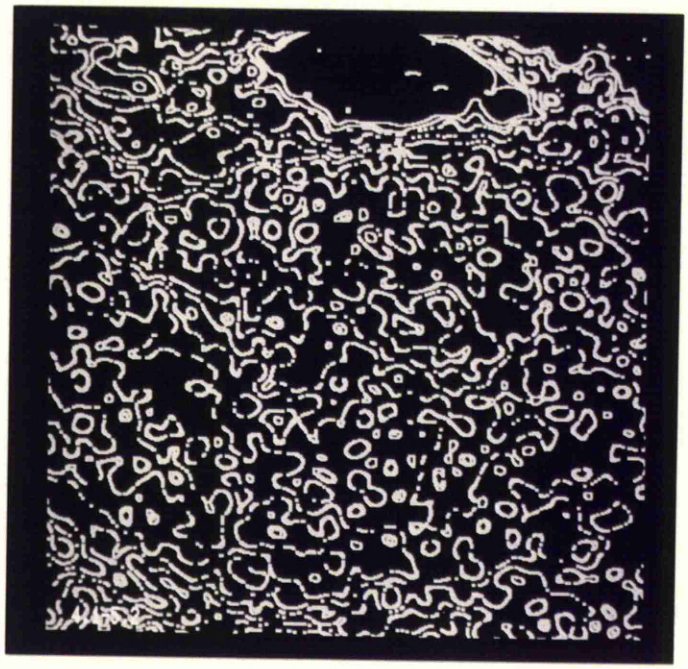


Plate 3.9: Zero crossings from difference-of-Gaussians with $\sigma_h = 4.1$ and $\sigma_l = 5.2$.

3.3.4 A Modified Difference-of-Gaussians Operator

The low-frequency low-pass, or wide-window, part of the DOG filter is intended to generate a background function, to be subtracted from the original image. This should have no trace of the desired features, in this case, the dark cell walls. In this way, background variations may be removed, leaving all of the cell walls remaining. This is evidently not the case in Plate 3.3 and parts of the cell walls are detectable as darker regions in the background function. This means that after the subtraction operation, parts of the cell walls will be reduced in strength.

To remove all traces of the cell boundaries from the background function, the low-frequency low-pass filter may be preceded by a non-linear line-filling operation, described by Lester et al. [22] for other purposes. This removes the dark cell walls, so that they cannot affect the background function. This dark line filler operates on a 3×3 neighbourhood, and examines pairs of opposing pixels to see if both are greater than the centre pixel. If one or more such pairs are found, the mean of each pair is calculated, and the centre pixel is replaced by the maximum of these mean values. In practice, a number of passes are required to totally remove all dark lines.

The ENDO program uses a 9×9 median filter (Plate 3.10), and three passes of the dark line filling operator (Plate 3.11), prior to wide-window filtering, to generate the background function. The median operation is applied to remove 'salt and pepper' noise from the image, before the dark line filling operator is applied. The wide-window filtering used in the ENDO program is not convolution with a Gaussian, but a simple mean averaging operator, taken over a local neighbourhood. This avoids the many multiplications required in the convolution operation, and permits incremental calculation of each pixel value, based on the new value calculated for the preceding adjacent pixel.

An operator which calculated an average over a 25×25 square was investigated, but this tended to introduce horizontal and vertical artifacts in the result images, due to its anisotropic point spread function. This was replaced with a 25×25 octagonal operator, which appears to reduce this problem (Plate 3.12). A circular operator would be expected to perform slightly better than this, but would have been difficult to implement in microcode with the same efficiency as the optimised incremental operators. The octagonal operator required only two minutes, as opposed to an estimated seven minutes for a 25×25 microcoded convolution.

Plate 3.10: The endothelial cell image, after 9×9 median filtering.

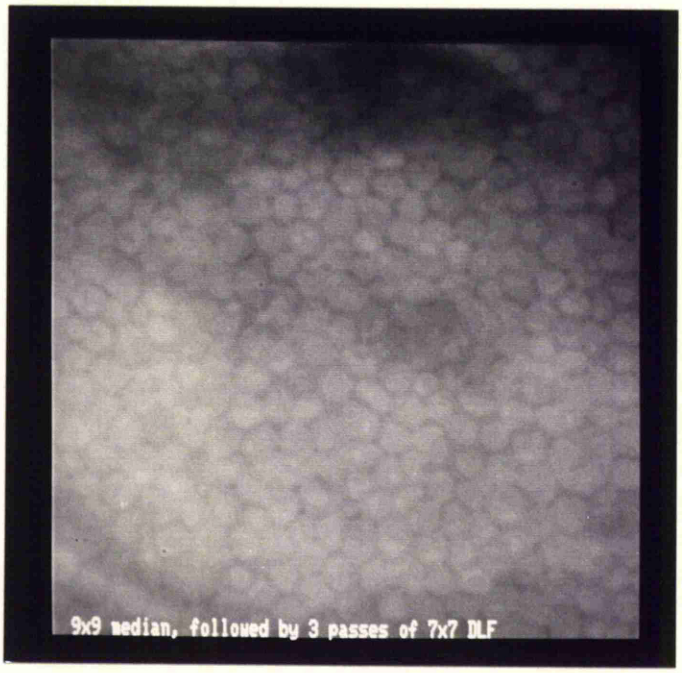


Plate 3.11: Plate 3.10, after three passes of the 7×7 dark line filling operator.

Plate 3.12: The background function derived from the endothelial cell image.

The background function is subtracted from the original image, with a constant grey-level displacement to prevent negative result values (Plate 3.13). This image still contains a large amount of high-frequency noise, and many techniques were tried to remove this without degrading the cell boundaries [147]. The best sequence found was a modification of a sequence suggested by Lester et al. for processing images of white blood cells [22]. This is a 7×7 median operator, followed by a 3×3 extremum operator and a 3×3 median. The extremum operator is a non-linear local operator in which each pixel is replaced by the local maximum or local minimum of the neighbourhood, depending on which is closest to the original pixel value. This operator has the effect of sharpening edges in grey-scale images. The median-extremum-median sequence results in some degradation of cell boundaries (Plate 3.14), but this appears to be inevitable.

3.3.5 Segmentation

The pre-processed image is thresholded at a level corresponding to the grey-level displacement added during the background subtraction process. This yields an image of partially incomplete cell boundaries (Plate 3.15), which is skeletonised by nine passes of a thinning operator (Plate 3.16). All true cell boundaries are reduced to a single-pixel width line by this operation. The resulting image is dilated (Plate 3.17) and negated to give an image of cellular regions (Plate 3.18). Regions which touch the outer boundary of the image are removed at this stage, to leave a binary image of regions for classification and measurement.

3.3.6 Results of Segmentation

The binary image obtained after pre-processing and segmentation contains a large number of regions, which fall into the following three categories.

i) Single-cell regions

These regions represent individual cells which have been successfully segmented (Plate 3.19).

ii) Multiple-cell regions

These regions represent groups of cells which remain connected together in clusters of two or more, due to gaps in the detected cell boundaries (Plate 3.20).

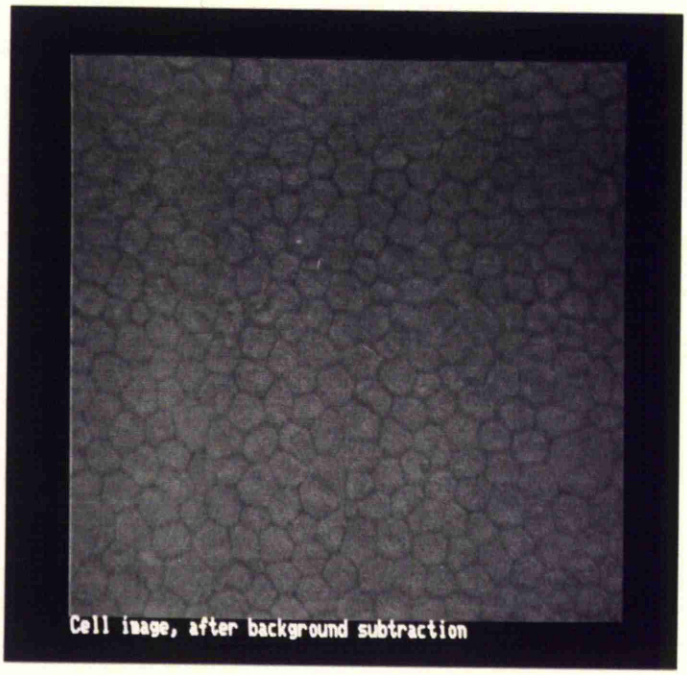


Plate 3.13: The endothelial cell image after background subtraction.

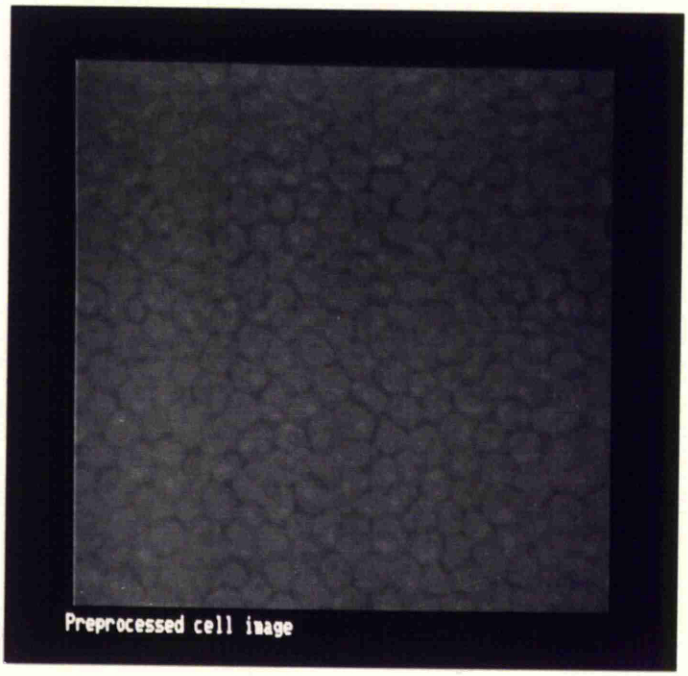


Plate 3.14: The pre-processed endothelial cell image.

Plate 3.15: The thresholded cell boundaries.

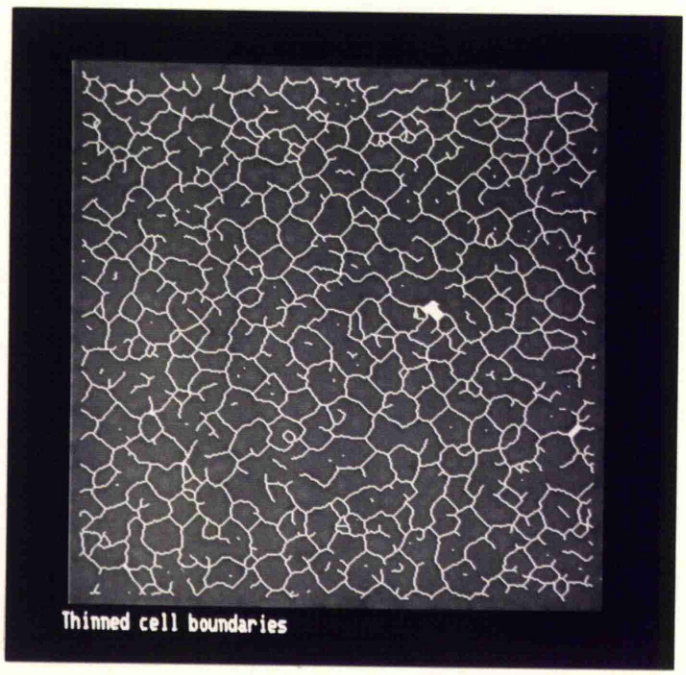


Plate 3.16: The thinned cell boundaries.

Plate 3.17: The dilated cell boundaries.

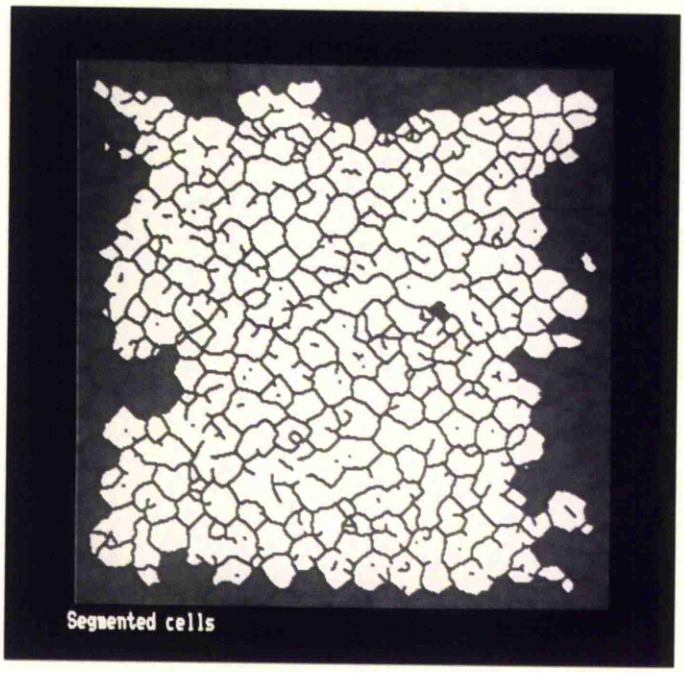


Plate 3.18: The segmented regions.

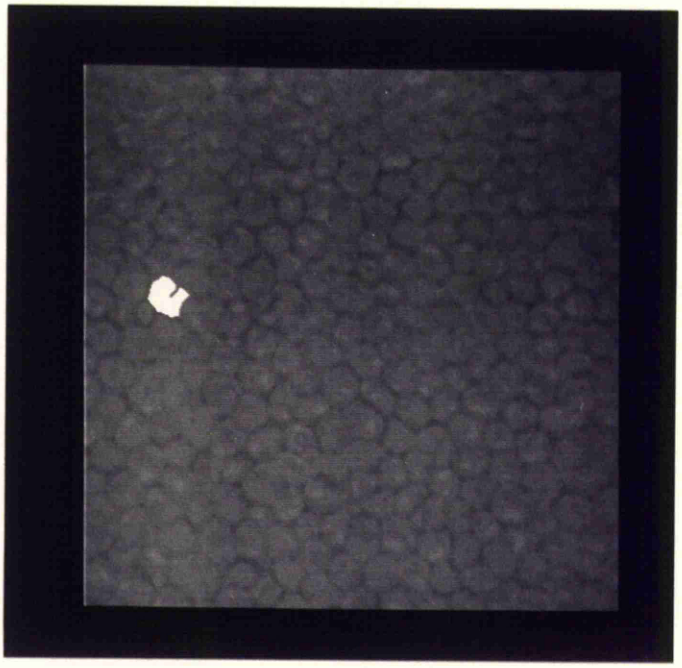


Plate 3.19: A single-cell region.

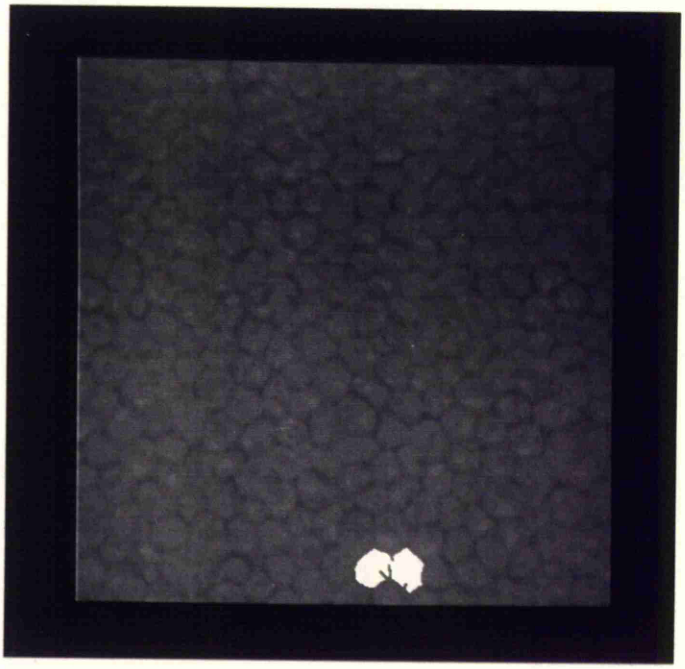


Plate 3.20: A multiple-cell region.

iii) Non-cellular regions

These fall into three sub-groups:

a) Small specks of noise, of one or two pixels in area.

b) Large regions, where insufficient contrast existed for correct segmentation.

c) Parts of cells, which have become fragmented due to spurious dark lines across them.

These classes must be sorted, and appropriate action taken in each case. Single-cell regions must be measured, and statistics stored; multiple-cell regions and non-cellular regions may be manually edited or discarded. To sort the regions, it is necessary to measure, and thus classify, each of them.

3.4 Cell Extraction

Before the detected regions may be sorted, and genuine cells extracted, it is necessary to make some measurements of the segmented regions. These are purely for classification purposes, and are distinct from the clinical measurements to be made at the end of the analysis.

3.4.1 Measurements for Classification

A number of simple measurements were used in the ENDO program, for the purpose of classification:

i) Area measurement

The area of each cell is easily calculated from its representation as a list of scan-line chords, and is simply the sum of the length of all chords. A system microcode routine exists to perform this measurement.

ii) Circularity measurement

The circularity is defined here to be the perimeter of the region divided by the square root of its area. This gives a measurement of shape, independent of feature size.

iii) Aspect ratio measurement

The aspect ratio of a region is the ratio of its length to its breadth, where length is defined as the longest dimension, and breadth is the width of the projection of the region onto an axis perpendicular to the length measurement.

Typically, single-cell regions have a fairly small area, low circularity, and low aspect ratio. Multiple-cell regions normally have a larger area, higher circularity, and higher aspect ratio. Non-cellular regions tend to have very high or very low areas, high circularity, and high aspect ratios. Frequently, notches occur in single-cell regions, due to dark regions in the centres of cells becoming attached to the boundary, causing a slot in the region, as seen in Plate 3.19. This causes the circularity of these regions to be artificially high. This is counteracted by taking one set of measurements, performing a binary closing operation, then taking a second set of measurements. This is seen in Plates 3.21 and 3.22. Six parameters for each region are thus found, and these are used to classify the region.

3.4.2 Methods of Cell Identification

A study of manually classified regions revealed large overlaps between the area, circularity and aspect ratio measured for single-cell and multiple-cell regions, and some overlaps with non-cellular regions. This applies both before and after the closing operation, and may be seen in Tables 3.1, 3.2 and 3.3. In spite of this overlap in their distribution, there appears to be sufficient information in these parameters to classify any particular region, on the basis of at least one of its measurements.

This information could be extracted using one of several standard statistical pattern recognition methods, such as linear piecewise classification [113], or Bayesian statistical analysis [148, 149]. These methods have the advantage that such systems are trainable, and are much more flexible than a fixed parameter thresholding scheme. A Bayesian classifier was used in the ENDO program, but a linear piecewise classifier would also appear to be suitable.

3.4.3 Bayesian Statistics

Bayesian decision theory is an established method of extracting useful classification data from experimental results [148, 149, 150, 151]. The basis of the method is that the probability, $p(D|S)$, of some condition D, given some observation S, may be expressed as

$$p(D|S) = \frac{p(D)p(S|D)}{p(S)},$$

where $p(D)$ is the a priori probability of D, $p(S)$ is the a priori probability of S, and $p(S|D)$ is the probability of the observation, S, given the condition D. The conditional probability, $p(S|D)$, may be

Plate 3.21: The single-cell region, after binary closing.

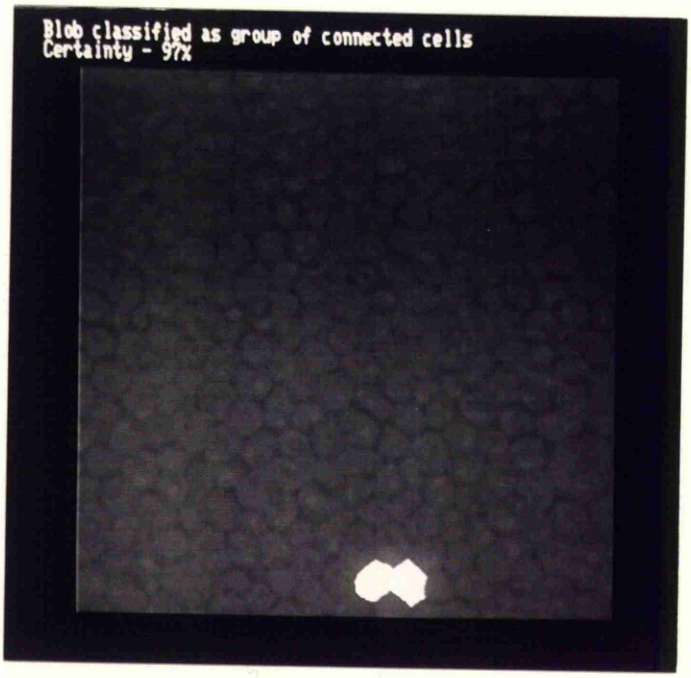


Plate 3.22: The multiple-cell region, after binary closing.

Table 3.1
Areas of Manually-Classified Regions.

Area (microns^2)	Unsmoothed			Smoothed		
	Single-Cell	Multiple-Cell	Non-Cellular	Single-Cell	Multiple-Cell	Non-Cellular
0 – 100	1	0	125	1	0	125
100 – 200	37	0	49	34	0	47
200 – 300	97	0	23	97	0	25
300 – 400	60	4	25	56	2	19
400 – 500	38	5	8	40	4	14
500 – 600	18	11	10	17	12	7
600 – 700	6	9	5	12	7	5
700 – 800	2	12	9	0	11	9
800 – 900	1	7	1	2	6	2
900 – 1000	0	2	0	1	7	2
1000 – 1500	0	14	1	0	14	1
1500 – 2000	0	3	1	0	4	1
2000 – ∞	1	5	1	1	5	1

Table 3.2
Circularities of Manually-Classified Regions.

Circularity	Unsmoothed			Smoothed		
	Single-Cell	Multiple-Cell	Non-Cellular	Single-Cell	Multiple-Cell	Non-Cellular
0.0 – 3.5	0	0	10	0	0	10
3.5 – 4.0	138	0	78	193	0	79
4.0 – 4.5	63	0	86	66	14	105
4.5 – 5.0	26	10	29	1	36	38
5.0 – 5.5	15	18	15	0	17	12
5.5 – 6.0	9	12	16	1	3	7
6.0 – 6.5	8	14	8	0	1	1
6.5 – 7.0	1	4	4	0	1	0
7.0 – 7.5	0	4	5	0	0	0
7.5 – 8.0	0	4	0	0	0	0
8.0 – 8.5	0	0	0	0	0	0
8.5 – ∞	1	6	1	0	0	0

Table 3.3
Aspect Ratios of Manually-Classified Regions.

Aspect Ratio	Unsmoothed			Smoothed		
	Single-Cell	Multiple-Cell	Non-Cellular	Single-Cell	Multiple-Cell	Non-Cellular
1.0 – 1.2	130	2	40	131	1	41
1.2 – 1.4	97	4	59	96	5	58
1.4 – 1.6	24	18	56	24	19	56
1.6 – 1.8	7	20	44	7	20	44
1.8 – 2.0	2	12	24	2	11	24
2.0 – 2.2	0	9	13	0	9	13
2.2 – 2.4	1	6	6	1	6	6
2.4 – ∞	0	1	13	0	1	13

determined by observing many cases in which the condition D is true, and noting whether S is true or not. Similarly, the probabilities $p(S)$ and $p(D)$ may be observed. It is possible to combine the results of several different observations. It may be shown that the probability of the condition D, given a set of observations, $S_1, \ldots S_n$, is

$$p(D|S_1, \ldots S_n) = \frac{p(D) \prod_{\forall i \in T} \Big(p(S_i|D)\Big) \prod_{\forall i \in F} \Big(1 - p(S_i|D)\Big)}{\text{Numerator} + \Big(1 - p(D)\Big) \prod_{\forall i \in T} \Big(p(S_i|\overline{D})\Big) \prod_{\forall i \in F} \Big(1 - p(S_i|\overline{D})\Big)},$$

where T is the set of integers, i, such that S_i is true, F is the set of integers, i, such that S_i is false, and $p(S_i|\overline{D})$ is the probability of the observation S_i, in the absence of the condition D.

The Bayesian classifier deals with information which is represented as boolean values; either true or false. It is not suited to deal directly with classification of continuous measurements, such as area, circularity and aspect ratio. For these to be used in classification, the range of each continuous measurement is divided into sections, and a corresponding number of boolean measurements are defined such that the boolean is true, only if the continuous measurement is in the specified range for that boolean. In the ENDO program, area, circularity and aspect ratio are divided the ranges already seen in Tables 3.1, 3.2 and 3.3.

Each region is taken in turn and the conditional probability that it belongs to each of the three region classes (single-cell, multiple-cell or non-cellular) is assessed. The region is assigned to the class which gives the greatest conditional probability. The data shown in Tables 3.1, 3.2 and 3.3 was used as

a training set for the classifier, which was used to classify the regions shown in Plate 3.18. The three resulting images may be seen in Plates 3.23, 3.24 and 3.25.

3.5 Manual Editing

The segmented regions are placed into one of three images, depending on the decision of the Bayesian classifier. These three images are presented to the operator, who may the edit the images using a light pen. The available operations are:

i) Move objects from one image to another.

ii) Cut apart multiple-cell regions, into single-cell regions.

iii) Join together fragmented cells.

iv) Draw completely new regions.

During editing, the cell regions are displayed as an overlay over the original grey-level image, so that the operator may assess the accuracy of the segmentation and classification, and manually trace missing boundaries. Plate 3.26 shows the multiple-cell regions after manual division, and Plate 3.27 shows the final manually-edited image of single-cell regions. After editing, the single-cell regions are smoothed by a binary closing operation and all three images are, again, presented to the operator for a second session of editing (Plate 3.28). This allows any mistakes, which may have been overlooked in the first session, to be corrected. On completion of the second editing session, the cells are measured to give clinically-useful statistics.

3.6 Clinical Measurements

The measurements of the cells which are made to give clinical data, for patient assessment, are as follows:

i) Mean cell area.

ii) Standard deviation of cell area.

iii) Mean cell circularity.

iv) Standard deviation of cell circularity.

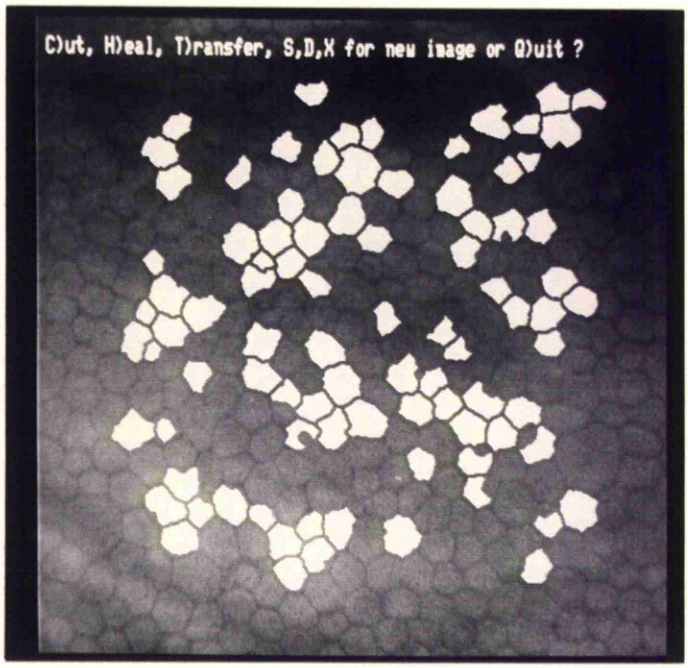


Plate 3.23: Regions classified as single-cell regions.

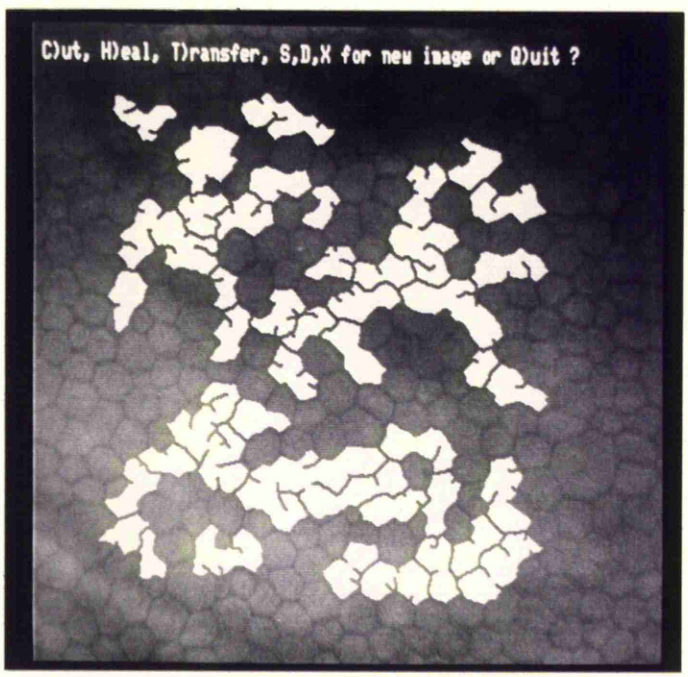


Plate 3.24: Regions classified as multiple-cell regions.

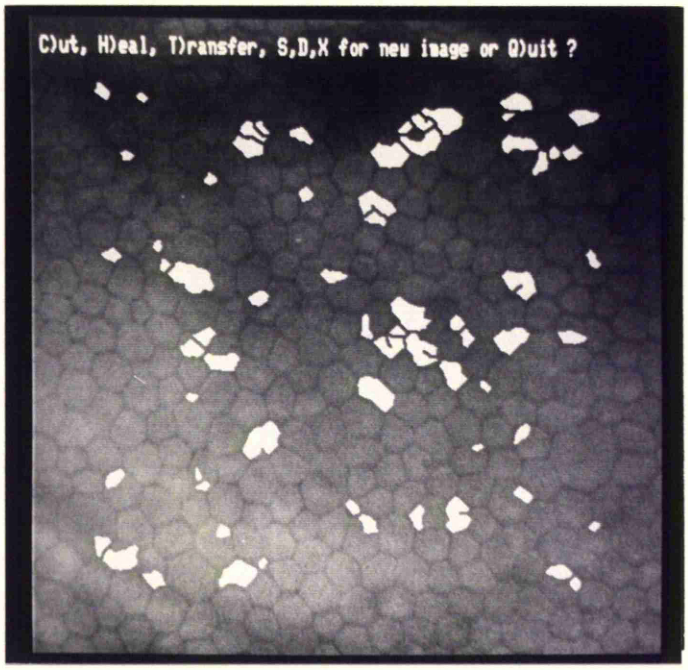


Plate 3.25: Regions classified as non-cellular regions.

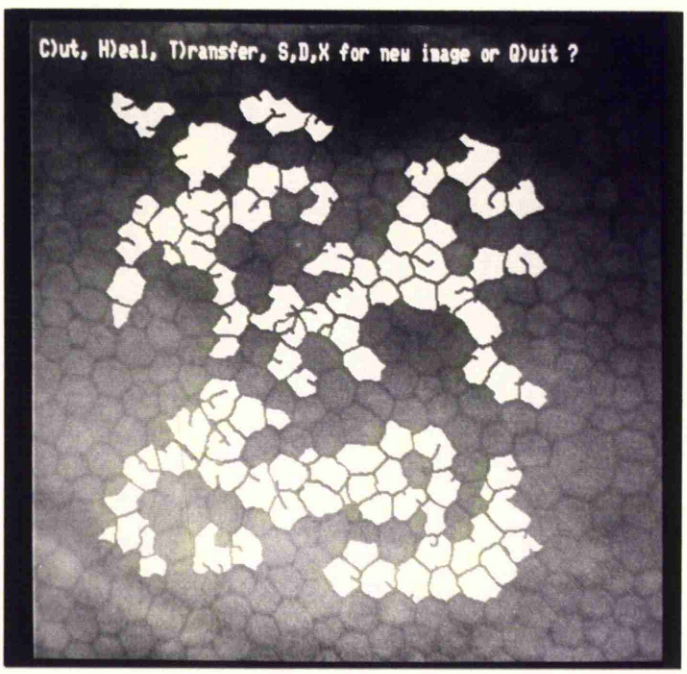


Plate 3.26: The multiple-cell regions, after manual separation.

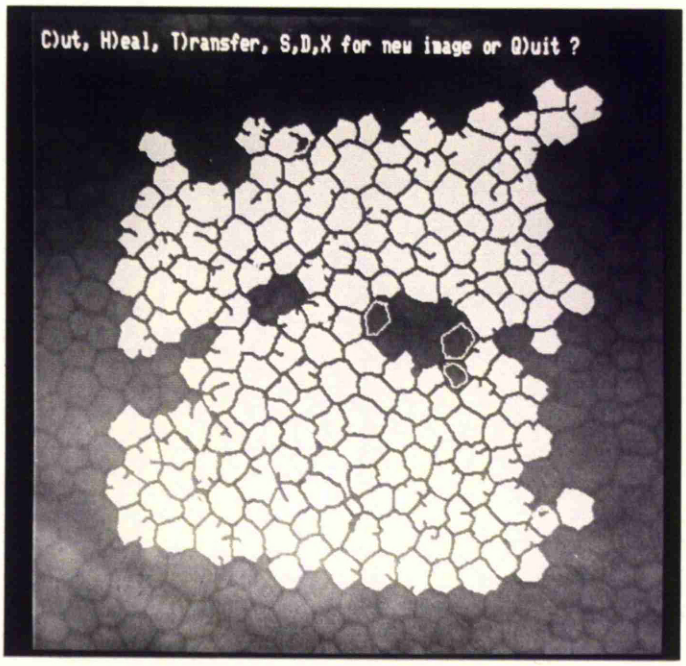


Plate 3.27: The manually-edited cell image.

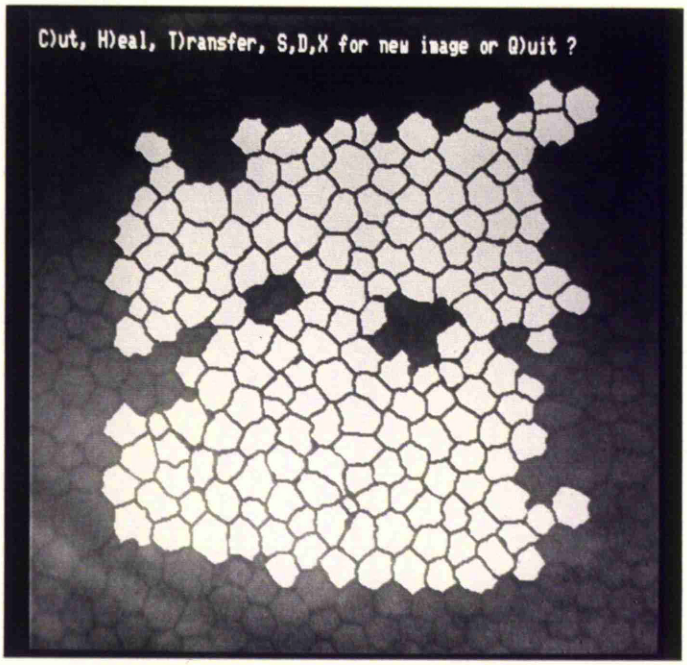


Plate 3.28: The manually-edited cell image, after smoothing.

v) Mean cell aspect ratio.

vi) Standard deviation of cell aspect ratio.

vii) Mean cell orientation.

viii) Standard deviation of cell orientation.

ix) Cell density, derived from mean cell area.

These values are thought to be of clinical significance in the assessment of endothelial cell photomicrographs. Changes in cell area are thought to give an indication of the extent of cell migration, and the cell orientation is believed to indicate the direction of migration [132].

The measurements are displayed to the operator (Plate 3.29), stored, and may be printed for each individual frame, or cumulatively for several photomicrographs from the same patient. The measurements of cell area are also presented in graphical form (Plate 3.30).

3.7 Performance of the ENDO Program

Examples of the processed images from several stages of the ENDO program have been seen in this chapter. The performance of each of the processing stages of the program are now discussed, in relation to these results.

3.7.1 Pre-Processing and Segmentation

After pre-processing and segmentation, many cell boundaries were partially lost, or sufficiently weakened to join two or more cells. The pre-processing sequence was varied in an attempt to improve this, but without success. In many cases, detailed examination of the original photomicrograph revealed breaks in the cell boundaries. These are not, however, thought to represent actual breaks in the endothelial cell walls. The time taken for the pre-processing and segmentation phases was approximately six minutes.

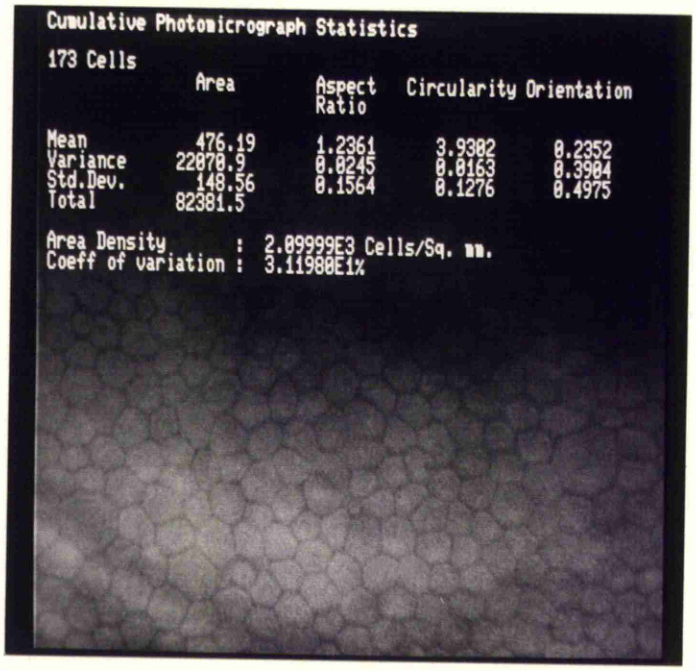


Plate 3.29: Clinical statistics for the detected cells.

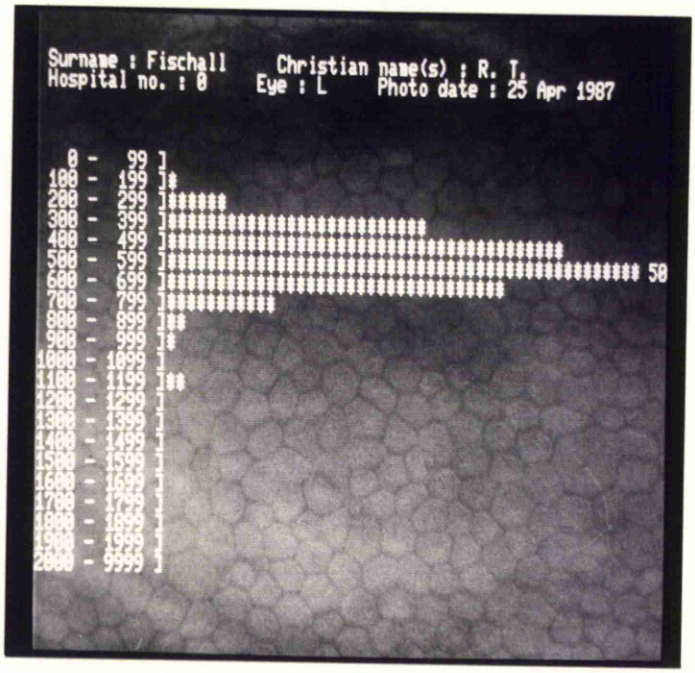


Plate 3.30: A histogram of measured cell areas.

3.7.2 Bayesian Classifier

The performance of the classifier was found to be good, and even with the relatively small training set shown earlier in Tables 3.1 to 3.3, the automatic classification was reliable, as may be seen from Plates 3.23, 3.24 and 3.25. The time taken to perform the automatic classification was about four minutes.

3.7.3 Manual Editing

Manual editing is primarily required to cut apart regions of connected cells, and to fuse together fragmented cells. On the image shown in the plates of this chapter, this took about sixteen minutes, but a trained operator may be expected to reduce this substantially.

3.7.4 Clinical Measurements

Clinical measurements were produced, but no long term studies were made, to compare automated measurement with manual assessment. This was mainly due to machine-time restrictions. The comparisons which were made indicated that the automatic procedure consistently produced cell area measurements which were about 20% smaller than the manual measurements. This may be accounted for by the fact that the cell walls were not measured as part of the cell area in the automatic process. A possible solution to this would be to add some compensating factor based on the perimeter of the cell. The time taken to make the clinical measurements was approximately two minutes.

3.8 Summary

A system for automatic detection and measurement of endothelial cell photomicrographs has been described. The system requires a certain amount of manual intervention to correct segmentation errors, but this time is less than that required to measure an equivalent number of cells manually. The overall processing time for an average frame was about 25 minutes. This is, unfortunately, too long to permit routine clinical use of the program, since the Magiscan 2 machines are in near-constant use for other projects, and no funds were available to purchase another machine specifically for cell analysis use.

"Like all other arts,
the Science of Deduction and Analysis
is one which can only be acquired
by long and patient study,
nor is life long enough to allow any mortal
to attain the highest possible perfection in it."

Sherlock Holmes, in 'A Study in Scarlet'
— Sir Arthur Conan Doyle

Chapter 4:
An Examination of Image Analysis Hardware Requirements

In general, medical images are complex, and require computationally expensive methods of analysis. In both of the ophthalmic applications described in chapters 2 and 3, the program execution time was found to be greater than the tolerable limit for economic operation. From this, it seems that a large increase in processing power is required, to give acceptable processing times for many medical image analysis applications. This increase in system throughput may be attained in one of two possible ways. The first of these is that improvements in technology may be used to construct faster sequential machines. In the past, this has been the normal course of action when a task requires more processing power than has been available at the time. There are, however, theoretical and practical limits to the performance attainable in this way [152, 153, 154, 155, 157].

An alternative approach is to exploit the parallelism which exists in the image processing and image analysis tasks, by using parallel hardware structures. Most parallelism in image analysis tends to fall into two categories: pixel-oriented and feature-oriented. In the case of pixel-oriented parallelism, the same function is applied at every point in the image, and only local pixel values are involved. Feature-oriented parallelism arises when a set of independent operations (not necessarily identical) are to be performed on a set of arbitrarily-placed objects. Here, no locality of processing activity may be assumed.

In the case of the ENDO program, pixel-oriented parallelism exists in the pre-processing and segmentation phase, and feature-oriented parallelism is present in the classification and measurement phases. Sequential execution on the Magiscan 2 requires six minutes of pixel-operations and six minutes of feature-operations. It is important that both pixel-oriented parallelism and feature-oriented parallelism are exploited, since an order of magnitude increase in speed in either of these will only reduce the total processing time to about seven minutes, as the total time becomes limited by the remaining sequential sections.

Most work on parallel image analysis has been concentrated on pixel-oriented parallelism, since it exhibits some characteristics which may simplify the hardware structures required [158]. In calculating a result for any pixel, image accessing is restricted to a local neighbourhood, and this allows an image to be efficiently distributed across the stores of a large number of processors. Such processors may be constructed in such a way as to operate in a lockstep fashion, thus simplifying overall control and synchronisation of the machine.

Feature-oriented parallelism does not exhibit these properties; in general, no simple partitioning of an image will permit a number of processors to access one, and only one, particular feature in the image. From this, it would appear that each processor must be able to access the entire image. Feature-oriented operations do not tend to have the simple structuring of pixel operations, and so lockstep operation is not possible. This leads to complex control and synchronisation problems.

Despite these problems, the need to make use of feature-oriented parallelism exists, and in the following chapters, suitable machine structures are examined, and a parallel machine design is described and simulated.

Is it thy will thy image should keep open
My heavy eyelids to the weary night?

'Sonnet LXI' — William Shakespeare

Chapter 5:

A Taxonomy for Parallel Machine Description

5.1 A Survey of Machine Taxonomies

To compare machine structures, it is necessary to adopt some uniform taxonomy. Such a taxonomy must be detailed enough to express the important differences between machine designs, but must not allow details to obscure the underlying structures. In this chapter, a number of proposed taxonomies are examined, and a new taxonomy based on simple architectural units and interconnection networks is described. A matrix-based representation scheme for interconnection networks is introduced, and a number of matrix operations are defined, which may be used to quantify the properties of interconnection networks. Some interconnection networks are described using this notation, and these are used in the definition of a number of machine classes in chapter 6.

5.1.1 Flynn's Taxonomy

Flynn [159] defined the now-universal SISD, SIMD, MISD and MIMD classes in an attempt to characterise machine structure. These are:

SISD Single Instruction stream — Single Data stream

This class represents the conventional sequential, or von Neumann, processor which executes a single instruction at a time, on data from a single memory unit.

SIMD Single Instruction stream — Multiple Data stream

These machines have multiple data processing units, and each operates on its own data stream. All of these units are controlled by a single sequence of instructions.

MISD Multiple Instruction stream — Single Data stream

These machines have multiple data processing units but, here, each has its own instruction stream, and all operate on the same data, which is passed from one processor to another.

MIMD Multiple Instruction stream — Multiple Data stream

This is the most general class, and incorporates those machines which have multiple data

processing units under the control of more than one instruction streams, and also operate on more than one data stream.

Although widely adopted, Flynn's taxonomy gives only a basic view of machine structure, particularly in the case of the MIMD class, which includes many diverse structures. A form of classification which considers more details would, in many cases, be useful.

5.1.2 Shore's Taxonomy

Shore [160] defined six machine classes, numbered from Machine I to Machine VI:

I A sequential processor, which operates on one word of memory at a time, described by Shore as a 'horizontal processor'.

II A 'vertical processor', which operates on bit-slices of the memory in parallel, in a similar manner to an associative machine.

III Orthogonal machines; these are a combination of Machines I and II, with both horizontal and vertical processing units.

IV This is similar to Flynn's SIMD class, with multiple processors, each with a private data memory, and all controlled by a single instruction processing unit. No connections between the processors are provided.

V This is similar to Machine IV, but with inter-processor connections.

VI The 'logic in memory array', where multiple data processors are physically distributed across a memory array, with a single instruction processor in overall control.

Machines I and II are similar; the only difference is that Machine I operates in parallel on the short dimension of a two-dimensional memory array (word-parallel), whereas Machine II operates in parallel on the long dimension (bit-slice). Machines IV and V may be regarded as more powerful versions of Machine II, which may operate on slices of more than one bit. Machine VI is only distinguished from Machines II, IV and V by its physical layout. Shore's taxonomy has no class for machines operating on multiple instruction streams, and, therefore, has limited applications.

5.1.3 Hockney and Jesshope's Taxonomy

Hockney and Jesshope [161] describe a machine structure description language (similar in many respects to PMS [162]) in which machines are described in terms of the types of unit present and the nature of connections between them. The language allows new types of unit to be defined and incorporated as components of other units. This complex language provides a mechanism to describe hardware in detail, but no classification of machines is made.

5.1.4 Other Taxonomies

Many other taxonomies and classification schemes have been proposed, but many of these do not cover all machine designs, or are based on arbitrary machine features.

Haynes [163] describes a number of classes of machine — those with special functional units (such as systolic array units); associative processors; lattice processor arrays; data flow machines; functional language processors; and multiple-processor MIMD. This does not, however, cover all possible machine structures.

Basu [165] describes a taxonomy in which machines are classified into sixteen classes according to concurrency level; method of algorithm realisation; execution characteristics and control structure. Several of these classes are empty, however, and a number are indistinct.

Enslow [154, 167, 169, 170] defines a multiprocessor to be a machine with multiple processing units which share a common memory and peripherals, and are united by a single operating system. Interaction between processors in machines of this class must take place at a low level. This is contrasted with a multi-computer, in which essentially independent processors interact by passing data at a 'file' level. This is a very general form of machine classification.

Lorin [153] describes a number of machines in terms of instruction units, execution units and store units, but no distinct machine classes are described.

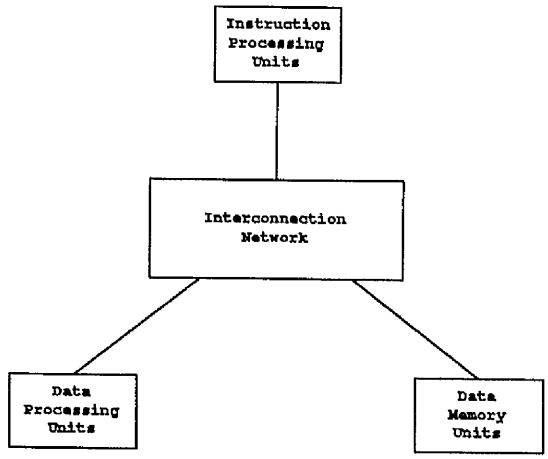


Fig. 5.1: A machine composed of IPUs, DPUs and DMUs.

5.2 A Taxonomy of Machines

None of the above succeed in defining a taxonomy to give sufficient detail of each machine in a structured manner, whilst avoiding unnecessary detail, to allow distinct machine classes to be easily recognised. A taxonomy intended to achieve this is now presented.

A general view of any processor is that it is composed of units of three basic types: instruction processing units, data processing units and memory units, similar to Lorin's I, E and S units [153]. These are linked together by means of an interconnection network (Fig. 5.1). Each type of unit is defined in terms of specification of its function, rather than a description of its implementation. This separation of function from implementation is seen as important, since the software of the machine is constrained only by the functional specification of the machine, whereas the implementation of this specification determines the actual machine efficiency and performance.

5.2.1 Instruction Processing Units

The function of an instruction processing unit (IPU) is to organise, fetch and decode a single stream of instructions, and to send the decoded instructions to one or more data processing units. In general, the location of the instruction memory is not of importance, and it is assumed that the instruction memory is internal to the IPU, unless specifically indicated to the contrary.

5.2.2 Data Processing Units

The function of a data processing unit (DPU) is to perform arithmetic and logical operations on data. The implementation of a DPU may involve the use of internal data memory, or microcode instruction processors, which in themselves might be considered to be separate units, but since these are not essential to the functional specification of a DPU, these implementation details are not considered here. Since the combination of an IPU connected directly to a DPU is a commonly encountered structure, this configuration will be referred to as a processing unit (PU).

5.2.3 Data Memory Units

The primary function of a data memory unit (DMU) is to store data. A small amount of processing, such as semaphore operations [172], or increment-in-memory instructions, may take place in the memory unit.

5.2.4 The Interconnection Network

The purpose of the interconnection network is to transfer information between the IPUs, DPUs and DMUs, as shown in Fig. 5.1. In general, only a few of the possible links which could be made between the IPUs, DPUs and DMUs are of any practical use, and so the implemented interconnection network does not connect each unit to every other. A method of describing the functional specification and implementation of interconnection networks, in such a way that useful characteristics of the network may be derived, is now presented. This scheme involves the reduction of interconnection networks to a partial crossbar form, and the use of a matrix representation derived from this, to obtain quantitative characteristics of the network. A number of simple single-stage networks have been arranged in a partial crossbar form by Hockney and Jesshope [161], and Maekawa et al. [174] used a matrix representation to describe a multiple time-shared bus network, but in neither case were any network characteristics derived from these.

An interconnection network is defined here to be some mechanism which accepts data items from a set of source units, s, and delivers these to some specified member of a set of destination

units, d. It should be noted that any architectural unit (IPU, DPU, or DMU) always appears to the interconnection network as, separately, a source unit and a destination unit, for the units output and input traffic, respectively. The time taken to perform this transfer, and the amount of interconnection network resource occupied at any time, are implementation-dependent.

5.2.5 A Matrix Representation for Interconnection Networks

The connections provided by any interconnection network, may be represented by an interconnection matrix, Z. This matrix contains one row for every source unit, and one column for every destination unit, connected by the network. Each element, Z_{ij}, denotes the number of simultaneous unidirectional communications paths, provided by the network, from source unit s_i to destination unit d_j. This defines the connections required for the machine to function in some specified manner, but not the way in which these are to be provided.

As an example, a sequential, von Neumann machine may be represented as one IPU, i, one DPU, d and one DMU, m, interconnected by a network. Each of these units has a network source connection (the unit output), and a network destination connection (the unit input), represented by s_i, d_i, s_d, d_d, s_m and d_m. The set of source units, s, is thus $\{s_i, s_d, s_m\}$, and the set of destination units, d, is $\{d_i, d_d, d_m\}$. The interconnection matrix, in which one row corresponds to each source set element, and one column corresponds to each destination set element, is

$$\begin{array}{c} \\ s_i \\ s_d \\ s_m \end{array} \begin{array}{c} \begin{array}{ccc} d_i & d_d & d_m \end{array} \\ \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \end{array}.$$

The '1' in the first row indicates that the source unit s_i, the IPU, sends some form of data to the destination unit d_d, the DPU. This is the instruction stream. The second row indicates that the source unit s_d, the DPU, sends data to d_i, the IPU, and to d_m, the DPU. These are the paths for test results in conditional instructions, and for data storage, respectively. The last row shows the data retrieval path from s_m to d_d. Since these four paths have different purposes, it is likely that their implementation will also be different; for example, the bandwidth, size, and cost of the DPU to IPU connection will, in practice, be much smaller than the DPU to DMU connection, because of its relative infrequence of use.

The sequential machine is a simple example, and for more complex machines, the matrices involved may be very large. To deal with these more easily, the network may be partitioned into sub-matrices but, before this is discussed, a number of matrix functions and descriptive terms are defined.

5.2.6 Definitions of Matrix Functions and Descriptive Terms

A number of definitions are now made, to explain the notation for, and facilitate discussion of, the proposed matrix representation of interconnection networks.

i) A matrix is said to be a **zero-matrix**, depicted here as (0), if all it's elements are zero. This is expanded here, such that an **r-matrix** is defined as a matrix in which all elements have the value r. This is written (r).

ii) A matrix is defined to be **fully connected** if, and only if, all elements are greater than zero.

iii) A matrix is defined to be **partially connected** if it is neither a zero-matrix nor a fully-connected matrix, and no element is less than zero.

iv) The **total** of a matrix is defined here to be the sum of all the elements of a matrix.

v) The **linkage** of a matrix is defined to be the number of non-zero elements in a matrix.

vi) The **minor matrix**, $M_{ij}(Y)$, of a matrix Y is defined to be a matrix identical to Y, but with its ith row and jth column set to zero.

vii) Any network which has p inputs and q outputs will be referred to as a $p \times q$ network, since it may be represented by a $p \times q$ matrix.

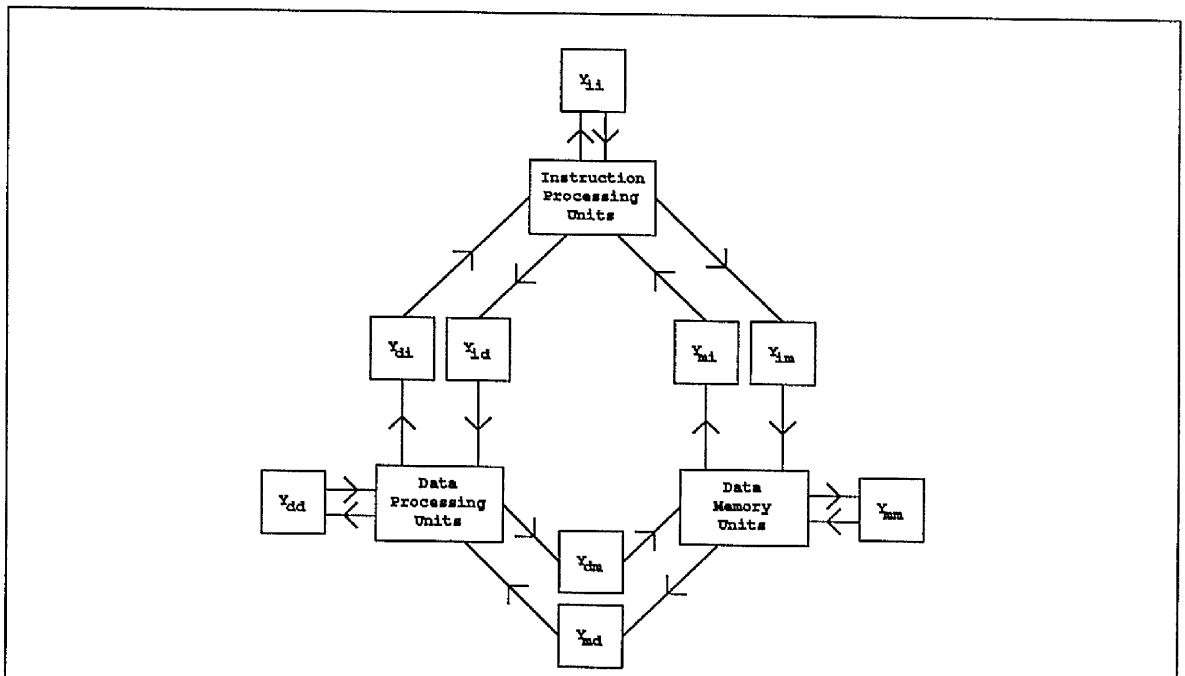


Fig. 5.2: A machine with partitioned interconnection.

5.2.7 Partitioning the Interconnection Network

The interconnection network shown in Fig. 5.1 may be partitioned in such a way as to isolate the connections between the sets of functional units, as shown in Fig. 5.2. The corresponding division of the interconnection matrix, Z, gives nine sub-matrices, representing the nine non-intersecting sub-networks:

$$Z = \begin{pmatrix} Y_{ii} & Y_{id} & Y_{im} \\ Y_{di} & Y_{dd} & Y_{dm} \\ Y_{mi} & Y_{md} & Y_{mm} \end{pmatrix}.$$

Each sub-matrix represents a part of the overall connection, as shown in Table 5.1. These sub-matrices are frequently simple functions, such as the identity matrix, I, the zero-matrix, (0), or the one-matrix, (1). It should be noted that, for most conventional uniprocessor and multiprocessor architectures,

$$Y_{ii} = Y_{im} = Y_{mi} = Y_{mm} = (0).$$

It is emphasised that these functions define which units are to be connected, and do not specify the physical nature of the connection.

Table 5.1 Sub-Matrix Connections		
Matrix	From	To
Y_{ii}	IPU	IPU
Y_{id}	IPU	DPU
Y_{im}	IPU	DMU
Y_{di}	DPU	IPU
Y_{dd}	DPU	DPU
Y_{dm}	DPU	DMU
Y_{mi}	DMU	IPU
Y_{md}	DMU	DPU
Y_{mm}	DMU	DMU

5.2.8 Circuit-Switched and Packet-Switched Interconnections

A distinction is often made between circuit-switched networks [175, 176], in which paths are requested from one unit to another, and are held open until relinquished, and packet-switched networks [177], where data 'packets' are presented to the network, which routes them to the appropriate destination unit, which may, at some later time, reply in a similar manner. In this descriptive notation, no explicit distinction is made between these types of network. Here, networks are assumed to be essentially circuit-switched, although any network may be used in a packet-switched mode by allowing intermediate units to forward data packets to other destination units. If necessary, otherwise unconnected 'forwarding nodes' could be created within the interconnection network, to perform this function. In general, networks which are frequently used are likely to be used in a circuit-switched mode, whereas those which are not are more likely to be used in a packet-switched mode.

5.2.9 Implementation of IPUs, DPUs and DMUs

There are many different methods of implementation of the basic architectural units. Any of these could be used with this taxonomy, provided that they satisfy the functional specification. Units which could be used as examples are well documented elsewhere [178, 179, 180]. These are not described here, but it is assumed that any IPUs, DPUs and DMUs which are mentioned are of a conventional design.

5.2.10 Implementation of Interconnection Networks

The functional specification of the interconnection network, as an interconnection matrix, has already been described. The implementation of such a network is now discussed.

It appears that any interconnection network may be implemented as a sequence of partial crossbar networks connected in series and in parallel. This has not been proved rigorously, but many networks have been examined, and no counter-examples have been found. All networks known as single-stage networks [176] may be implemented as a single partial crossbar. The implementation of each of the non-zero sub-matrices of Z may be considered as a sequence of partial crossbar networks:

$$Y = X_0.X_1.X_2 \ldots X_{n-1},$$

where each X is a matrix representing a partial crossbar, such that the element X_{ij} denotes the number of physical connections (0 or 1) from input i to output j of the partial crossbar, X. In such a representation, the matrix product element, Y_{ij}, represents the total number of possible pathways from source i to destination j. If two interconnection networks are placed in parallel, between the same set of sources and destinations, then the number of possible connections is given by the matrix sum of the two interconnection matrices.

In such a sequence of crossbars, any line may be driven by only one source unit at any time, and normally only one destination unit may be connected to a particular source unit at any time. As a result of this, when any connection i, j is selected across some network represented by X, the remaining network resources available to make further connections are given by the minor matrix $M_{ij}(X)$ (the matrix X with row i and column j set to zero). In some cases, however, a broadcast mode may be used to connect one source unit to several destination nodes simultaneously. In such cases, all appropriate destination columns must be zeroed, but not the source row.

5.2.11 Characterisation of Interconnection Networks

Four functions may be defined to characterise a network from its definition as a sequence of matrices. These are its cost, bandwidth and two measures of latency.

5.2.11.1 Network Cost

The object of this function is to give a measure of the amount of hardware required to implement any particular network. An approximation to the cost of a network, Y, may be defined in terms of the number of crosspoint equivalents in this implementation of Y,

$$\mathrm{C}(Y) = \sum_{\forall k} \mathrm{total}(X_k).$$

5.2.11.2 Network Bandwidth

The bandwidth function is a measure of the mean number of connections which a network may provide simultaneously. If all possible connections are assumed to be equally likely, then a measure of the bandwidth of a single-stage network Y, of size $p \times q$, may be defined as

$$\mathrm{B}(Y) = \mathrm{B}_0(Y, N_S, N_D),$$

where

$$N_S = \{s_0, s_1, s_2, \ldots s_{p-1}\}, \qquad \text{the set of source units,}$$

$$N_D = \{d_0, d_1, d_2, \ldots d_{q-1}\}, \qquad \text{the set of destination units,}$$

and

$$\mathrm{B}_n(Y, N_S, N_D) = \begin{cases} \dfrac{\mathrm{linkage}(Y)}{(p-n)(q-n)} + \underset{\substack{\forall s \in N_S \\ \forall d \in N_D}}{\mathrm{mean}} \left(\mathrm{B}_{n+1}\left(Y', N'_S, N'_D\right) \right), & \text{if } n < \min(p, q); \\ 0, & \text{otherwise.} \end{cases}$$

Here, $Y' = M_{ij}(Y)$,
$N'_S = N_S \setminus \{s\}$,
$N'_D = N_D \setminus \{d\}$, and
$\setminus$ is the set subtraction operator.

The first term in the expansion of $\mathrm{B}_0(Y', N'_s, N_d)$ is the number of different ways in which the first connection across this network may be made, divided by the number of possible pairs of unconnected source and destination units. The second term is a recursive expression for the mean number of subsequent connections which may be made, given that this first connection prevents any connection to one particular source unit and one particular destination unit. In this way, $\mathrm{B}(Y)$ is calculated as the mean number of simultaneous connections that may be made across this network.

To calculate the value of this function is a somewhat time-consuming process, even for relatively small networks, but for many regularly structured matrices it may be possible to derive analytic expressions for B(Y). Expressions for the bandwidth of some networks, and calculated values for others, are given in later parts of this chapter.

For multiple-stage networks, where Y is the product of a sequence of matrices $X_0.X_1.X_2 \ldots X_{n-1}$, it is important to note that the blocking effect of establishing a link from some source unit, s, to some destination unit, d, has a greater effect than the elimination of these source and destination units from future connections. In these networks, a path must be selected through the network, starting at the source unit i, through X_0 to some internal node of the network, then on through each X matrix in turn, to eventually reach the final destination node j. No connection through any of the X matrices used for this connection may be used by any subsequently established, but concurrent, connection. This means that a connection between some pair of units may prevent a completely separate connection between two other units, by occupying some interior node of the network which is required for this second connection. To deal with this, the definition of the minor matrix function, $M_{i,j}$ is extended, such that in the case where its parameter is a sequence of matrices, $X_0.X_1.X_2 \ldots X_{n-1}$, the function yields a sequence of matrices $X'_0.X'_1.X'_2 \ldots X'_{n-1}$ such that a path from unit i to unit j has been selected and eliminated from future consideration. Such paths take the form of sequences of intermediate node numbers, k_0 to k_n, such that $k_0 = i$, $k_n = j$. For a path to be established,

$$\left(X_a\right)_{k_a,k_a+1} \geq 1$$

must be true for all $0 \leq a < n$.

5.2.11.3 Network Latency

The latency of a network is a measure of the time taken to traverse the network. The maximum latency $\mathrm{L_{max}}(Y)$ may be defined as the maximum number of passes through the $p \times q$ network represented by Y, required to make any possible connection. Similarly, the mean latency, $\mathrm{L_{mean}}(Y)$, may be defined as the mean number of passes required, taken over all possible connections. These measures of latency

may be expressed as

$$\mathrm{L_{max}}(Y) = \max_{\substack{0 \le i < p \\ 0 \le j < q \\ i \neq j}} \left(l(i,j) \right)$$

and

$$\mathrm{L_{mean}}(Y) = \operatorname*{mean}_{\substack{0 \le i < p \\ 0 \le j < q \\ i \neq j}} \left(l(i,j) \right),$$

where $l(i,j)$ is the least integer such that

$$\left(Y^{l(i,j)} \right)_{ij} \geq 1.$$

Here, the function $l(i,j)$ yields the number of passes through a network which are required to send data from source unit i to destination unit j. It should be noted that the maximum and mean latencies are evaluated over all possible connections, except where $i = j$. This is because, for most (although not all) uses of complex networks, the set of source units is, in fact, the same as the set of destination units. In these cases, every source s_i is physically connected to the same unit as the destination d_i, in a unit mapping. For such networks, it is not appropriate to include the path from some unit to itself in the latency calculation, and so $l(i,i)$ is not considered. It should also be noted that for all fully-connected matrices, $\mathrm{L_{mean}}(Y) = \mathrm{L_{max}}(Y) = 1$.

5.2.11.4 Calculation of Cost, Bandwidth and Latency Values

The calculation of $\mathrm{C}(Y)$, $\mathrm{B}(Y)$, $\mathrm{L_{max}}(Y)$ and $\mathrm{L_{mean}}(Y)$ for the networks described in the remainder of this chapter required many hours of CPU time on the MU6-G research processor [182], in the Department of Computer Science at the University of Manchester. To reduce this time to a maximum of about three CPU-hours per value, a technique was used which tested only some pathways through the network, chosen at random. Using this 'Monte-Carlo' method, the number of pathways examined is reduced, and the required run-time may be shortened, at the expense of lowered accuracy. This loss is not significant, since these figures are presented to give only an approximate measure of the characteristics of the networks. In the tables of this chapter, the values marked with an asterisk (*) are calculated using this method. Each of these values is the mean of several runs, and is estimated to be accurate to within plus or minus one digit in the least significant place quoted.

5.3 A Description of Commonly Used Networks

A number of commonly used networks are now described, using the matrix notation, and some values of the functions $C(Y)$, $B(Y)$, $L_{mean}(Y)$ and $L_{max}(Y)$ are derived from this representation.

5.3.1 Fully-Connected Networks

Fully-connected interconnection networks permit any source unit to be connected to any destination unit, given exclusive use of the network. Any subsequent connections may be made or blocked, depending on the network and the connections already established. There are a number of common examples of fully-connected networks.

5.3.1.1 The Single Time-Shared Bus

A single time-shared bus which connects p source units to q destination units may be represented by two partial crossbars, as illustrated in Fig. 5.3. These may in turn be represented in a matrix form, thus:

$$X_0 = \begin{pmatrix} 1 \\ 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix}, \qquad X_1 = (1 \quad 1 \quad 1 \quad \ldots \quad 1),$$

$$Y = X_0.X_1 = \begin{pmatrix} 1 & 1 & 1 & \ldots & 1 \\ 1 & 1 & 1 & \ldots & 1 \\ 1 & 1 & 1 & \ldots & 1 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & 1 & \ldots & 1 \end{pmatrix}.$$

The single time-shared bus allows any source unit to connect to any destination unit, since $Y = (1)$. The cost of the network is represented here as the number of crosspoints required to connect each of the $p+q$ units to the common bus:

$$C(Y) = p + q.$$

The network bandwidth is low, since establishing a connection from any source unit, i, to any destination unit, j, precludes all other connections, and so for any i, j, the minor matrix $M_{ij}(Y) = (0)$. This gives

$$B(Y) = 1.$$

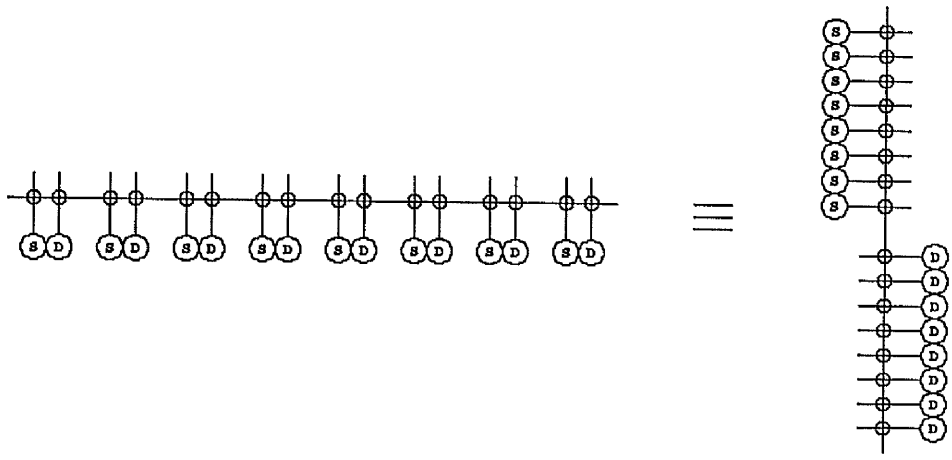


Fig. 5.3: A single time-shared bus network.

Since any unit may communicate with any other, the maximum and mean latencies are unity:

$$L_{max}(Y) = L_{mean}(Y) = 1.$$

From these functions, the single time-shared bus may be seen to be a low-cost, low-bandwidth network.

5.3.1.2 The Multiple Time-Shared Bus

To provide a connection with a higher bandwidth than the single time-shared bus, a number of buses may be used. A group of r time-shared buses, which connect p source units to q destination units, may again be represented by two partial crossbars, as shown in Fig. 5.4. These may be represented by the matrices:

$$X_0 = \begin{pmatrix} 1 & 1 & \dots & 1 \\ 1 & 1 & \dots & 1 \\ 1 & 1 & \dots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & \dots & 1 \end{pmatrix}, \qquad X_1 = \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & 1 & 1 & \dots & 1 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & 1 & \dots & 1 \end{pmatrix},$$

$$Y = X_0.X_1 = \begin{pmatrix} r & r & r & \dots & r \\ r & r & r & \dots & r \\ r & r & r & \dots & r \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ r & r & r & \dots & r \end{pmatrix}.$$

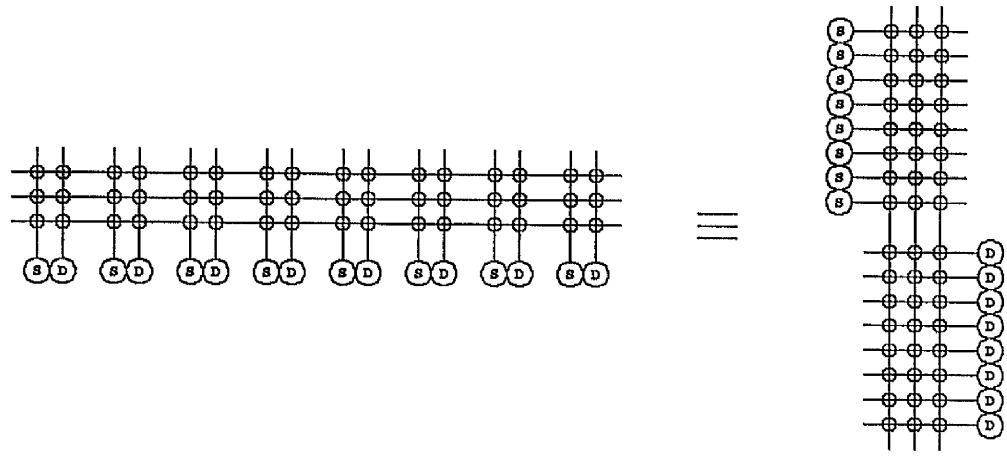


Fig. 5.4: A multiple time-shared bus network.

The multiple time-shared bus provides r different pathways from each source unit to every destination unit, since Y is an r-matrix. The cost of the network is r times that of the single time-shared bus. The bandwidth is increased proportionally to r and is limited only by the number of source, or destination, units. The latencies are, again, unity.

$$\mathrm{C}(Y) = r(p+q),$$

$$\mathrm{B}(Y) = \min(p, q, r),$$

$$\mathrm{L_{max}}(Y) = \mathrm{L_{mean}}(Y) = 1.$$

This network provides a higher bandwidth than the single time-shared bus, at a proportionately higher cost.

5.3.1.3 The Full Crossbar

The full crossbar is the limiting case of the multiple time-shared bus, where the number of buses, r, is the minimum of p and q. One of X_0 or X_1 becomes redundant, as shown in Fig. 5.5, and the network may be replaced by a single-stage network, X:

$$Y = X = \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & 1 & 1 & \dots & 1 \\ 1 & 1 & 1 & \dots & 1 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & 1 & \dots & 1 \end{pmatrix}.$$

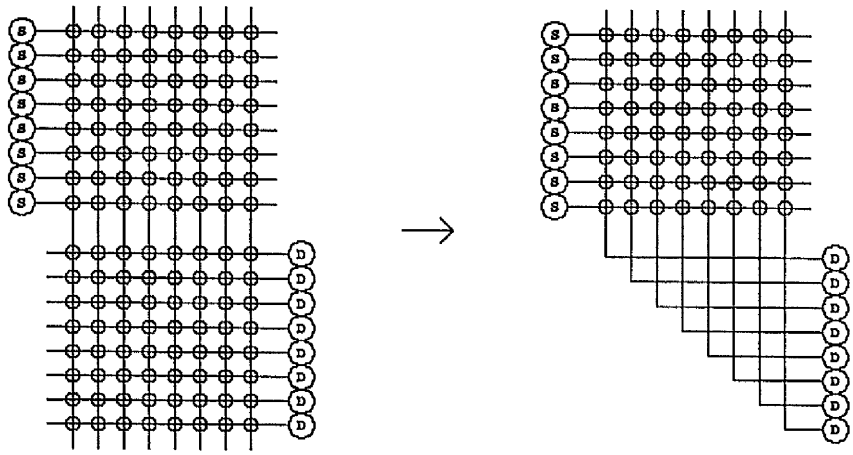


Fig. 5.5: A full crossbar network.

This reduction of two crossbars to a single crossbar does not affect the bandwidth or latencies of the network, but it should be noted that the multiple time-shared bus exhibits a certain amount of fault-tolerance, which the full crossbar does not.

The full crossbar provides simultaneous pathways from each source unit to every available destination unit. The cost of this network is equal to the number of crosspoints in the network, which is the product of p and q. The bandwidth is the highest possible, since either all source units (if $r = p$) or all destination units (if $r = q$) may be connected at any one time.

$$\mathrm{C}(Y) = pq,$$

$$\mathrm{B}(Y) = r = \min(p, q),$$

$$\mathrm{L}_{\max}(Y) = \mathrm{L}_{\mathrm{mean}}(Y) = 1.$$

Since the full crossbar requires only one matrix, it is always less costly to use a full crossbar than a multiple-bus system with greater than $\frac{pq}{p+q}$ buses. Special cases of the full crossbar are the 1×1 crossbar, which may be considered to represent a direct uninterruptible connection from one unit to another, and the 2×2 crossbar, which may be used to represent a switch or exchange unit [183] (Fig. 5.6).

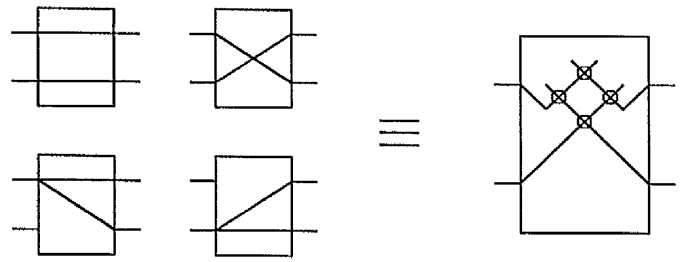


Fig. 5.6: The switch unit as a crossbar.

5.3.1.4 The Hierarchical Bus

The hierarchical bus structure comprises a number of 'clusters' of units, within which units are interlinked by a single time-shared bus. These clusters are themselves linked by a higher-level bus. Hierarchical structures of arbitrary depth may be constructed in this manner. The intra-cluster bus provides a pathway from each source unit to every destination unit within its cluster. For some unit to communicate with another outside its own cluster, the inter-cluster bus must be used.

A direct matrix representation of the hierarchical bus is possible, but involves the use of complex series/parallel arrangements of matrices. Methods of combining sequences of matrices in this way have not, as yet, been fully developed. However, the hierarchical bus may be manipulated, as shown in Fig. 5.7, into a sequence of crossbars which gives an approximate representation of this network. To do this, the intra-cluster buses are extended, without introducing any extra crosspoints, so that each bus crosses all of the clusters. By moving the crosspoints and their attached units along these buses, the source and destination units may be separated. To allow a crossbar form to be generated, the intra-cluster buses are broken, and re-linked by two unit crossbars. This does introduce a slight inaccuracy into the cost figure, since these two unit crossbars do not exist in the actual network. However, in practice, some form of bus management mechanism is required, and this may compensate for the inaccuracy.

A hierarchical bus network, with l levels of buses, may be represented, by this method, as $2l$ partial crossbars. As an example, nine units connected in three clusters of three units, as shown in

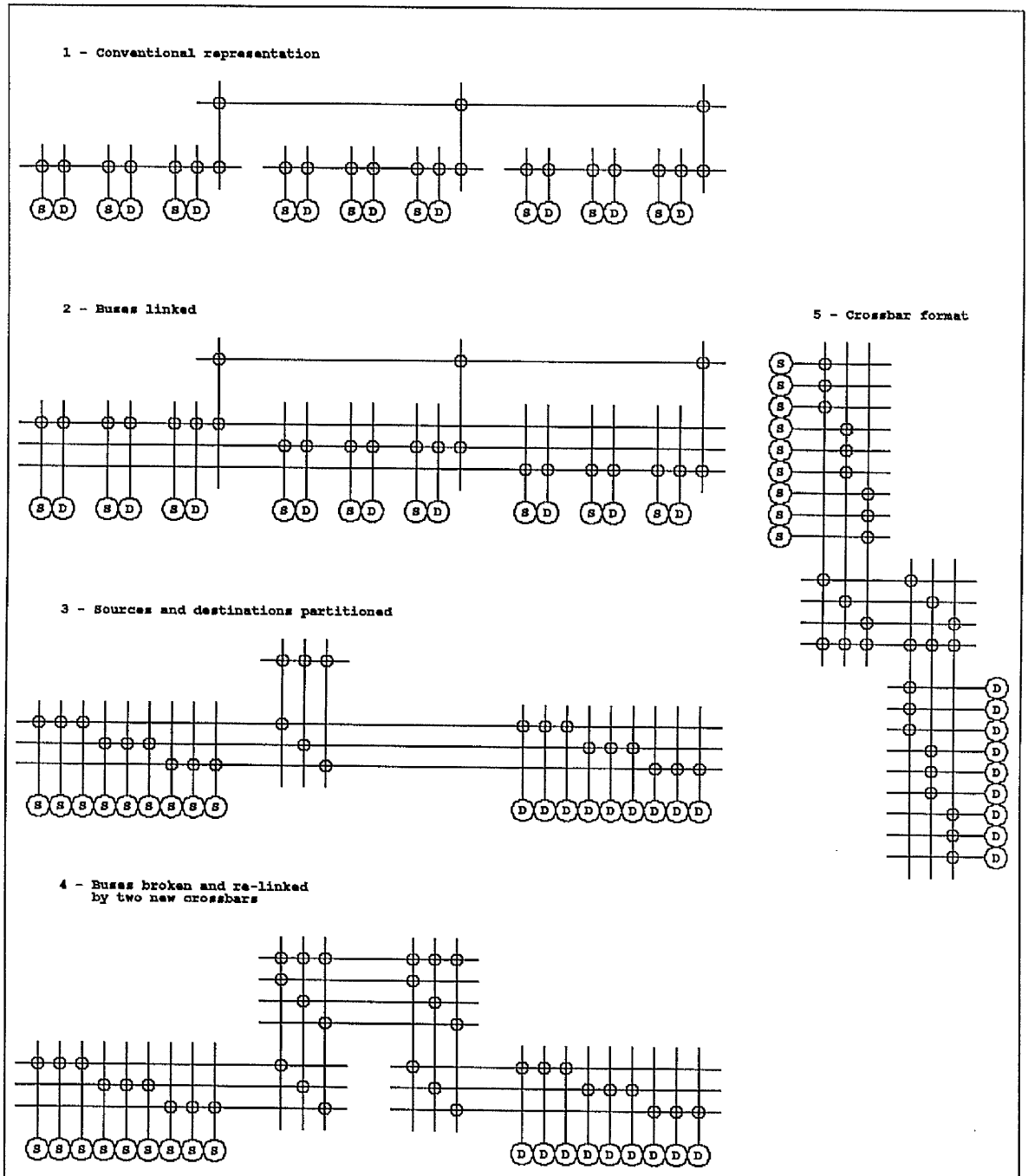


Fig. 5.7: A hierarchical bus network.

Fig. 5.7, may be represented by the four matrices:

$$X_0 = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix}, \qquad X_1 = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix},$$

$$X_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \end{pmatrix},$$

$$X_3 = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \end{pmatrix},$$

and

$$Y = X_0.X_1.X_2.X_3 = \begin{pmatrix} 2 & 2 & 2 & 1 & 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 1 & 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 2 & 2 & 2 & 1 & 1 & 1 \\ 1 & 1 & 1 & 2 & 2 & 2 & 1 & 1 & 1 \\ 1 & 1 & 1 & 2 & 2 & 2 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 2 & 2 & 2 \\ 1 & 1 & 1 & 1 & 1 & 1 & 2 & 2 & 2 \\ 1 & 1 & 1 & 1 & 1 & 1 & 2 & 2 & 2 \end{pmatrix}.$$

The hierarchical bus structure is not easily represented in an algebraic form, and so the derivation of analytic expressions for cost and bandwidth is difficult. The cost, bandwidth and latency for a number of hierarchical networks are shown in Table 5.2. From these figures, it may be seen that the bandwidth of the hierarchical bus increases with the number of clusters, but decreases as the cluster size increases. The bandwidth calculation is a measure of the mean number of available connections, taken over all possible permutations of source-destination connections. The hierarchical bus does not show good performance under these conditions, and is normally only used if most communication through the network is expected to be 'local traffic' and may, therefore, be handled by the intra-cluster bus.

5.3.1.5 The Indirect Binary n-Cube

The indirect binary n-cube [184], also known as the generalised cube [186, 188], is an n-stage network which connects 2^n source units to 2^n destination units. The name is derived from the fact that the connections at the nth stage correspond to the connections in the nth dimension of the direct binary n-cube network, which is a partially-connected network, described later in this chapter. The indirect binary n-cube is topologically identical to the STARAN flip network [190]. The conventional representation of the indirect binary n-cube uses 2×2 switch units, described earlier. This is shown in Fig. 5.8, for $n = 3$. By replacing

Table 5.2
Cost, Bandwidth and Latencies
for Hierarchical Bus Networks

Clusters	C(Y)	B(Y)	$L_{max}(Y)$	$L_{mean}(Y)$
2,2	16	1.222	1	1.00
3,3	20	1.180	1	1.00
4,4	24	1.633	1	1.00
2,2,2	24	1.403	1	1.00
3,2,2	26	1.358	1	1.00
3,3,2	28	1.343	1	1.00
3,3,3	30	1.343	1	1.00
2,2,2,2	32	1.524	1	1.00
3,2,2,2	34	1.481	1	1.00
3,3,2,2	36	1.460	1	1.00
3,3,3,2	38	1.45*	1	1.00
3,3,3,3	40	1.45*	1	1.00
4,4,4,4	48	1.42*	1	1.00

* (Values calculated using Monte-Carlo method)

each switch unit by a 2×2 crossbar, and rearranging slightly, the representation shown in Fig. 5.9 may be obtained. From this, it may be seen that any indirect binary n-cube may be represented by n $2^n \times 2^n$ matrices, and for $n = 3$,

$$X_0 = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix}, \qquad X_1 = \begin{pmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \end{pmatrix},$$

$$X_2 = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{pmatrix}, \qquad Y = X_0.X_1.X_2 = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}.$$

The cost of the indirect binary n-cube may be expressed as

$$C(Y) = 2n2^n.$$

The network latencies, $L_{max}(Y)$ and $L_{mean}(Y)$ are unity, as for all fully-connected networks. The bandwidth of this network has only been completely and exactly calculated for the smallest network of

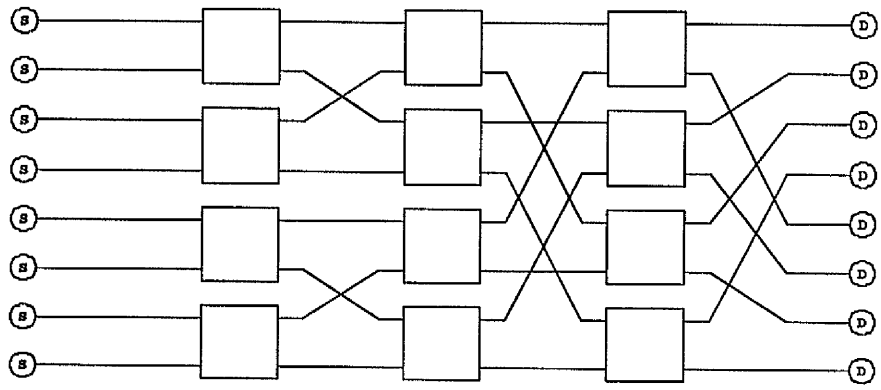


Fig. 5.8: The indirect binary 3-cube network, constructed from 2×2 switch units.

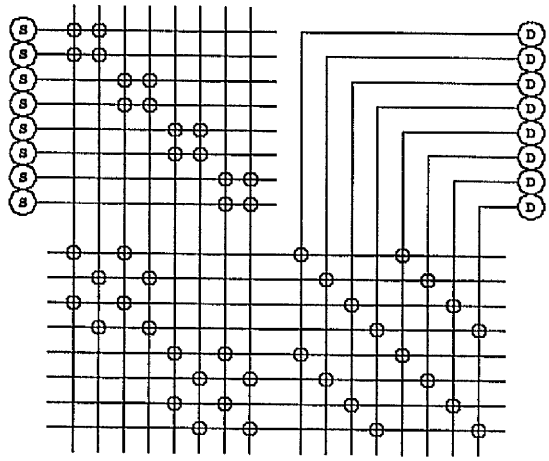


Fig. 5.9: The indirect binary 3-cube network in partial crossbar form.

Table 5.3
Cost, Bandwidth and Latencies for
Indirect Binary n-Cube Networks

n	2^n	C(Y)	B(Y)	$L_{max}(Y)$	$L_{mean}(Y)$
2	4	16	3.639	1	1.00
3	8	48	6.15*	1	1.00
4	16	128	10.0*	1	1.00

* (Values calculated using Monte-Carlo method)

interest, $n = 2$. Networks larger than this involve extremely large amounts of calculation to evaluate B(Y), and, with the available resources, only the following terms have been calculated exactly:

$$B(Y_2) = \frac{16}{16} + \frac{8}{9} + \frac{3}{4} + \frac{1}{1} = 3.639,$$

$$B(Y_3) = \frac{64}{64} + \frac{44}{49} + \frac{28\frac{4}{11}}{36} + \frac{17}{25} + \text{other terms},$$

$$B(Y_4) = \frac{256}{256} + \frac{208}{225} + \text{other terms},$$

where Y_n represents an indirect binary n-cube network. The indirect binary n-cube network is, however, highly symmetric, and it appears likely from the partial results obtained, that some relatively simple analytic expression exists for this bandwidth function. Table 5.3 shows C(Y) and B(Y) for these networks, calculated using the approximate 'Monte-Carlo' method described earlier in this chapter. From these figures, it may be seen that this network has a relatively low cost, compared with the full crossbar or time-shared buses, and has a reasonably high bandwidth.

5.3.1.6 The Omega Network

The omega network [191] is an n-stage network, usually constructed from switch units, in a similar manner to the indirect binary n-cube. The omega network comprises a series of perfect shuffle permutations, each of which is followed by 2^{n-1} switch units (Fig. 5.10). The perfect shuffle is a permutation which maps 2^n inputs to 2^n outputs. If the number i may be represented as an n-bit binary number $b_{n-1} \ldots b_2 b_1 b_0$, then the input, s_i, maps to the output whose binary representation is $b_{n-2} \ldots b_2 b_1 b_0 b_{n-1}$. This permutation derives its name from the observation that this is the same transformation which occurs if a deck of numbered cards are shuffled with a 'perfect shuffle', in which the deck is divided into two halves, which are then interleaved.

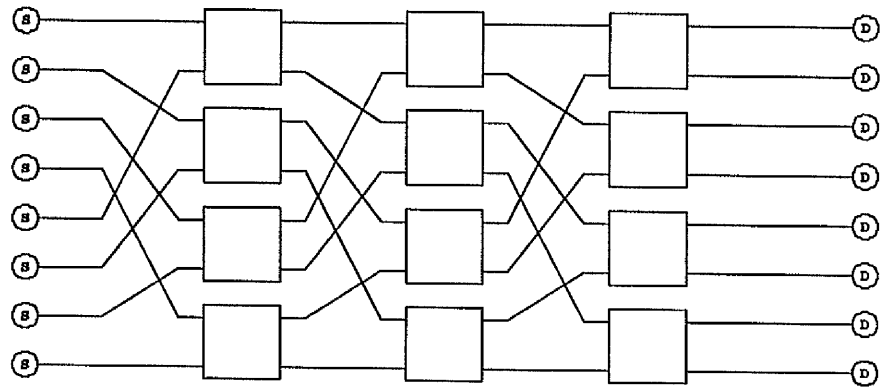


Fig. 5.10: An omega network, constructed from 2×2 switch units.

The switch units in the omega network may be replaced by 2×2 crossbar networks, and re-arranged in the same way as the previous network, the indirect binary n-cube. This results in a matrix representation which is similar to the indirect binary n-cube, differing only in that the order of the matrices X is reversed. If a network Y is implemented as an indirect binary n-cube, and

$$Y = X_0.X_1.X_2 \ldots X_{n-1},$$

then the corresponding omega network is given by

$$Y' = X_{n-1} \ldots X_2.X_1.X_0.$$

This series of matrices may be manipulated by interchanging pairs of rows and pairs of columns, and, in this way, may be shown to be identical to the indirect binary n-cube, but with re-numbered inputs and outputs. Thus, the omega network provides an identical interconnection pattern to the indirect binary n-cube, and differs only in the particular paths which are blocked by individual connections. This was shown, using a different method, by Siegel and Smith [188]. The cost, bandwidth and latency of the omega network are the same as those for the indirect binary n-cube.

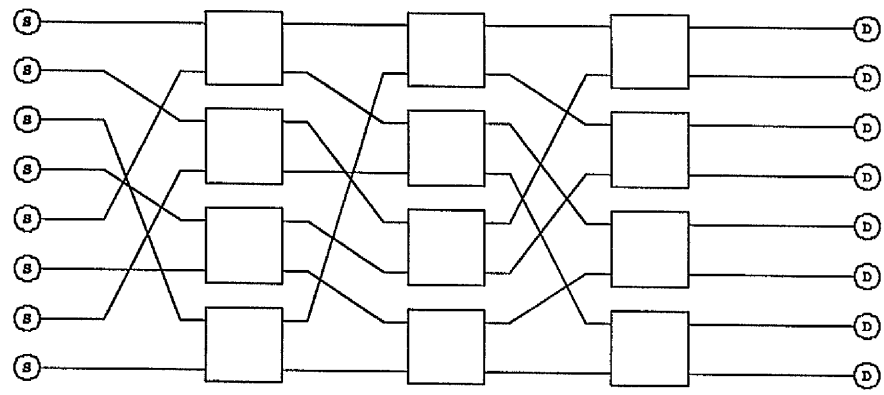


Fig. 5.11: A delta network, constructed from 2×2 switch units.

5.3.1.7 Delta Networks

Delta networks [192] are a class of networks, composed of 2×2 switch units, which connect 2^n inputs to 2^n outputs. The indirect binary n-cube and the omega network are symmetric cases of delta networks. Delta networks are not, in general, symmetric and are composed of n stages of switch units, connected by arbitrary permutations. Not all possible permutations result in usable networks. One example of a delta network is shown in Fig. 5.11, and this may be represented by the matrices

$$X_0 = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}, \qquad X_1 = \begin{pmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \end{pmatrix},$$

$$X_2 = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix}, \qquad Y = X_0.X_1.X_2 = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}.$$

The cost, bandwidth and latencies for fully-connected delta networks are believed to be the same as for the indirect binary n-cube and the omega networks.

5.3.1.8 The Augmented Data Manipulator

The augmented data manipulator (ADM) [188, 193] is an n-stage, indirect version of the partially-connected PM2I network (described later in this chapter). The ADM connects 2^n inputs to 2^n outputs, and is shown in Fig. 5.12, for $n = 3$. At the kth stage of the network, each node, i, connects to nodes i and $i \pm 2^k$ in the next stage. This may be placed in a partial crossbar form, as shown in Fig. 5.13. In the general case, this network may be represented by n matrices, and for the example shown,

$$X_0 = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix}, \qquad X_1 = \begin{pmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \end{pmatrix},$$

$$X_2 = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{pmatrix}, \qquad Y = X_0.X_1.X_2 = \begin{pmatrix} 1 & 3 & 2 & 3 & 1 & 3 & 2 & 3 \\ 3 & 1 & 3 & 2 & 3 & 1 & 3 & 2 \\ 2 & 3 & 1 & 3 & 2 & 3 & 1 & 3 \\ 3 & 2 & 3 & 1 & 3 & 2 & 3 & 1 \\ 1 & 3 & 2 & 3 & 1 & 3 & 2 & 3 \\ 3 & 1 & 3 & 2 & 3 & 1 & 3 & 2 \\ 2 & 3 & 1 & 3 & 2 & 3 & 1 & 3 \\ 3 & 2 & 3 & 1 & 3 & 2 & 3 & 1 \end{pmatrix}.$$

From inspection of the matrices, the connections provided by this network may be seen to be a superset of those provided by the indirect binary n-cube, since every element of the ADM matrices is greater than or equal to the corresponding element of the indirect binary n-cube matrices. The cost of this network may be shown to be

$$C(Y) = (3n - 1)2^n,$$

but the bandwidth is, in a similar manner to the indirect binary n-cube, computationally difficult to obtain, and only those values shown in Table 5.4 have been calculated. The cost of the ADM may be seen to be greater than that of the indirect binary n-cube, but the bandwidth is only marginally higher. Latencies are, again, unity.

The inverse augmented data manipulator (IADM) [195] is a similar network, using the same matrices, but in reverse order. The connections provided by the IADM are a superset of those provided by the omega network.

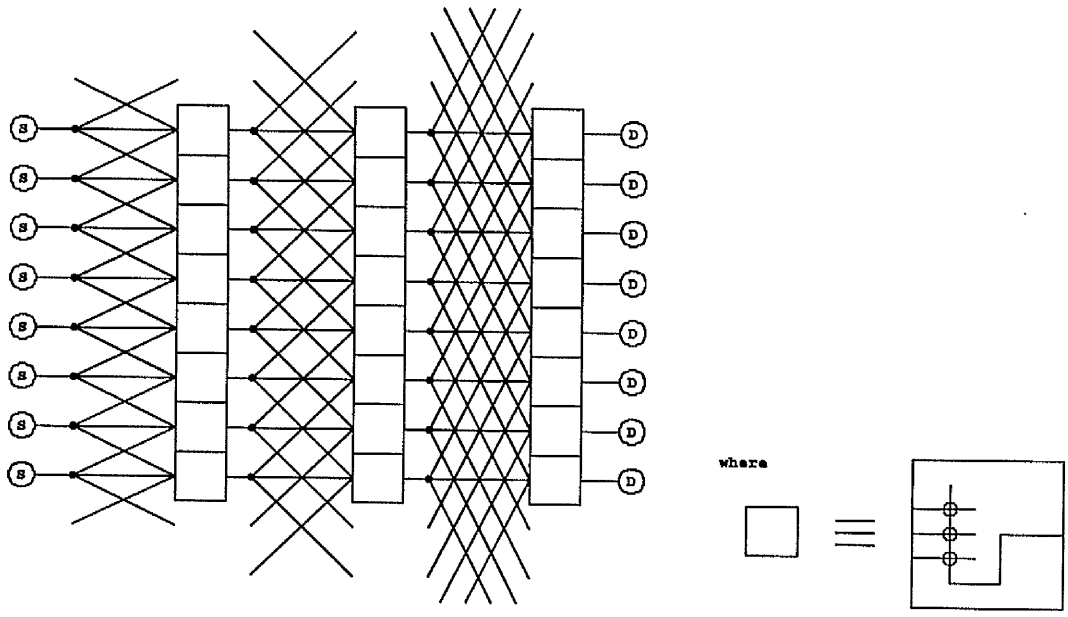


Fig. 5.12: An augmented data manipulator network.

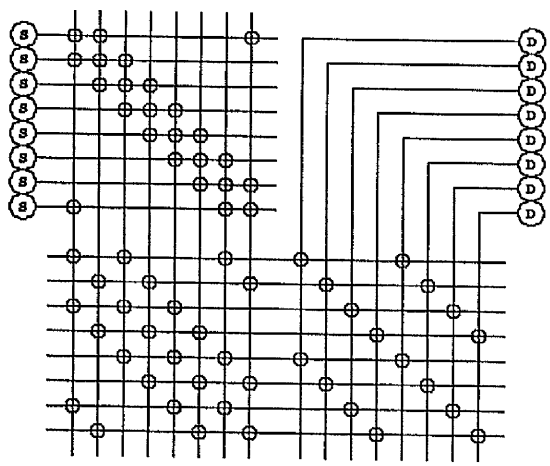


Fig. 5.13: The augmented data manipulator network in partial crossbar form.

Table 5.4
Cost, Bandwidth and Latencies for
Augmented Data Manipulator Networks

n	2^n	C(Y)	B(Y)	$L_{max}(Y)$	$L_{mean}(Y)$
2	4	20	3.827	1	1.00
3	8	64	6.16*	1	1.00
4	16	176	10.0*	1	1.00

* (Values calculated using Monte-Carlo method)

5.3.1.9 The Gamma Network

The gamma network [196] is a network which connects 2^n inputs to 2^n outputs. Each internal node may be connected to one of three nodes in the next stage, by means of a 3×3 crossbar (Fig. 5.14). The network may be reduced to a partial crossbar form, as shown in Fig. 5.15, and thus it may be represented by $n+1$ matrices. For $n = 2$:

$$X_0 = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \end{pmatrix}, \quad X_1 = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix},$$

$$X_2 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}, \qquad Y = X_0.X_1.X_2 = \begin{pmatrix} 2 & 3 & 2 & 2 \\ 2 & 2 & 3 & 2 \\ 2 & 2 & 2 & 3 \\ 3 & 2 & 2 & 2 \end{pmatrix}.$$

The gamma network shown in Fig. 5.14 has a high level of redundancy. Although this may be useful in respect of fault-tolerance, it does not increase the bandwidth of the network, which is limited by the number of sources and destinations. This also applies to larger gamma networks, but as network size

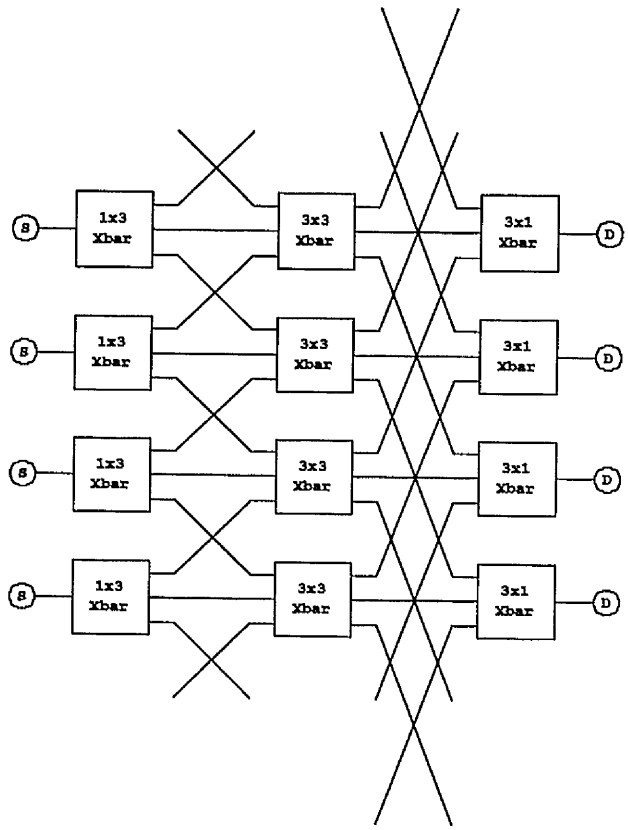


Fig. 5.14: A gamma network.

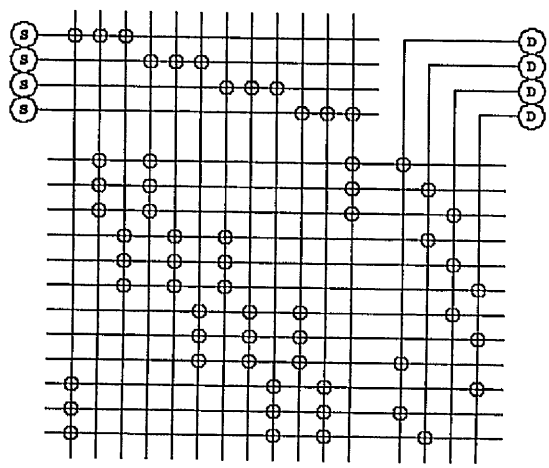


Fig. 5.15: The gamma network in partial crossbar form.

increases, the redundancy level is reduced. The cost of the gamma network may be shown to be

$$C(Y) = 3 \times 2^n(3n - 1).$$

The smallest gamma network for which this cost is less than the cost of a full crossbar is the 64×64 network, for which n=6, and so this is probably the smallest network which would be used in practice. This network is, however, too large for its characteristics to be investigated. Other networks may be devised, which are similar to the gamma network, but use crossbars larger than 3×3. Such networks must be very large to be economically viable.

5.3.1.10 The SW Banyan

The SW network [197], also known as the banyan [199, 200], or SW banyan [201, 202], is described as a network based on the tree structure. However, since no processing elements are present at the nodes of this 'tree', it may be regarded as a form of hierarchical bus network. The network is constructed from a tree-like hierarchical structure in which which all nodes of the tree above (closer to the root than) a certain level, are duplicated and connected to that level (Fig. 5.16). This duplication may be repeated s times, at each of the l levels of the tree, to give an (s, l) banyan. A network based on an f-ary tree, or hierarchical bus with cluster size f, is said to have a fanout of f. A rectangular banyan is one in which the fanout, f, is equal to the spread, s. These networks may be rearranged into a partial crossbar form, as shown in Fig. 5.17. The cost of a (s, l) banyan, with a fanout of f, may be shown to have a cost given by

$$C(Y) = s^l(1 + f) + \sum_{i=1}^{l-1} \left(s^i f^{l-i} \right) + (s + 1)f^l$$

.The latencies of these networks is unity, but the bandwidth is, again, difficult to obtain analytically. This has not been evaluated for the general case, but the (2, 2^n) banyan may be shown to be equivalent to either the indirect binary n-cube, or the omega network, depending on the direction of communication through the network. Examples of these networks have already been presented.

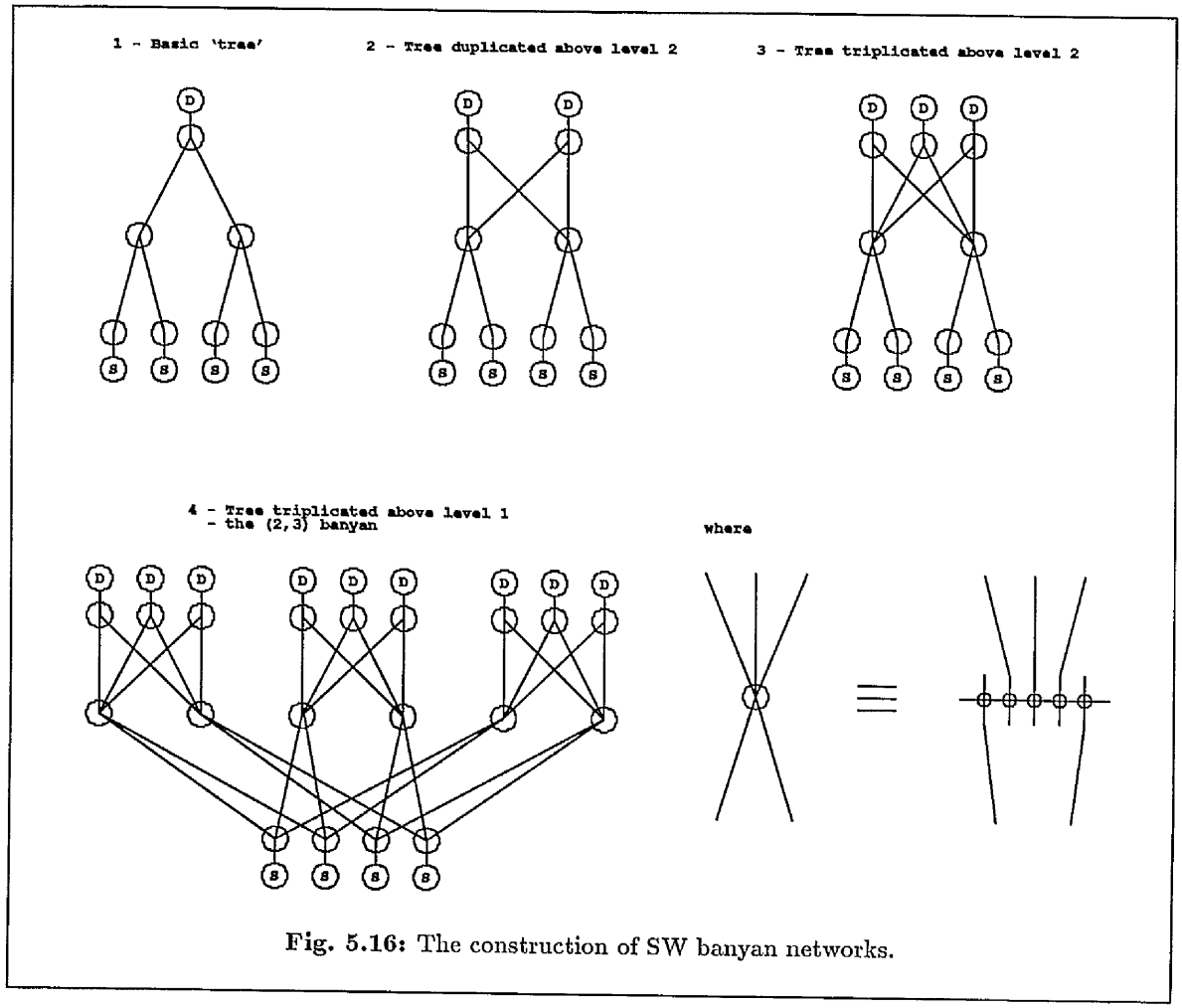


Fig. 5.16: The construction of SW banyan networks.

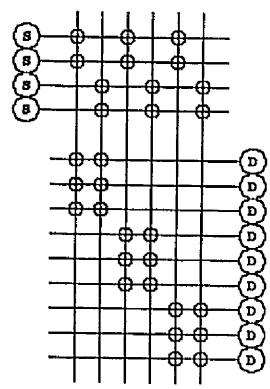


Fig. 5.17: An SW banyan network in partial crossbar form.

5.3.2 Partially-Connected Networks

All of the networks examined so far have been fully-connected networks. This means that any unit may send data directly to any other. A number of partially-connected networks are now examined, in which it is not possible to make every connection directly. For some connections in these networks, it is necessary for data to be sent to some unit other than its destination unit, which will forward the data to its ultimate destination.

Partially-connected networks are only easily usable in situations where the set of source units is, in fact, the same as the set of destination units, as described earlier. This is required so that data sent to some destination, to be forwarded, is always be able to re-emerge into the network from some source associated with the forwarding unit. If every forwarding unit did not have a network source, this would not be possible. Normally, the mapping of source to destinations is a unit mapping.

This mapping of sources to destinations means that there exist connections between the sources and destinations, other than those specified directly by the interconnection network. To represent these connections, an identity matrix must be added to the connection matrix, to obtain correct latency and bandwidth values. The cost function, $C(Y)$, is artificially increased by this addition, but this could be considered to represent the additional hardware required, in these networks, to perform the forwarding operations.

5.3.2.1 The Unidirectional Ring

The unidirectional ring is a network which connects p source units to p destination units, where each source unit, s_i, may connect to one of two destinations — its corresponding destination, d_i, and the next one in sequence, d_{i+1} [204]. In this way a closed ring is formed, with the last unit connecting back to the first. The ring may be depicted in partial crossbar form, as shown in Fig. 5.18. From this,

$$Y = X = \begin{pmatrix} 1 & 1 & 0 & \dots & 0 & 0 \\ 0 & 1 & 1 & \dots & 0 & 0 \\ 0 & 0 & 1 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 1 & 1 \\ 1 & 0 & 0 & \dots & 0 & 1 \end{pmatrix}.$$

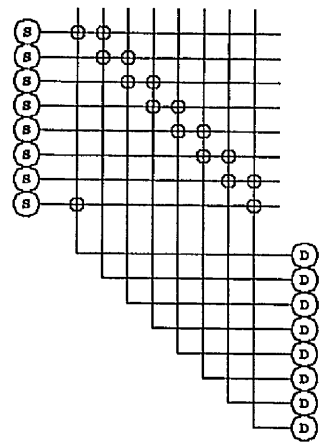


Fig. 5.18: A ring network in partial crossbar form.

Table 5.5
Cost, Bandwidth and Latencies
for Unidirectional Ring Networks

p	$C(Y)$	$B(Y)$	$L_{max}(Y)$	$L_{mean}(Y)$
2	4	2.000	1	1.00
3	6	2.083	2	1.50
4	8	2.156	3	2.00
5	10	2.214	4	2.50
6	12	2.261	5	3.00
7	14	2.300	6	3.50
8	16	2.332	7	4.00
16	32	2.5*	15	8.00

* (Value calculated using Monte-Carlo method)

Table 5.6
Cost, Bandwidth and Latencies
for Bidirectional Ring Networks

p	$C(Y)$	$B(Y)$	$L_{max}(Y)$	$L_{mean}(Y)$
3	9	3.000	1	1.00
4	12	3.132	2	1.33
5	15	3.209	2	1.50
6	18	3.310	3	1.80
7	21	3.40*	3	2.00
8	24	3.48*	4	2.29
16	48	3.96*	8	4.27

* (Values calculated using Monte-Carlo method)

The cost and latencies of the ring network may be expressed as

$$C(Y) = 2p,$$

$$L_{max}(Y) = p - 1,$$

$$L_{mean}(Y) = \frac{p}{2}.$$

(Shown in appendix 1.)

These are shown, with calculated bandwidth values, for some networks in Table 5.5. It may be seen that the network cost is low, and, consequently, the bandwidth is also quite low. Latency is high because of the large degree of forwarding that is required in this network.

5.3.2.2 The Bidirectional Ring

The unidirectional ring network may be augmented by the provision of a second ring, to allow communication in the reverse direction. The matrix for this bidirectional ring network is

$$Y = X = \begin{pmatrix} 1 & 1 & 0 & 0 & \dots & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & \dots & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & \dots & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & \dots & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & \dots & 0 & 1 & 1 \end{pmatrix}.$$

This modification decreases the latency of the ring, but also increases the cost:

$$C(Y) = 3p,$$

$$L_{max}(Y) = \begin{cases} \frac{p}{2} - \frac{1}{2}, & \text{if } n \text{ is odd,} \\ \frac{p}{2}, & \text{if } n \text{ is even,} \end{cases}$$

$$L_{mean}(Y) = \begin{cases} \frac{1}{4}(p+1), & \text{if } n \text{ is odd,} \\ \frac{p^2}{4(p-1)}, & \text{if } n \text{ is even.} \end{cases}$$

(Shown in appendix 1.)

The bandwidth of the network is increased, as shown in Table 5.6.

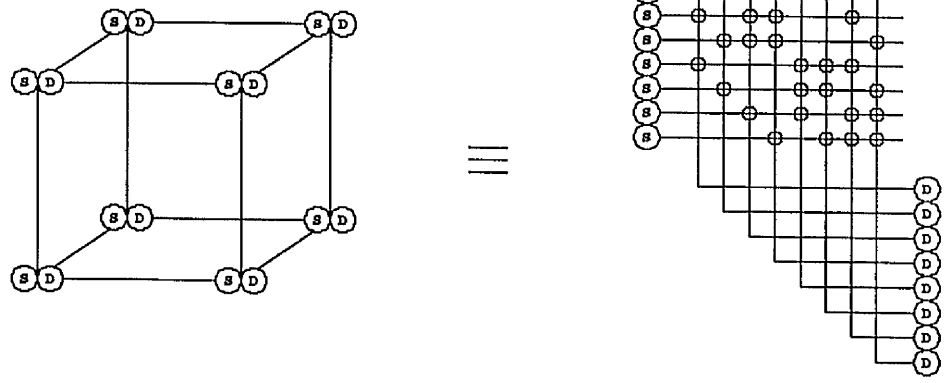


Fig. 5.19: A binary n-cube network.

5.3.2.3 The Binary n-Cube Network

The binary n-cube (or boolean n-cube) network is a single-stage network which connects 2^n inputs to 2^n outputs [205]. The name is derived from the resemblance to an n-dimensional cube, in which adjacent vertices are linked. This is shown, for $n = 3$, in Fig. 5.19. If the n-bit number i is represented as a binary number $b_{n-1} \ldots b_2 b_1 b_0$, then each input, s_i, may be connected to its corresponding output, d_i, or to one of the outputs whose binary representation is $b_{n-1} \ldots \overline{b_k} \ldots b_2 b_1 b_0$, for any k $(0 \leq k < n)$. Each source unit is thus connected to $n + 1$ destination units, and for $n = 3$,

$$Y = X = \begin{pmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{pmatrix}.$$

The cost and latencies may be calculated thus:

$$\mathrm{C}(Y) = 2^n(n+1),$$

$$\mathrm{L}_{\max}(Y) = n,$$

$$\mathrm{L}_{\mathrm{mean}}(Y) = \frac{n2^{n-1}}{2^n - 1}.$$

(Shown in appendix 1.)

Table 5.7
Cost, Bandwidth and Latencies
for Binary n-Cube Networks

n	2^n	C(Y)	B(Y)	$L_{max}(Y)$	$L_{mean}(Y)$
2	4	12	3.382	2	1.33
3	8	32	4.27*	3	1.71
4	16	80	5.3*	4	2.13

* (Values calculated using Monte-Carlo method)

Some values of these functions, together with the bandwidth, are shown in Table 5.7. It may be seen that, in comparison with the unidirectional and bidirectional ring networks, the binary n-cube has a slightly higher bandwidth and a lower latency, at a slightly higher cost.

5.3.2.4 The Tree Structure

The tree structure (Fig. 5.20) is a single-stage network in which each interior node is connected to one parent node and a number of child nodes. Exterior, or leaf, nodes connect only to the parent node. A single root node connects only to child nodes. A tree in which each node has two child nodes is referred to as a binary tree, and this is the most commonly encountered form. A full binary tree with 15 units, forming four levels of one, two, four and eight units, may be rearranged to the partial crossbar form shown in Fig. 5.21, and thus represented by the matrix

$$Y = X = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \end{pmatrix}.$$

The cost and maximum latency for a $p \times p$ binary tree network may be shown to be

$$C(Y) = 3p - 2,$$

$$L_{max}(Y) = 2\sqrt{p+1} - 2.$$

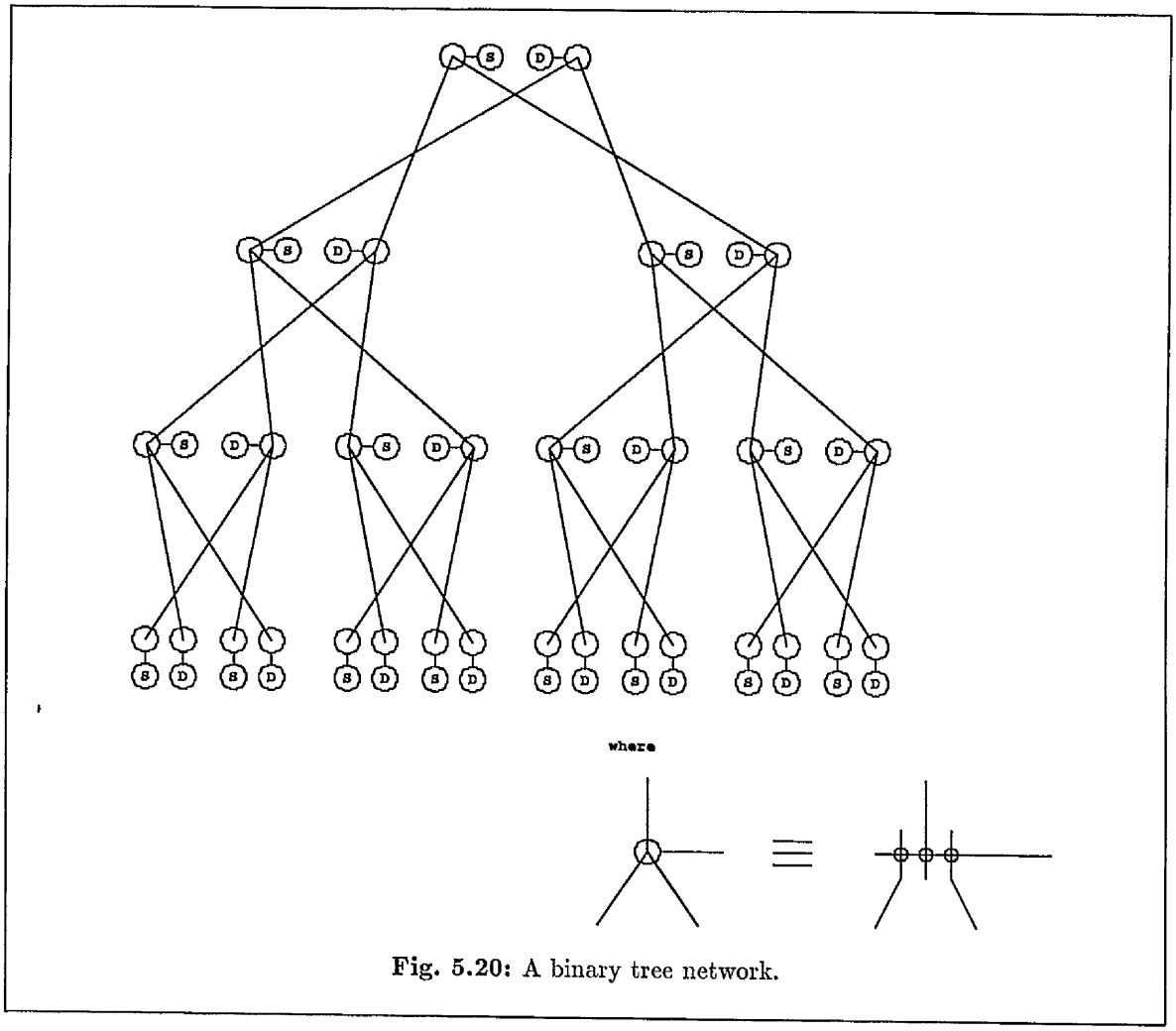


Fig. 5.20: A binary tree network.

Table 5.8
Cost, Bandwidth and Latencies
for Binary Tree Networks

p	C(Y)	B(Y)	$L_{max}(Y)$	$L_{mean}(Y)$
3	7	2.382	2	1.33
7	19	3.099	4	2.29
15	43	3.72*	6	3.50

* (Value calculated using Monte-Carlo method)

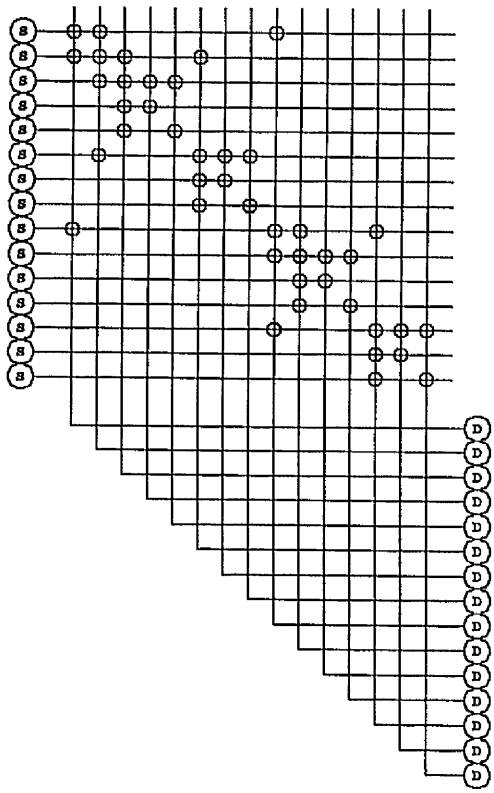


Fig. 5.21: The binary tree network in partial crossbar form.

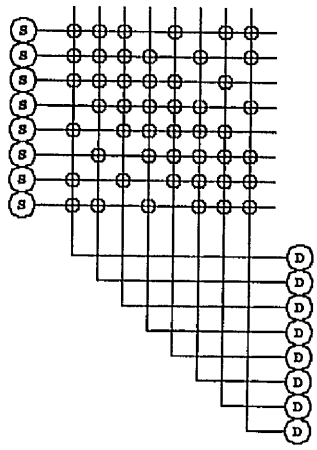


Fig. 5.22: A PM2I network in partial crossbar form.

These are calculated, together with $B(Y)$ and $L_{mean}(Y)$ in Table 5.8. The high latency values are due to the highly localised connections in this network.

Tree networks may be augmented by cross-connections between lower nodes [206, 207] in order to reduce the maximum and mean latencies. The tree network may be regarded as the dual of the hierarchical bus, since each has nodes where the other has links, and vice versa. This network is a popular choice for VLSI structures, since it may be efficiently implemented in a two-dimensional layout [209].

5.3.2.5 The PM2I Network

The PM2I network [210, 211, 212] is a single-stage network which connects 2^n inputs to 2^n outputs (Fig. 5.22). Input s_i is connected to output d_i and all outputs $d_{i\pm 2^k}$ for all $0 \leq k < n$. For $n = 3$, the connection matrix is

$$Y = X = \begin{pmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 \end{pmatrix}.$$

The cost of the PM2I network may be shown to be

$$C(Y) = n2^{n+1}.$$

The value of $L_{max}(Y)$ is not easily derived for the general case, but is clearly no greater than n, since the connections given by this network are a superset of those given by the binary n-cube, for which $L_{max} = n$. Values of $C(Y)$, $B(Y)$, $L_{max}(Y)$ and $L_{mean}(Y)$ for the PM2I network are shown in Table 5.9. It may be seen that the bandwidth is greater than that of the binary n-cube, but this increase is slightly less than the proportional increase in cost.

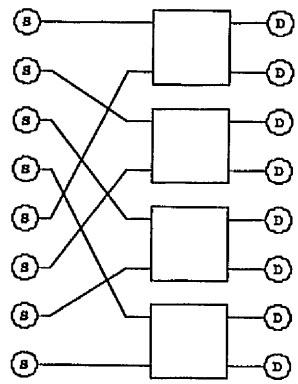


Fig. 5.23: A shuffle-exchange network.

Table 5.9
Cost, Bandwidth and Latencies
for PM2I Networks

n	2^n	C(Y)	B(Y)	$L_{max}(Y)$	$L_{mean}(Y)$
2	4	16	4.000	1	1.00
3	8	48	6.13*	2	1.29
4	16	128	8.24*	2	1.53

* (Values calculated using Monte-Carlo method)

Table 5.10
Cost, Bandwidth and Latencies
for Shuffle-Exchange Networks

n	$n2^n$	C(Y)	B(Y)	$L_{max}(Y)$	$L_{mean}(Y)$
2	4	10	2.711	2	1.50
3	8	22	3.22*	3	2.11
4	16	46	3.6*	4	2.83

* (Values calculated using Monte-Carlo method)

5.3.2.6 The Shuffle-Exchange Network

The shuffle-exchange network, as described by Stone and subsequent workers [213, 214, 215], is a single-stage network comprising a perfect shuffle permutation (described earlier in connection with the omega network), followed by a switch unit connecting adjacent lines (Fig. 5.23). This may be represented in matrix form as

$$Y = X = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix}.$$

This is slightly different to the shuffle-exchange network described by Siegel [210], where the network described is a perfect shuffle connection in parallel with an exchange connection, which gives the matrix

$$Y = X = \begin{pmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{pmatrix}.$$

The cost of the shuffle-exchange network (as defined by Stone, and depicted in Fig. 5.23) may be seen to be

$$\mathrm{C}(Y) = 3 \times 2^n - 2.$$

Maximum latency may be shown to be

$$\mathrm{L}_{\max}(Y) = n.$$

These, and some values of $\mathrm{B}(Y)$ and $\mathrm{L}_{\mathrm{mean}}(Y)$ are shown in Table 5.10. The shuffle-exchange network may be seen to be a relatively low-cost network, which exhibits a marginally lower bandwidth than the binary tree network, but has a lower latency.

5.3.2.7 The Rectangular Lattice Network

The rectangular lattice network, sometimes known as the Illiac network, is a single-stage network in which the sources and destinations are arranged in a rectangular lattice structure and each source unit may be connected to the adjacent destination units. The lattice may be constructed in any number of dimensions, but the most commonly encountered form is the two-dimensional lattice, shown in Fig. 5.24. A variant of this network allows connections to be made to eight neighbouring units, by including diagonal connections. The network shown in Fig. 5.24 may be re-arranged to give the partial crossbar shown in Fig. 5.25. This gives the matrix representation

$$Y = X = \begin{pmatrix} 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \end{pmatrix}.$$

The cost and maximum latency for a four-connected square network of p inputs and p outputs may be shown to be

$$\mathrm{C}(Y) = 4p,$$

$$\mathrm{L_{max}}(Y) = \begin{cases} \frac{\sqrt{p}-1}{2} - \frac{1}{2}, & \text{if } n \text{ is odd,} \\ \frac{\sqrt{p}-1}{2}, & \text{if } n \text{ is even.} \end{cases}$$

These are shown, with $\mathrm{B}(Y)$ and $\mathrm{L_{mean}}(Y)$, in Table 5.11. The cost and bandwidth may be seen to compare favourably with other networks, but maximum latency is high for large networks.

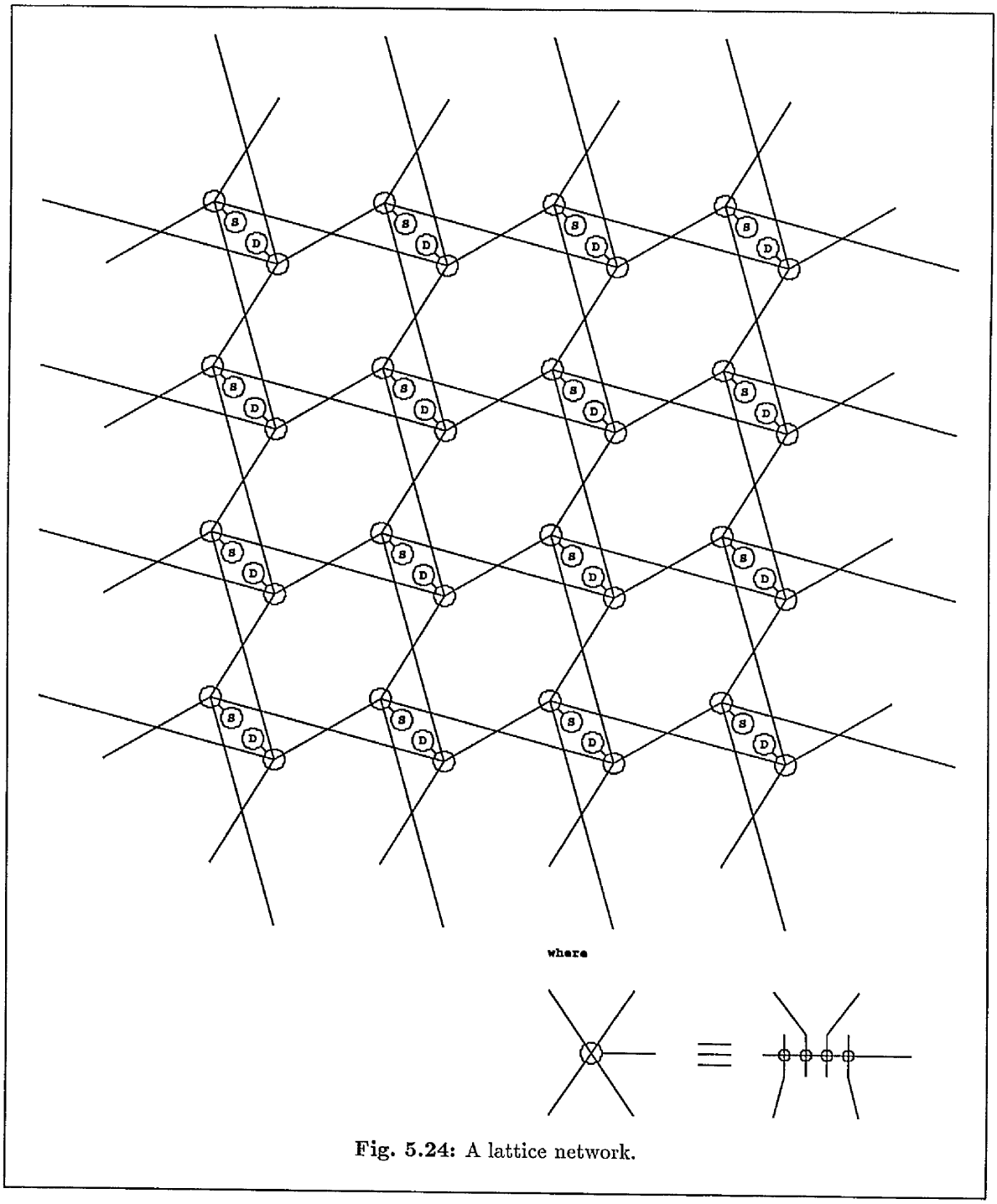


Fig. 5.24: A lattice network.

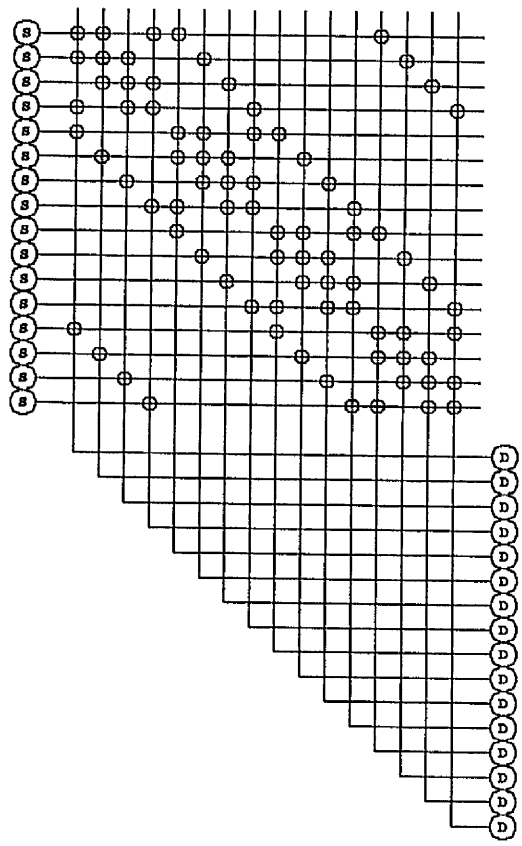


Fig. 5.25: The lattice network in partial crossbar form.

Table 5.11
Cost, Bandwidth and Latencies
for Lattice Networks

Size	p	C(Y)	B(Y)	$L_{max}(Y)$	$L_{mean}(Y)$
2 × 2	4	12	3.132	2	1.00
3 × 3	9	45	5.22*	2	1.33
4 × 4	16	80	5.42*	4	2.66

* (Values calculated using Monte-Carlo method)

5.3.2.8 The Partitioned Indirect Binary n-Cube Network

The partitioned indirect binary n-cube [184] network is a single-stage network which connects $n2^n$ inputs to $n2^n$ outputs. This network is derived from the indirect binary n-cube by breaking the network after each group of switch units, to create more source and destination nodes. The indirect binary n-cube network may be re-arranged, by moving internal connections only, to the form shown in Fig. 5.26. This re-arrangement affects only the physical layout, and does not involve any re-numbering of sources or destinations, as does the conversion of the indirect binary n-cube into the omega network. The re-arranged network may be partitioned into n groups by breaking the network as shown in Fig. 5.27. The partitioned network is illustrated for $n = 2$ in Fig. 5.28, and for $n = 3$ in Fig. 5.29. The source and destination units are divided into n partitions, each of 2^n units, which may be numbered from 0 to $2^n - 1$, within the group. If the n-bit number i may be represented as a binary number $b_{n-1} \ldots b_2 b_1 b_0$, then, within the jth group, each input, s_i, may be connected to one of the outputs d_i or d_{i+2^j} in the next group, where the addition is performed using modulo 2^j arithmetic. This network may be arranged in a partial crossbar form, as shown in Fig. 5.30, and, for $n = 3$, this gives the connection matrix

$$Y = X = \begin{pmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{pmatrix}.$$

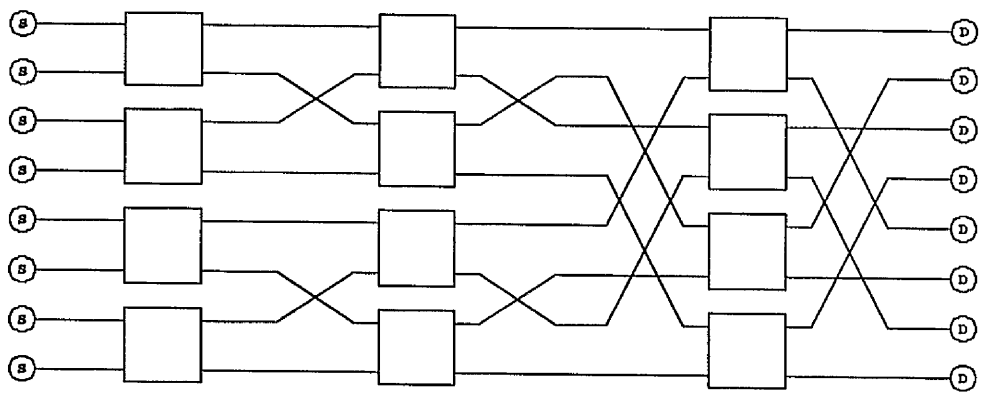


Fig. 5.26: The indirect binary 3-cube network, in a re-arranged form.

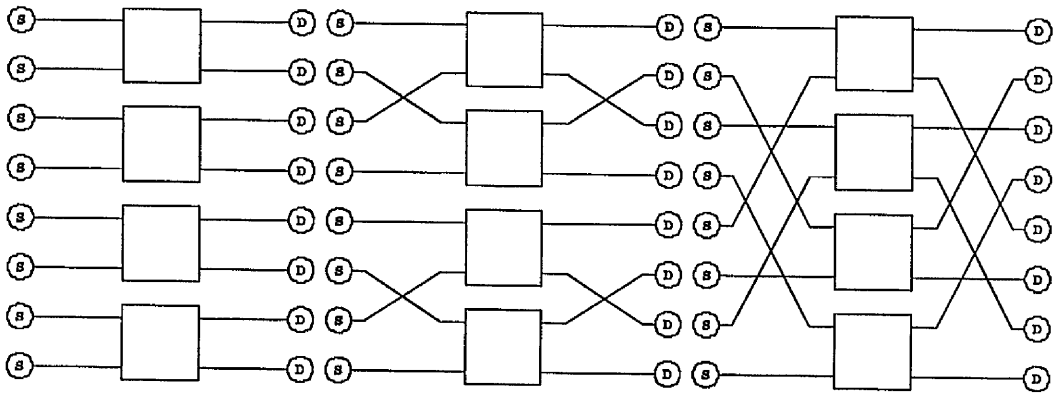


Fig. 5.27: The partitioned indirect binary 3-cube network.

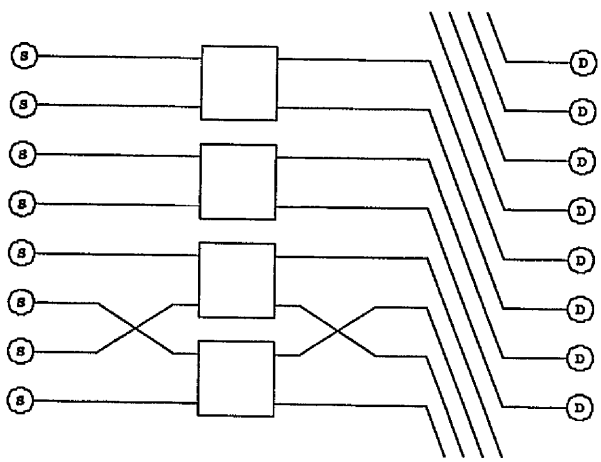


Fig. 5.28: The partitioned indirect binary 2-cube network, in a grouped form.

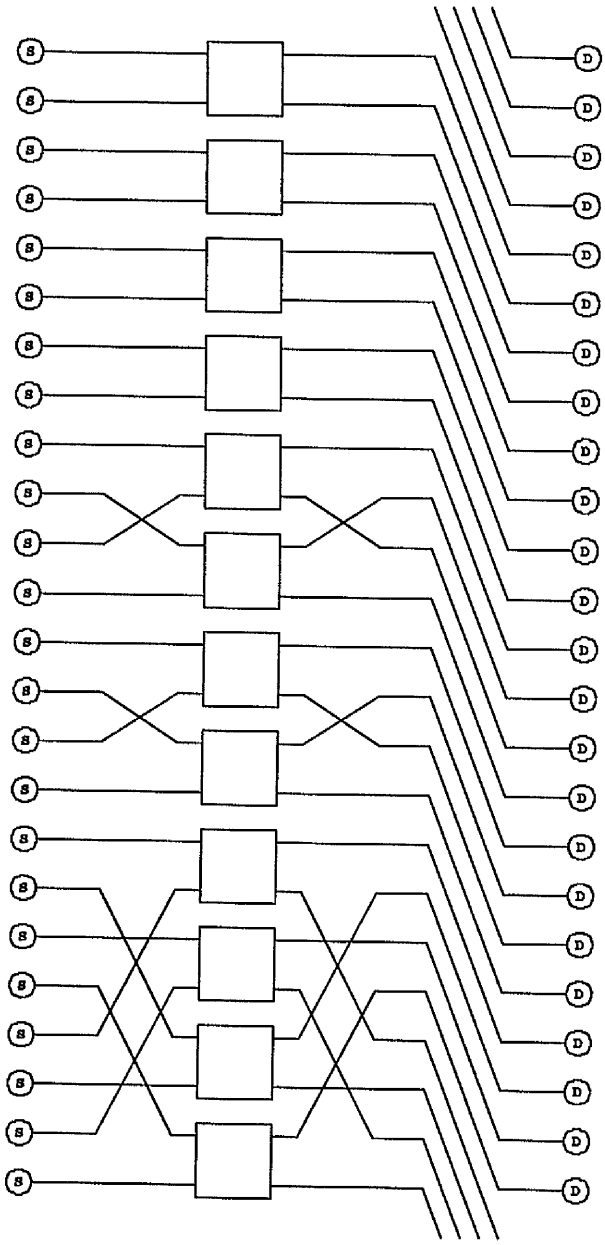


Fig. 5.29: The partitioned indirect binary 3-cube network, in a grouped form.

Table 5.12
Cost, Bandwidth and Latencies for
Partitioned Indirect Binary n-Cube Networks

n	$n2^n$	C(Y)	B(Y)	$L_{max}(Y)$	$L_{mean}(Y)$
2	8	24	3.49*	3	2.00
3	24	72	3.8*	5	3.26

* (Values calculated using Monte-Carlo method)

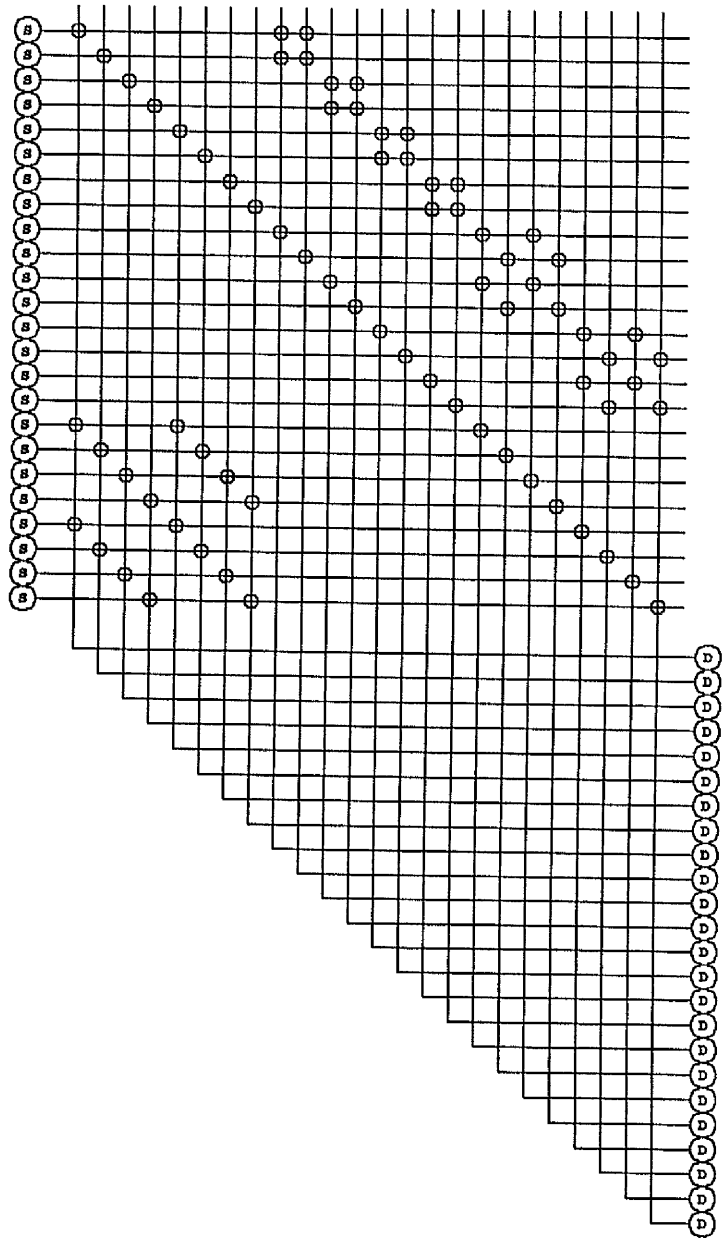


Fig. 5.30: The partitioned indirect binary 3-cube network in partial crossbar form.

The cost and and maximum latency for this network may be shown to be

$$C(Y) = 2^{n+1}(n+1),$$

$$L_{max}(Y) = 2n - 1,$$

and it is shown in appendix 1 that the mean latency is given by

$$L_{mean}(Y) = \frac{3 \times 2^{n-1} n(n-1) + n}{n2^n - 1}.$$

These are shown, with the bandwidth, in Table 5.12. It may be seen that the numbers of units connected by this network rises rapidly with n. For this reason, only two values could be calculated, but these show good bandwidth at low cost.

5.4 A Comparison of Interconnection Networks

A number of interconnection networks have been described, and some of their characteristics have been discussed. Table 5.13 shows values of $C(Y)$, $B(Y)$, $L_{max}(Y)$ and $L_{mean}(Y)$ for networks with approximately eight inputs and eight outputs (8×8 networks), taken from the tables in the preceding sections of this chapter. Table 5.14 shows values for networks which are about 16×16 in size. These tables are ordered according to the cost of the network. Networks which cannot be constructed in these exact sizes are placed in the tables according to their cost per connected unit.

The single time-shared bus may be seen to have a very low cost, but provides a very low bandwidth connection. The unidirectional ring network has the same cost as the single time-shared bus, and has a higher bandwidth, but the unidirectional ring suffers from a high latency. The binary tree network has a slightly higher cost, and its bandwidth is quite high, but, again, the latency is quite high. The shuffle-exchange network is marginally more expensive than the previous networks, but has a reasonably high bandwidth, and a moderate latency.

The bidirectional ring has a slightly higher cost than the shuffle-exchange network, and has a slightly higher bandwidth, but a much higher latency. The partitioned indirect binary n-cube has the same cost as the bidirectional ring, and provides a similar bandwidth, but with a reduced latency, which implies that this network may be preferable to a bidirectional ring, in many applications. The

Table 5.13
Cost, Bandwidth and Latencies for
Various Small Networks

Network	Size	$C(Y)$	$B(Y)$	$L_{max}(Y)$	$L_{mean}(Y)$
Single Time-Shared Bus	8	16	1.00	1	1.00
Unidirectional Ring	8	16	2.33	7	4.00
Binary Tree	7	19	3.10	4	2.29
Shuffle-Exchange	8	22	3.22*	3	2.11
Bidirectional Ring	8	24	3.48*	4	2.29
Partitioned 2-Cube	8	24	3.49*	3	2.00
3,3,2 Hierarchical Bus	8	28	1.34	1	1.00
2 Time-Shared Buses	8	32	2.00	1	1.00
Binary 3-Cube	8	32	4.27*	3	1.71
3×3 Lattice	9	45	5.22*	2	1.33
3 Time-Shared Buses	8	48	3.00	1	1.00
PM2I	8	48	6.13*	2	1.29
Indirect Binary 3-Cube	8	48	6.15*	1	1.00
ADM	8	64	6.16*	1	1.00
Full Crossbar	8	64	8.00	1	1.00

* (Values calculated using Monte-Carlo method)

Table 5.14
Cost, Bandwidth and Latencies for
Various Medium-Sized Networks

Network	Size	$C(Y)$	$B(Y)$	$L_{max}(Y)$	$L_{mean}(Y)$
Single Time-Shared Bus	16	32	1.00	1	1.00
Unidirectional Ring	16	32	2.5*	15	8.00
Binary Tree	15	43	3.72*	6	3.50
Shuffle-Exchange	16	46	3.6*	4	2.83
Bidirectional Ring	16	48	3.96*	8	4.27
Partitioned 3-Cube	24	72	3.8*	5	3.26
4,4,4,4 Hierarchical Bus	16	48	1.42*	1	1.00
2 Time-Shared Buses	16	64	2.00	1	1.00
Binary 4-Cube	16	80	5.3*	4	2.13
4×4 Lattice	16	80	5.42*	4	2.66
3 Time-Shared Buses	16	96	3.00	1	1.00
PM2I	16	128	8.24*	2	1.53
Indirect Binary 4-Cube	16	128	10.0*	1	1.00
ADM	16	176	10.0*	1	1.00
Full Crossbar	16	256	16.00	1	1.00

* (Values calculated using Monte-Carlo method)

hierarchical bus may be seen to have a much lower bandwidth than any other network of comparable cost, but this is primarily due to the even distribution of network traffic which was assumed in the calculation of the bandwidth value, discussed earlier. Multiple time-shared buses appear to have a relatively high cost, and provide only low bandwidth connections, although their latency is very low (unity).

The binary n-cube is a network which gives a high bandwidth, but has quite a high latency. The lattice network has similar cost, bandwidth and latency to the binary n-cube, for the networks shown. The PM2I network has a high bandwidth and low latency, but is an expensive network. The indirect binary n-cube has the same cost as the PM2I network, and provides a higher bandwidth connection, with lower latency. From this, the indirect binary n-cube would appear to be a superior network to the PM2I, for most purposes. The ADM has a higher cost than the indirect binary n-cube, but provides a bandwidth which is approximately the same, and each has a latency of unity. This would indicate that the extra expense of the ADM is not worthwhile. The full crossbar provides the maximum possible bandwidth, but is very expensive.

From these tables, it may be seen that the networks which show high bandwidth and low latency are expensive, but costs may be reduced if either low bandwidth or high latency may be tolerated.

5.5 Rook Polynomials for Bandwidth Calculation

It has been mentioned that the evaluation of the bandwidth function is a time-consuming process, for all but the smallest interconnection networks. However, it is possible that the time required to calculate this may be reduced by the application of rook polynomial techniques. The study of rook polynomials [216, 217] is a branch of combinatorial mathematics which is concerned with finding the number of arrangements in which a given number of rooks may be placed on an arbitrarily-shaped chessboard, in such a way that no rook threatens any other. This may be calculated without enumerating all possibilities, although the problem is still appreciable for large boards. This problem is similar to that of finding the number of different ways in which a connection may be made through an interconnection network represented by a matrix, since, in each case, selecting a particular square (or crosspoint) excludes

the use of any other square in the same row or the same column. This technique may be of use in two different ways:

i) It should be possible to write a program which uses rook polynomial methods to evaluate the bandwidth function, for any specified size and topology of interconnection network.

ii) For regular interconnection networks, it may be possible to use rook polynomials to derive a general analytic expression for the network bandwidth.

This technique may be applied directly to single-stage networks, since these may be represented by a single matrix, which corresponds to a single chessboard board, but for multi-stage networks, where sequences of matrices are involved, more advanced techniques would be required.

5.6 Summary

A number of taxonomies have been examined, and a new taxonomy proposed, using basic functional units connected by interconnection networks. A matrix-based representation scheme for interconnection networks has also been described, and it has been shown that useful parameters may be derived directly from this representation, and this has been illustrated for a number of published interconnection networks.

"I am ill at these numbers:
I have not art to reckon my groans;"

Polonius, in 'Hamlet, Prince of Denmark'
— William Shakespeare

Chapter 6:

Descriptions of Some Parallel Machines

6.1 Classification of Parallel Machines

A number of classes of machine are now be defined, using the notation introduced in chapter 5. Examples of the members of each class are given, but only machines of particular architectural significance are described here. A larger number of machines are described in appendix 2.

6.2 Sequential (von Neumann) Machines

A sequential machine comprises one IPU, one DPU and one DMU, as shown in Fig. 6.1. This corresponds to Flynn's SISD class. In matrix notation, this becomes

$$\begin{array}{lll} Y_{ii} = (0), & Y_{id} = (1), & Y_{im} = (0), \\ Y_{di} = (1), & Y_{dd} = (0), & Y_{dm} = (1), \\ Y_{mi} = (0), & Y_{md} = (1), & Y_{mm} = (0). \end{array}$$

This representation of the sequential machine has been described in detail in the previous chapter. All interconnections are implemented as 1×1 crossbars (direct connections). This class of machines encompasses most conventional machines; three well-known examples of are the Digital Equipment Corporation PDP11 [218], the Data General Nova [219], and the Manchester/Ferranti Atlas [220, 178].

6.3 Heterogeneous Sequential Machine with Multiple DPUs

In this class of machine, a single IPU supplies instructions to several DPUs which operate on shared data from a single memory unit (Fig. 6.2). Parallelism is exploited, without excessive software complexity, by using one IPU capable of supplying several DPUs with different instructions, which may be executed in parallel. Strict control of instruction sequencing is thus maintained by the IPU, and software is essentially sequential. Overlapping of instructions without register conflict is usually performed by the hardware. This is a commonly encountered structure, and is found in such machines as the CDC STAR-100 [222, 223, 224, 225], the Delft University DIP-1 [227], and PICAP II [229, 230, 231, 5], and also in the

Fig. 6.1: A sequential machine.

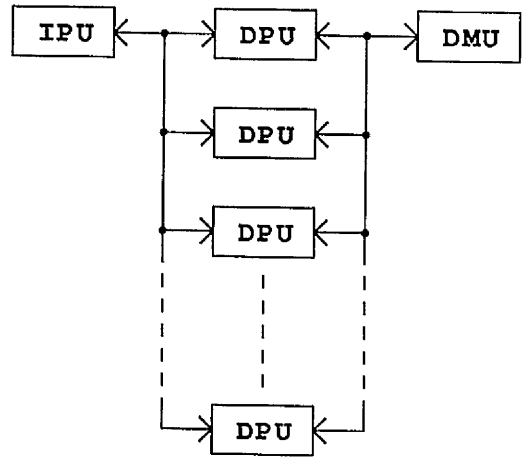


Fig. 6.2: A heterogeneous sequential machine with multiple DPUs.

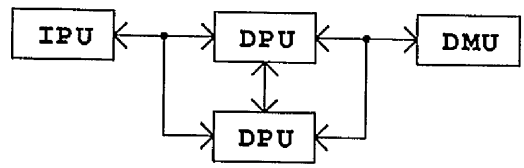


Fig. 6.3: A vector-serial machine.

central processing units of the CDC 6600 [232], the CDC 7600 [224], and the Texas Instruments ASC [233, 234, 223, 161, 224]. The matrix representation of these machines is

$$\begin{array}{lll} Y_{ii} = (0), & Y_{id} = (1), & Y_{im} = (0), \\ Y_{di} = (1), & Y_{dd} = *, & Y_{dm} = (1), \\ Y_{mi} = (0), & Y_{md} = (1), & Y_{mm} = (0). \end{array}$$

* – See below.

The single IPU communicates with all DPUs, so $Y_{id} = Y_{di} = (1)$. The DMU is connected to all DPUs, and so $Y_{dm} = Y_{md} = (1)$. In most machines of this type, the DPUs do not communicate directly with each other, and so $Y_{dd} = (0)$. In some machines, however, this is not so. The IBM 190/91 [224] and the CDC Cyber 205 [235, 224] allow data to be passed from one DPU to another, without the necessity of placing the result in memory first. All data paths, other than Y_{dd}, are normally implemented as full crossbars, since high bandwidth is required, and Y_{id} may be a broadcast-mode connection. These machines may have a store distributed across several physical units, but they are conceptually one, since any processor may access any memory location.

Some important sub-classes of the heterogeneous sequential machine class exist, and are now briefly described.

6.3.1 Vector-Serial and Array-Serial Machines

Vector-serial and array-serial machines are specialised towards processing one-dimensional and two-dimensional data structures, respectively. In their most basic form, these machines comprise an IPU, i, a main DPU, d_m, a DPU for manipulating addresses, d_a, and a DMU, m (Fig. 6.3). These are interconnected by the matrix

$$\begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix},$$

where $s = \{s_i,\ s_{d_m},\ s_{d_a},\ s_m\}$, and $d = \{d_i,\ d_{d_m},\ d_{d_a},\ d_m\}$. The secondary DPU, d_a, is dedicated to generating a sequence of memory addresses, performing memory read operations, passing the retrieved data to the main DPU, d_m, and replacing results in memory. This data streaming unit may be a very simple device, and is little more than a counter in a machine such as PICAP I [237, 238], whereas in

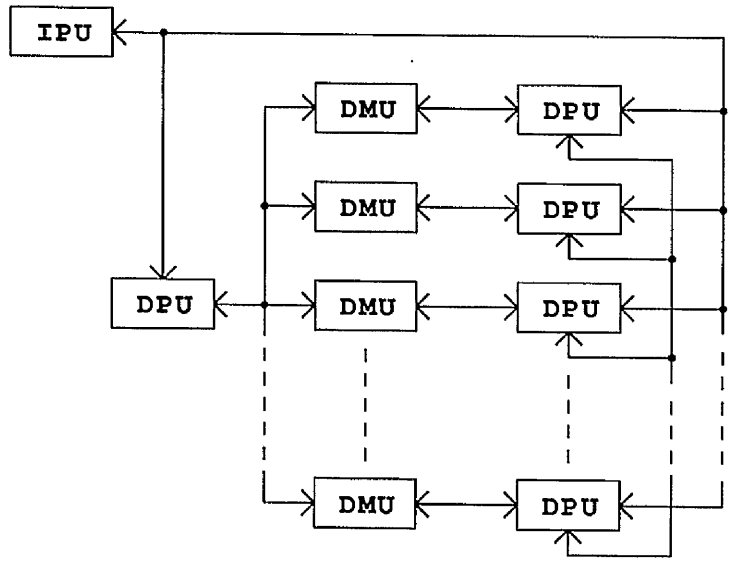


Fig. 6.4: An orthogonal machine.

MU5 [240, 241, 224], d_a is a full integer arithmetic unit. In array-serial machines, the secondary processor may be specialised towards two-dimensional array address generation, as in the Magiscan 2 [118].

6.3.2 Orthogonal Machines

Another sub-class of the heterogeneous sequential machine class is that of the orthogonal machines. In these machines, a single IPU supplies a conventional word-oriented DPU, in addition to a bit-slice oriented unit comprising a number of identical DPUs (Fig. 6.4). These are commonly referred to as the horizontal and vertical processing units. The matrix representation is

$$\begin{array}{lll} Y_{ii} = (0), & Y_{id} = (1), & Y_{im} = (0), \\ Y_{di} = (1), & Y_{dd} = *, & Y_{dm} = *, \\ Y_{mi} = (0), & Y_{md} = *, & Y_{mm} = (0). \end{array}$$

* – See below.

The IPU is connected to all DPUs, and so $Y_{id} = Y_{di} = (1)$. The sub-matrices Y_{dm} and Y_{md} are composed of identity matrices with one-matrices juxtaposed, since the horizontal DPU may access all DMUs, whereas each vertical DPU may only access its corresponding DMU. Y_{dd} varies from machine to machine.

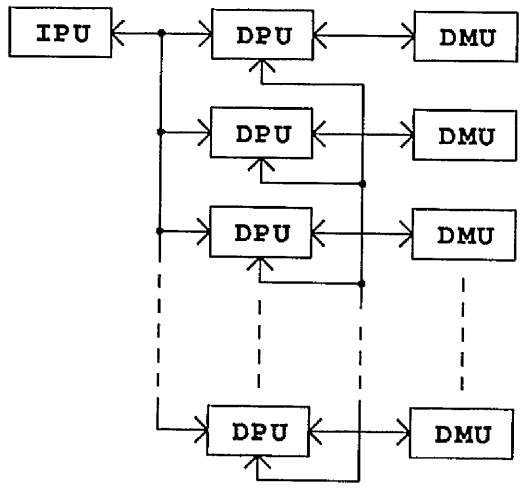


Fig. 6.5: A homogeneous sequential machine with multiple DPUs.

Examples of this sub-class are the OMEN machines [243, 244], and the ICL distributed array processor (DAP) [245, 247, 248, 250, 251]. In the DAP, the vertical unit is itself arranged in a two-dimensional manner, and so Y_{dd} represents a two-dimensional lattice interconnection. The orthogonal machines could also be considered to belong to the next class, the homogeneous sequential machines, since most (although not all) of the DPUs are identical.

6.4 Homogeneous Sequential Machine with Multiple DPUs

These machines are similar to the heterogeneous sequential machines, but all of the DPUs in the machine are identical. A single IPU supplies an instruction stream to the DPUs, each of which has its own DMU, as shown in Fig. 6.5. For this class of machines,

$$Y_{ii} = (0), \quad Y_{id} = (1), \quad Y_{im} = (0),$$
$$Y_{di} = (1), \quad Y_{dd} = *, \quad Y_{dm} = I,$$
$$Y_{mi} = (0), \quad Y_{md} = I, \quad Y_{mm} = (0).$$

* – See below.

The IPU is connected to all DPUs, so $Y_{id} = Y_{di} = (1)$, and, since each DPU connects to one DMU, $Y_{dm} = Y_{md} = I$, the identity matrix. Y_{id} is, again, a broadcast-mode connection, to allow all DPUs to receive the same instruction simultaneously. The inter-processor connection pattern, defined by Y_{dd}, varies from machine to machine, but some interconnection networks are commonly encountered.

Y_{dd} may be (0), giving no inter-processor connections, as in PEPE [252, 253, 254, 154, 255]. In STARAN [256, 257], Y_{dd} was (0), but Y_{dm} was, instead of simply an identity matrix, a network based on the indirect binary n-cube [190]. This allows efficient implementation of algorithms such as the fast Fourier transform, since the elements of arrays may be easily interchanged according to the perfect shuffle of their address. The Coulter Diff3 and Diff4 machines [258, 251] are examples of homogeneous sequential machines which are specialised towards image processing applications. In these machines, specialised DPUs perform Golay primitive operations [259].

The rectangular lattice network is a common inter-processor connection pattern (Y_{dd}) for homogeneous sequential machines. A number of variants on the basic network exist, and these may connect each DPU to four, six or eight neighbouring units. In general, machines designed for image manipulation favour eight-neighbour connections, whilst machines used for array calculations have four neighbour links. Processors at the edge of the array, with less than the requisite number of neighbours, may use wrap-around connections to connect opposite edges of the array, forming a toroidal structure. Alternatively, these edge units may connect to a single 'boundary' point.

ILLIAC IV [260, 261, 262, 263], comprised 64 processing units, each containing a 64-bit floating-point processor with 2048 words of store. These were connected in a square array, with each processor linked to four neighbours. In CLIP4 [264, 265, 266, 267, 251, 268, 250], 9216 processors are connected in a 96×96 square array, with eight-neighbour links. Here, the DPUs are simple 1-bit processors, and the DMUs are limited to 32 bits of store. The Massively Parallel Processor (MPP) [269, 270, 268, 271] comprises a 128×128 array of processors, with four-neighbour connectivity. Each processing unit has a one-bit ALU, six one-bit registers, and a variable-length shift register. A 32-bit store is provided on-chip, but provision is made for additional external memory.

PCLIP [273] is a pyramid-based machine, in which the processing units are arranged in a number of layers, which decrease in size towards the apex of the machine. Each layer is interconnected by a rectangular lattice network, and the processors are also connected to the layer above and below their own. Pyramid structures are claimed to be suited for some forms of image analysis, since an image may be stored and processed using a range of spatial resolutions in the differently-sized layers of the pyramid.

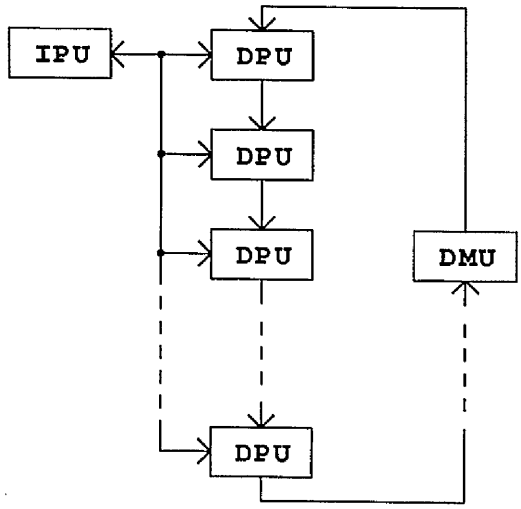


Fig. 6.6: A systolic machine.

Homogeneous sequential machines which use the binary n-cube [205], the indirect binary n-cube [184], the PM2I network [186], and the shuffle-exchange network [213], as Y_{dd} have also been proposed.

6.4.1 Systolic Machines

In a systolic machine, a number of DPUs are interconnected to perform manipulations on a single data stream (Fig. 6.6). Each DPU performs some part of the overall calculation, then passes its partial results to the next DPU. The advantage of these machines is that the processing elements may be made very simple, and the structures used are amenable to VLSI implementation. These machines are regarded as homogeneous sequential machines, since, although each DPU may have its own IPU, these must be globally synchronised, and therefore may be regarded as separate parts of one large IPU. This machine structure may be represented by the matrices

$$Y_{ii} = (0), \qquad Y_{id} = (1), \qquad Y_{im} = (0),$$

$$Y_{di} = (1), \qquad Y_{dd} = *, \qquad Y_{dm} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{pmatrix},$$

$$Y_{mi} = (0), \qquad Y_{md} = (1\ 0\ 0\ \ldots\ 0\ 0) \qquad Y_{mm} = (0).$$

* – See below.

The IPU broadcasts instructions to all DPUs, so $Y_{id} = Y_{di} = (1)$, but, in general, only a small number of DPUs connect directly to the DMU. The inter-processor connections, Y_{dd}, may be one of several possible networks, including the ring, the linear array, or the lattice network.

One example of a systolic machine designed for image processing is the Cytocomputer [275, 276, 277, 278, 45, 280] which comprises 113 sequential DPUs connected together in a linear array. Each DPU performs a simple processing operation using 3×3 neighbourhoods, as the image is passed sequentially through the systolic pipeline. The DIP-1 [227, 281] may also be considered to be systolic, though in this case the pipeline is dynamically reconfigurable, and so may be considered to be a heterogeneous sequential machine.

Kung et al. [282, 283] have developed a number of unnamed systolic machines, based on fixed program processing elements. The program and interconnection pattern depend on the calculation to be performed. Kuhn [284], Yen and Kulkarni, [285, 286], Ahmed [287], and Chuang [288] also describe a number of similar systolic structures. The bagel [289] is a systolic machine with a skewed torus interconnection. The skew allows the machine to function as a linear array or as a rectangular array.

6.5 Unshared-Memory Multiprocessors

The unshared-memory multiprocessors are a class of machines in which a number of essentially independent computing machines are connected by an inter-processor network, Y_{dd}. Each DPU is connected to its own IPU and private DMU, as shown in Fig. 6.7. The matrix representation of this class is

$$\begin{array}{lll} Y_{ii} = (0), & Y_{id} = I, & Y_{im} = (0), \\ Y_{di} = I, & Y_{dd} = *, & Y_{dm} = I, \\ Y_{mi} = (0), & Y_{md} = I, & Y_{mm} = (0). \end{array}$$

* – See below.

Each IPU connects to just one DPU, therefore $Y_{id} = Y_{di} = I$. Similarly, each DPU connects to only one DMU, and so $Y_{dm} = Y_{md} = I$. Y_{dd} may be one of many possible interconnection patterns, and is not necessarily fully-connected.

In ZMOB [290, 291, 292], 256 Z80A-based processors use a packet-switched ring structure to provide $Y_{dd} = (1)$.

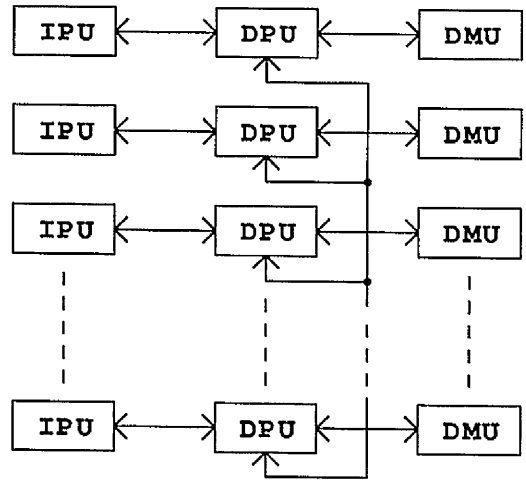


Fig. 6.7: An unshared-memory multiprocessor machine.

EMMA [293] is a multiprocessor machine with a hierarchical bus interconnecting the processing units. Thus, a higher bandwidth path is provided from each DPU to other local DPUs, than exists to non-local DPUs. The use of this type of network assumes some locality of communications, which does exist in certain processing tasks.

PASM [294, 295, 296, 297, 298] is a machine intended for image analysis tasks, in which a set IPUs, operating independently, each control a number of DPUs in an SIMD mode. Each DPU is connected to a double-buffered DMU. Y_{dd} is to be fully-connected, using either the indirect binary n-cube network or the augmented data manipulator [193].

X-Tree [206, 207, 299] is made up of a number of 'X-nodes', arranged as a binary tree, augmented by ring connections at each level. Each X-node comprises a full processing unit with private memory, and a four-way message passing switch unit, similar in many ways to the transputer [301, 302, 304].

The WAP [305] is described as a 'wavefront architecture' machine, and is based on a rectangular lattice network. It is similar in many ways to the systolic machines, but differs in that the processors are not fixed-program devices, and that the machine is not globally controlled, but relies on local synchronisation between processors when passing data across the array.

Ethernet [306, 307] is a high-bandwidth communications mechanism, based on the time-shared

bus, and this may be used to connect independent machines to form unshared-memory multiprocessor systems.

The connection machine [308, 309, 310] is designed to implement connectionist cognitive models [311, 312, 314]. The prototype connection machine contains 65536 processing elements or 'cells'. Each cell is a finite state machine, and contains an IPU, a DPU and a DMU. When the cell receives a message from some other cell, the cell state and message type are combined to determine the appropriate action. The cells communicate via a packet-switched network with a binary 12-tube topology.

The general operator processor (GOP) [315, 316] is a machine specially designed for image analysis applications. The GOP comprises a conventional bit-slice processor, connected to four vector-serial pipeline units, which are specialised for space-domain convolution operations.

A number of multiprocessor machines have been designed using a pyramid structure, composed of several layers of processors [317]. These machines are similar in structure to the homogeneous sequential pyramid machines described earlier, but, in this case, each layer is, itself, a lattice-connected homogeneous sequential machine with its own instruction processing unit. In some machines, such as PAPIA [319] and SPHINX [321], all processing units are identical, but, in others, more powerful processors are used towards the top of the pyramid, as in Uhr's machine [322, 323], so that the processing power of each layer is approximately equal.

6.6 Shared-Memory Multiprocessor

In machines of this class, a number of DPUs with individual IPUs operate on a shared memory (Fig. 6.8). This memory may be contained in a single physical unit, but more commonly it is distributed across several memory units. This class may be represented by the matrices

$$\begin{array}{lll} Y_{ii} = (0), & Y_{id} = I, & Y_{im} = (0), \\ Y_{di} = I, & Y_{dd} = *, & Y_{dm} = (1), \\ Y_{mi} = (0), & Y_{md} = (1), & Y_{mm} = (0). \end{array}$$

* – See below.

$Y_{id} = Y_{di} = I$, since each IPU is paired with one DPU, and $Y_{dm} = Y_{md} = (1)$, since each DPU may access the single DMU. Y_{dd} is often (0), with inter-processor communication taking place via the shared

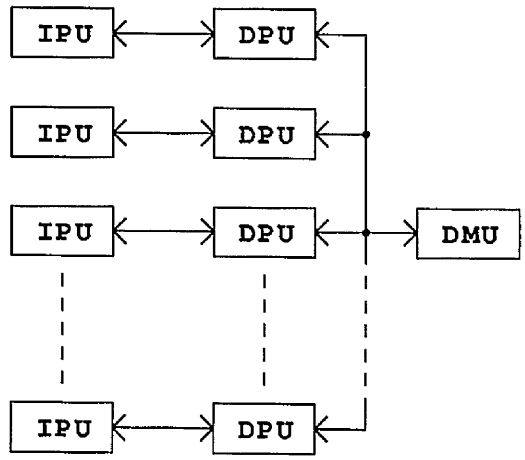


Fig. 6.8: A shared-memory multiprocessor machine.

memory. The networks Y_{dm} and Y_{md} are usually (1), in most machines, but implementation methods vary considerably.

The Paracomputer [324] is the archetype of this class. It is a machine in which all DPUs may access the DMU ($Y_{dm} = Y_{md} = (1)$) without any interference whatsoever. This is not, however, practical to implement, and is defined only to give a 'yardstick' against which to measure the performance of other members of this class.

The Ultracomputer [325, 326, 327] is designed to implement, as far as possible, the ideal Paracomputer. The design uses a large number of processing units (up to 4096), each with some private memory, connected to a distributed global store by a packet-switched omega network. The processing units are based on the CDC 6600 CPU, and the memory units have additional hardware to queue requests, and to perform fetch-and-add instructions.

C.mmp [328] consists of sixteen DEC PDP-11 processors connected by a circuit-switched full crossbar network to sixteen 64K × 16-bit memory units. This provides a high memory bandwidth, but it is an expensive interconnection network to implement, and is only practical for small numbers of devices.

In Cm* [329, 330, 331, 332], $Y_{dm} = Y_{md} = (1)$ is implemented as a three-level hierarchical bus

network. Each processing unit is a DEC LSI-11 which has a direct link to a local 64K-word memory unit. Up to fourteen processors make up each cluster, which are, in turn, connected by an intra-cluster bus, arbited by a microprogrammed bus controller.

The AHR [333, 334] is designed to execute LISP programs by parallel evaluation of function arguments. Up to 64 Z80-based processing units share a special structured memory unit, known as 'the grill', which stores LISP programs. Access to the grill is controlled by a device known as 'the distributor', which also allocates work, in the form of nodes to be evaluated, to idle processors. Communication between the processing units and the distributor is by two time-shared buses.

CHOPP [335, 336] is designed to have of the order of 10^5–10^6 custom VLSI processing units, interconnected by a packet-switched binary n-cube network. A shared memory is distributed across the machine. The processors are to be implemented using custom VLSI chips.

The C-VAS 3000 system [337] is a multiprocessor machine specialised towards image analysis. A Motorola MC68000 series processor running UNIX† is used for system management and execution of Pascal programs. A bit-slice processor is used for execution of image analysis operations, and a memory address processor is provided for image address calculations. A two megabyte program and data store is shared by the MC68000 and the bit-slice processor, in addition to the image store of between one and eight megabytes.

PUMPS [338] is a shared-memory multiprocessor, specially designed for image analysis, in which a number of task processing units, interconnected by a time-shared bus, are connected via a delta network to a shared memory. In addition to a shared-memory cache, each processor has a private memory, with its own cache. Block transfers are carried out, through the delta network, between the local stores and the shared memory. Both shared and private memory areas use paged virtual memory systems.

ALICE [340, 341] is a shared-memory multiprocessor based on a graph-reduction model of computing. In this machine, a program exists in the form of a large number of data packets which are placed in a shared store known as the 'pool'. Each data packet comprises an operator, some operands and a

† UNIX is a registered trademark of AT&T Bell Laboratories in the USA and other countries.

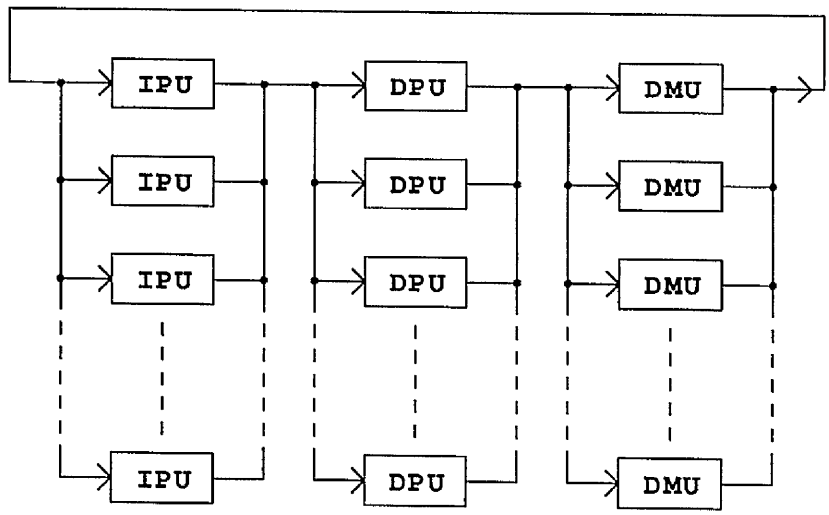


Fig. 6.9: A data flow machine.

pointer to the destination of the result. A number of processing 'agents' repeatedly remove data packets from the pool and re-write the packet, if this is possible. By repeated application of this process, all packets are evaluated, and the program is, thus, executed.

6.7 Data Flow Machines

This is a class of machines in which the availability of data determines the sequence of executed instructions. Thus, one or more IPUs are driven by DMUs, in such a way that when all of its operands are available, an instruction is decoded and passed, with its operands, to be executed in one of several DPUs. The machine structure, shown in Fig. 6.9, may be represented by the matrices

$$Y_{ii} = (0), \quad Y_{id} = *, \quad Y_{im} = (0),$$
$$Y_{di} = (0), \quad Y_{dd} = (0), \quad Y_{dm} = (1),$$
$$Y_{mi} = *, \quad Y_{md} = (0), \quad Y_{mm} = (0).$$

* – See below.

Each IPU connects to one or more DPUs, each DPU connects, in turn, to every DMU, and each DMU connects back to at least one of the IPUs. In this way, the machine forms a ring structure, around which data items and executable packets circulate.

The Manchester data flow machine [342, 343, 344, 345, 346, 347] implements each IPU, DPU and DMU as separate units. Data tokens are stored in a matching store (DMU), until all operands required for an operation are present. When the last operand required for an instruction arrives at the matching store, an executable packet is passed to the node store (IPU), where the instruction is obtained, and the packet proceeds to one of sixteen DPUs. Results are sent from the DPU, back to the DMU. Multiple rings allow more than one IPU or DMU to be used in the system.

In the MIT data flow machine [348, 349], programs are stored in a special VLSI structure composed of 'instruction cells', which contain a DMU and an IPU. A set of DPUs are connected to these cells, by a switching network similar to a delta network.

The generalised control flow machine [350] is based on similar principles to data flow, but the readiness of data is indicated by sending control tokens, instead of sending the entire data item itself. Data items reside in a conventional store, and control tokens are passed along a packet-switched ring network containing a number of different units. Code memory processors (IPUs) change 'next instruction address' (NIA) tokens into instructions, and data computation units (a combined DPU and DMU) perform computations, and produce more NIA packets.

6.8 Stochastic Machines

These are special-purpose pattern recognition machines which rely on statistical properties of their input data. Their basic principle of operation is that forming a relatively small number of pseudo-random functions of a large input data set may be representative of the input pattern. Presenting a similar pattern, at a later time, may result in similar output functions which may be matched with the stored pattern. The WISARD [352, 353] is a stochastic machine specialised for visual pattern recognition. A group of fifteen memory units, $m_{d1} \ldots m_{d15}$, are loaded with a training set, such that when their addressing inputs are presented with a pseudo-random function of a particular input image (stored in an image memory, i_i), the outputs of these memory units will be affirmative. These groups are known as discriminators. The machine is co-ordinated by a conventional processor, consisting of an IPU, i_c, a DPU, d_c, and a DMU, m_c. This reads the response from several discriminators, and, thus, classifies the

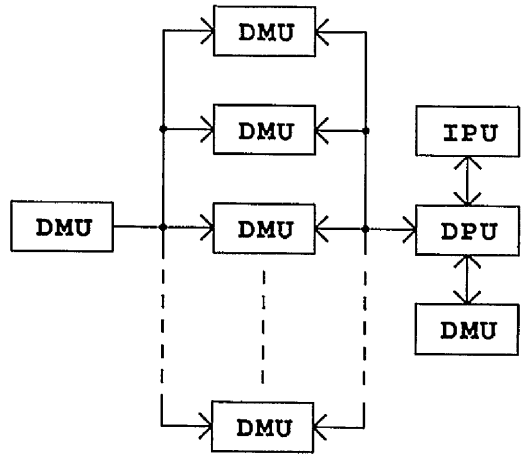


Fig. 6.10: A stochastic machine.

image. This machine is illustrated in Fig. 6.10, and may be represented by the matrix

$$
\begin{array}{c}
 \\
i_c \\
d_c \\
m_c \\
m_i \\
m_{d1} \\
\vdots \\
m_{d15}
\end{array}
\begin{array}{c}
\begin{array}{ccccccc} i_c & d_c & m_c & m_i & m_{d0} & \cdots & m_{d14} \end{array} \\
\left(\begin{array}{ccccccc}
0 & 1 & 0 & 0 & 0 & \cdots & 0 \\
1 & 0 & 1 & 0 & 0 & \cdots & 0 \\
0 & 1 & 0 & 0 & 0 & \cdots & 0 \\
0 & 0 & 0 & 0 & 1 & \cdots & 1 \\
0 & 1 & 0 & 0 & 0 & \cdots & 0 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\
0 & 1 & 0 & 0 & 0 & \cdots & 0
\end{array} \right)
\end{array}.
$$

Here, the machine is represented by one large matrix, since, in this case, the partitioned matrix form is not composed of simple sub-matrices. It may be seen that the IPU, i_c, the DPU, d_c, and the DMU, m_c, are connected in exactly the same manner as the sequential processor described in the previous chapter. The image memory, m_i, is connected to every discriminator memory, which are in turn connected to the DPU.

This type of machine is fast and trainable, but the image data is somewhat undersampled, and the number of classes of objects which can be distinguished is limited by the number of discriminators. These factors limit the applications of such machines.

6.9 Summary

A number of machine classes have been defined, and these have been described in terms of a notation involving basic functional units, and interconnection networks represented by matrices. These machine classes are not necessarily complete, but the taxonomy has been shown to be capable of representing all of these classes, and it is believed that any other machine classes could also be described in this way. The next chapter discusses the applicability of these classes of machine to the tasks involved in image analysis.

Each grain of sand is different.
No two are alike.
Some are similar in one way,
some are similar in another way,...

You'd think the process
of subdivision and classification
would come to an end somewhere,
but it doesn't.
It just goes on and on.

'Zen and the Art of Motorcycle Maintainance'
— Robert M. Pirsig

Chapter 7:
Applicability of Hardware Structures to Image Analysis Problems

7.1 Sequential (von Neumann) Machines

The use of sequential machines in image analysis has already been discussed in chapter 4. It was concluded that these machines, although easier to program, have inherent speed limitations, and are, therefore, not a cost-effective solution for many large tasks, such as image analysis.

7.2 Heterogeneous Sequential Machine with Multiple DPUs

Two sub-classes of the heterogeneous sequential machine class have been described in chapter 5. Machines in the vector-serial sub-class can usually achieve good performance when the required operations are expressed in a vector form. This is feasible for most pixel-oriented image processing applications, but is not usually possible for feature-oriented image analysis. Where such vectorisation is possible, the processing rate becomes limited by the speed of the main data processing unit, which must be a fast and, therefore, expensive component. In pipelined vector-serial machines, the time taken to fill the processing pipeline may be significant, and this may affect the performance for operations on relatively short vectors [223, 224].

Orthogonal machines normally have a large number of processing units in the 'vertical' direction, and again, this may be used efficiently to exploit pixel-oriented parallelism. However, it is necessary, in these machines, to use the scalar or 'horizontal' data processing unit to deal with any feature-oriented operations. In many cases, this may severely limit the overall performance of the machine, if the scalar processor is relatively slow in comparison to the overall processing capability of the 'vertical' DPUs, since the latter remains idle for a high proportion of the processing time.

In general, machines in the heterogeneous sequential machine class tend to have a small number of DPUs, and, although it may be possible to keep all of these units busy, the overall speedup is limited by the number of processors.

7.3 Homogeneous Sequential Machine with Multiple DPUs

Within the homogeneous sequential machine class, the lattice-connected machines are the most numerous, and these have many advantages in image processing. Pixel-oriented parallelism is easily expressed in a form which may be efficiently executed on these machines, and the inter-processor connectivity of these machines reflects the local connectivity patterns frequently encountered in image processing. The use of a local DMU for each DPU reduces the overhead of pixel address calculations, since much of this is implicit in the location of the processor.

Large numbers of DPUs are used in these machines, and it is necessary, therefore, that these are simple units, and are usually limited to single-bit operation. This does, however, allow the use of variable-length arithmetic, which is proportionately faster than fixed word-length arithmetic and, therefore, gives better use of resources. When image processing operations on binary images, or images of only a few bit planes, are performed on bit-parallel machines, most of the multi-bit arithmetic unit is unused for most instructions. In such cases, bit-serial operations allow arithmetic operations to use the minimum required word-length. The simplicity of the processing element may, however, also cause problems, in that programs may be long, and many instructions may be required for relatively simple operations [250].

The rigid SIMD control structure of the homogeneous sequential class forms a major restriction in its use. Conditional execution of statements is only possible by causing sets of DPUs to idle, whilst others execute instructions. This means that all possible branches of conditional statements must be decoded by the IPU, which may be a time-consuming process. This restriction, that all processors must execute the same instruction at the same time, means that feature-oriented parallelism is not easily exploited and, in general, only the pixel-oriented parts of image analysis tasks are performed on these machines.

Fixed-program systolic machines have been used for image processing [286], but their use is, again, limited to pixel-oriented operations. Other systolic machines, such as the Cytocomputer [278, 277, 45] are well suited to pixel-oriented operations, but the maximum parallelism available, in this linearly-connected machine, is the number of sequential stages into which the processing task may be divided.

In STARAN, pixel-oriented parallelism could be effectively used by allocating one processor to each pixel [354]. The flip network, which connected the DPUs to the DMUs, could then be used to provide displaced image data and, thus, provide rapid access to both local neighbourhood data and data displaced by powers of two, as required for operations such as the Fast Fourier Transform.

In general, the homogeneous sequential class appears suitable only for pixel-oriented image processing operations, and not for feature-oriented image analysis tasks.

7.4 Unshared-Memory Multiprocessors

To perform any form of image manipulation on unshared-memory multiprocessors, the image must be either copied to all the processors, or divided into sections and these distributed to the individual processing units.

To deal with pixel-oriented parallelism, the image may be easily partitioned across the machine, and good performance may be obtained in this way [355, 356, 296, 297]. In the case of feature-oriented tasks, however, features are not usually arranged in such a way as to permit the image to be easily divided into sections, and so each processor must be able to examine the entire image. This, in turn, requires the provision of one copy of the image for each processing unit. To co-ordinate such a system would require a complex control structure involving a large amount of inter-processor communication, in addition to large amounts of image memory.

7.5 Shared-Memory Multiprocessors

Shared-memory multiprocessors have an advantage over the unshared-memory class, since an image may be partitioned by the processors, to take advantage of pixel-oriented parallelism, or the image may be examined in a feature-oriented manner by all processors simultaneously. This type of structure requires a high-bandwidth connection between the DPUs and the shared DMU, since all processors will require frequent access to this. This may be achieved by using a distributed store unit, and by use of local cache stores for frequently-accessed data. A suitable control strategy must also be developed for such a machine, to control and synchronise the actions of the independent DPUs.

7.6 Data Flow Machines

Data flow architectures seem to be capable of good performance when dealing with both pixel-oriented parallelism and feature-oriented parallelism. However, these machines, and the functional programming languages associated with them, are not well-suited to the partial modification of large data structures [357]. As this form of operation occurs frequently in feature-oriented image analysis operations, this would appear to limit the use of data flow machines in these applications.

7.7 Stochastic Machines

These machines appear to perform well for simple-shape matching operations, but would appear to have limited application, since they are incapable of performing many, more general, image processing and image analysis tasks.

7.8 Summary

The overall performance which may be obtained from any machine architecture appears to depend on the characteristics of the task which is to be performed. Many of the classes of machines which have been discussed seem to be suitable for tasks which involve highly repetitive pixel-oriented processing of images. However, most of these machines appear to exhibit fundamental problems when dealing with feature-oriented tasks. Of the classes described in this chapter, the shared-memory multiprocessor machines seem to be the only ones suited to deal with this form of task.

Since the importance of exploiting the available feature-oriented parallelism has already been emphasised in earlier chapters, it is important that machines of the shared-memory multiprocessor class be investigated for cost-effective image analysis systems. The following chapters describe a proposed structure for such a machine, and present the results of simulation of this machine.

Chapter 8:
A Proposed Machine Structure for Image Analysis: The Indirect Binary n-Tube

8.1 A Class of Machines Suitable for Image Analysis

The shared-memory multiprocessor class has been argued, in the preceding chapter, to be a suitable class of machines for use in image analysis. The machines in this class cover a large range of processing power and cost. These factors are determined by the choice of processing units, the memory size and technology, and the choice of interconnection networks. For machines with a large number of processors, the interconnection cost may be relatively high, compared with the cost of the processing elements since, for most networks, the cost grows with the number of connections at a faster than linear rate. In this chapter, a number of interconnection networks are considered which may be suitable for the implementation of a low-cost, shared-memory multiprocessor for image analysis applications.

8.1.1 Interconnection Networks for Shared-Memory Multiprocessors

The shared-memory multiprocessor class has been described, in chapter 6, in terms of the nine sub-matrices into which the machine interconnection matrix may be divided. Of these, eight are defined by the shared-memory multiprocessor class:

$$\begin{array}{lll} Y_{ii} = (0), & Y_{id} = I, & Y_{im} = (0), \\ Y_{di} = I, & Y_{dd} = * & Y_{dm} = (1), \\ Y_{mi} = (0), & Y_{md} = (1), & Y_{mm} = (0). \end{array}$$

* – See below.

Each IPU connects to one, and only one, DPU, so $Y_{id} = Y_{di} = I$, and each DPU may access the shared DMU, so $Y_{dm} = Y_{md} = (1)$. A suitable interconnection function must be selected for Y_{dd}, the inter-processor connection matrix. Since machine cost is a significant factor here, and direct processor-to-processor connections are not essential for this class of machine, no inter-processor connections will be made, that is, $Y_{dd} = (0)$. All communication must, therefore, take place by way of the shared memory

unit. These nine matrices define the required interconnection functions, and a suitable implementation must be selected for each one.

8.1.2 Implementation of Interconnection Networks

Of the nine interconnection matrices, only the implementation of the four non-zero matrices, Y_{id}, Y_{di}, Y_{dm} and Y_{md}, need be considered.

Y_{id} and Y_{di} are identity matrices, and connect each IPU to its corresponding DPU. These connections must be implemented as direct connections (1×1 crossbars). In practice, this will probably be unnecessary, since the DPUs and IPUs are likely to be single processing units.

Y_{dm} and Y_{md} are 1-matrices. This is an expensive function to implement, and so this requires discussion. Any of the networks described in chapter 5 could, in principle, be used to implement Y_{dm} and Y_{md}. A fully-connected network would be desirable, since these tend to have higher bandwidths, but their relatively high cost makes their use impractical for systems with a large number of processing units.

It would appear that the use of a partially-connected network is necessary, and this should have as low a cost as possible. To provide an interconnection function of (1), a packet-switched system must be used, where units are capable of forwarding data to other units, in cases where no direct path exists. In a partially-connected system, it is possible to combine the functions of Y_{dm} and Y_{md} in a single network. In the matrix representation, this combines the four matrices Y_{dd}, Y_{dm}, Y_{md}, and Y_{mm} into one larger matrix.

8.1.3 Packet-Switched Ring Structures

Of the partially-connected networks described in chapter 5, the simple unidirectional ring network has the lowest cost. It is capable of high throughput, but suffers from a high latency which makes its use undesirable in such a critical position as Y_{dm} or Y_{md}. Augmented ring structures based on other partially-connected networks may, however, reduce this latency, and the suitability of a number of such networks is investigated in the next section.

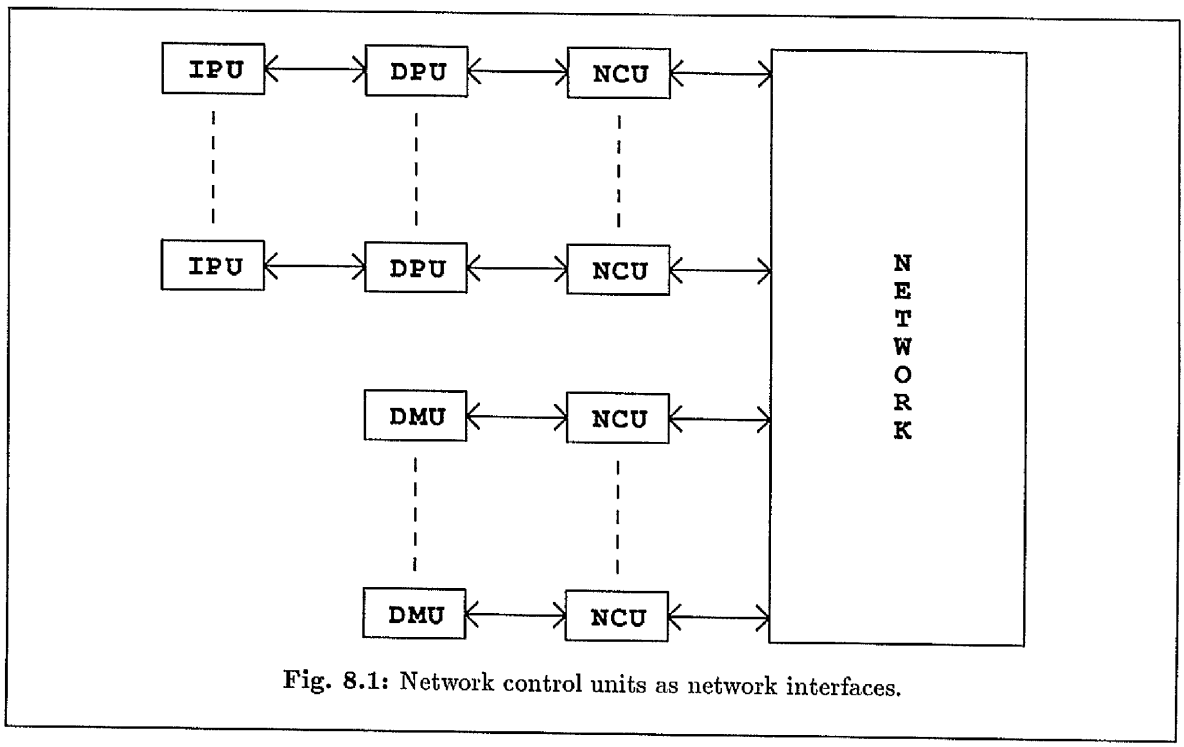


Fig. 8.1: Network control units as network interfaces.

In such ring-based structures, several possible paths may exist from one node to another, and special network control units (NCUs) may be required, to perform routing decisions on such a network. These units handle the transmission, forwarding and reception of data packets between processing units, memory units and the network, as shown in Fig. 8.1. Network control units are described in more detail in chapter 9.

8.2 Ring-Based Networks

A number of the ring-based networks described in chapter 5 may be suitable to connect the PUs (combined DPU/IPUs), the DMUs and their associated network control units. A number of these are now considered and compared in terms of the mean latencies which occur when a number of DPUs and DMUs are interconnected by these networks.

8.2.1 Latencies of Ring-Based Structures

The mean latency, $L_{mean}(Y)$, for a network has been defined as the mean number of passes through a network required for a packet to travel from one node to another, averaged over all possible destinations. In the simple unidirectional ring network, all PUs and DMUs are located on a single loop. In this case, the network control units may be simpler in structure than for some other networks, since no routing control is required. The main disadvantage of the ring structure is the high latency involved in any transfer. The mean latency for a ring which interconnects a total of p units is shown in appendix 1 to be given by

$$L_{mean}(Y) = \frac{p}{2}.$$

The bidirectional ring network provides two rings, which circulate data packets in opposite directions. Here, the optimal network routing algorithm is to use the ring which is passing tokens in the direction along which the desired destination is closer. It should be noted that this gives better performance than simply using one ring to implement each of Y_{dm} and Y_{md}. The mean latency for a bidirectional ring system which interconnects p units is shown in appendix 1 to be given by

$$L_{mean}(Y) = \begin{cases} \frac{1}{4}(p+1), & \text{if } n \text{ is odd,} \\ \frac{p^2}{4(p-1)}, & \text{if } n \text{ is even.} \end{cases}$$

These two simple ring networks exhibit a high latency, which increases approximately linearly with the number of units in the machine. The high latency of the unidirectional ring arises because packets must travel long distances on the return path even if the outward path is short. This is relieved somewhat in the case of the bidirectional ring but, for large p, the latency is still of order p. One possible solution to this problem is to place network control units on a number of rings, interlinked in some way, so that packets may cross between them when necessary.

8.2.2 The Partitioned Indirect Binary n-Cube as a Set of Rings

The partitioned indirect binary n-cube network may be considered as a set of 2^n rings, interconnected at each stage, as shown in Fig. 8.2 for $n = 2$. Packets circulate in one direction only, and are routed towards their destination by the network control units. The required routing algorithm is straightforward; at the ith stage, a packet should switch rings if the number of the next unit on the current ring differs from the packet destination unit number in the ith bit position (with a suitable unit numbering scheme, described in chapter 9). The mean latency for a packet-switched network with a partitioned indirect binary n-cube topology is shown in appendix 1 to be

$$L_{mean}(Y) = \frac{3.2^{n-1}n(n-1)+n}{n2^n-1}.$$

The number of units, p, connected by such a network is

$$p = n2^n.$$

For large networks, the latency becomes primarily dependent on n, rather than p. This is much better than the linear increase in latency given by the simple ring-based connections. A significant problem with this network is that the number of units p on a partitioned indirect binary n-cube grows exponentially with n, as shown in Table 8.1 This means that the choice of available sizes of network is limited, and so flexibility is reduced. A network is now considered which provides a greater flexibility and introduces an amount of fault-tolerance into the network.

8.2.3 The Partitioned Indirect Binary n-Tube

By repeating the basic pattern of the partitioned indirect binary n-cube, a set of longer rings may be produced (Fig. 8.3), which allow any multiple of the total unit count shown in Table 8.1 to be used. This is referred to, here, as a partitioned indirect binary n-tube, after its resemblance to a toroidal tube when constructed in three dimensions (Fig. 8.4). A tube which consists of R concatenated cubes will be termed an R-repeated tube. This structure provides a much greater flexibility in the size of system which may be constructed, and provides a mechanism whereby existing systems may be upgraded, by the extension of the tube network, to allow extra units to be added.

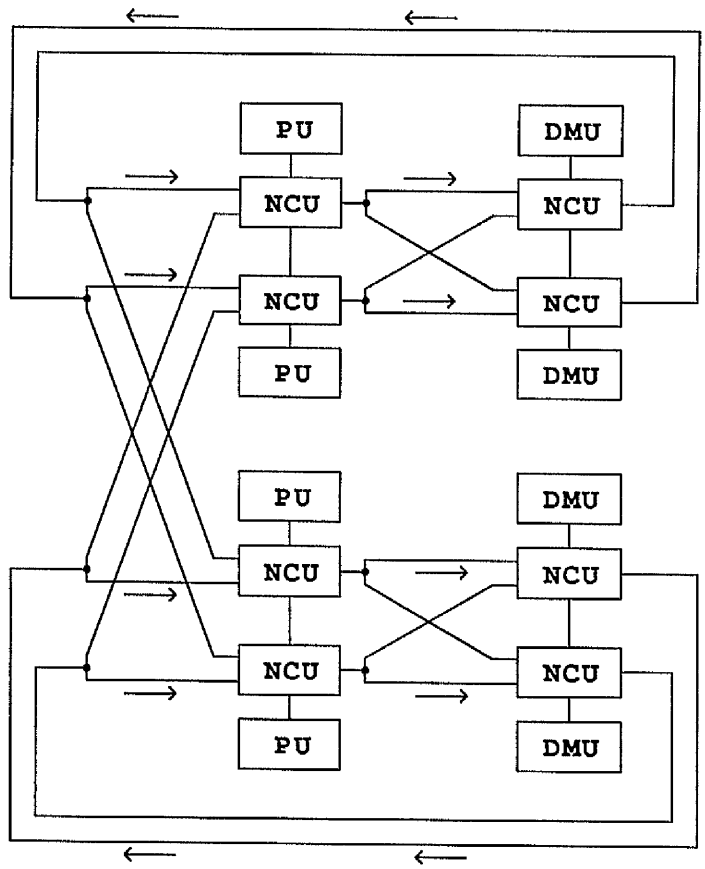


Fig. 8.2: The partitioned indirect binary 2-cube.

Table 8.1
Numbers of Units in Partitioned
Indirect Binary n-Cubes

n	Width	No. of Units
2	4	8
3	8	24
4	16	64
5	32	160
6	64	384
7	128	896
8	256	2048
9	512	4608
10	1024	10240

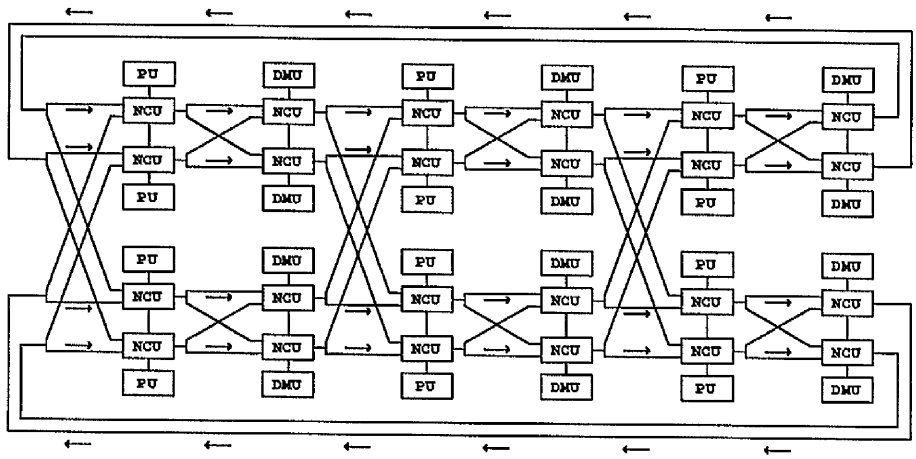


Fig. 8.3: A 3-repeated partitioned indirect binary 2-tube.

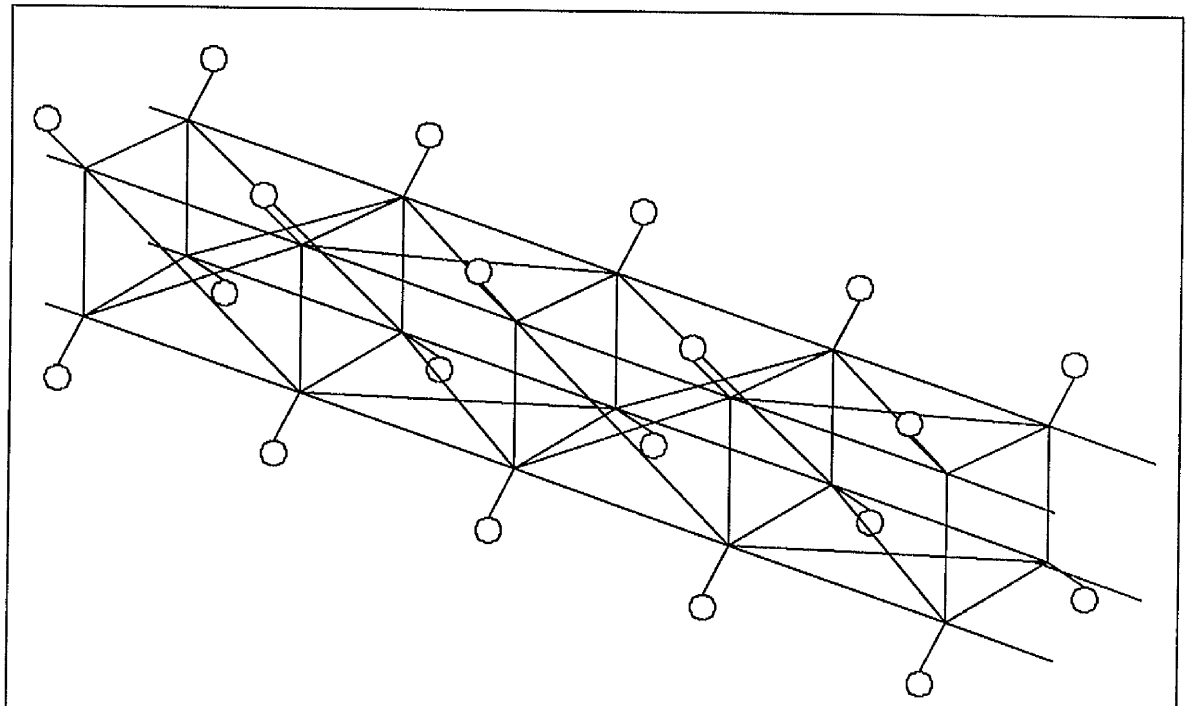


Fig. 8.4: A partitioned indirect binary 2-tube, constructed in three dimensions.

Table 8.2
Cost, Bandwidth and Latencies for
Partitioned Indirect Binary n-Tube Networks

n	R	$Rn2^n$	C(Y)	B(Y)	$L_{max}(Y)$	$L_{mean}(Y)$
2	1	8	24	3.49*	3	1.75
2	2	16	48	3.5*	5	2.75
2	3	24	72	3.1*	7	3.75
3	1	24	72	3.7*	5	3.13

* (Values calculated using Monte-Carlo method)

The partitioned indirect binary n-tube is a partially-connected network, and may be represented in a partial crossbar form and, from this, a matrix representation may be derived. For a 2-repeated partitioned indirect binary 2-tube,

$$Y = X = \begin{pmatrix} 1&0&0&0&1&1&0&0&0&0&0&0&0&0&0&0 \\ 0&1&0&0&1&1&0&0&0&0&0&0&0&0&0&0 \\ 0&0&1&0&0&0&1&1&0&0&0&0&0&0&0&0 \\ 0&0&0&1&0&0&1&1&0&0&0&0&0&0&0&0 \\ 0&0&0&0&1&0&0&0&1&0&1&0&0&0&0&0 \\ 0&0&0&0&0&1&0&0&0&1&0&1&0&0&0&0 \\ 0&0&0&0&0&0&1&0&1&0&1&0&0&0&0&0 \\ 0&0&0&0&0&0&0&1&0&1&0&1&0&0&0&0 \\ 0&0&0&0&0&0&0&0&1&0&0&0&1&1&0&0 \\ 0&0&0&0&0&0&0&0&0&1&0&0&1&1&0&0 \\ 0&0&0&0&0&0&0&0&0&0&1&0&0&0&1&1 \\ 0&0&0&0&0&0&0&0&0&0&0&1&0&0&1&1 \\ 1&0&1&0&0&0&0&0&0&0&0&0&1&0&0&0 \\ 0&1&0&1&0&0&0&0&0&0&0&0&0&1&0&0 \\ 1&0&1&0&0&0&0&0&0&0&0&0&0&0&1&0 \\ 0&1&0&1&0&0&0&0&0&0&0&0&0&0&0&1 \end{pmatrix}.$$

The cost and latencies for an R-repeated partitioned indirect binary n-tube are

$$C(Y) = R2^n(n+1),$$

$$L_{max}(Y) = Rn + n - 1,$$

and

$$L_{mean}(Y) = \frac{Rn}{Rn2^n - 1}(1 + 2^{n-1}(Rn + 2n - 3)).$$

(Shown in appendix 1.)

These are shown, together with the network bandwidth in Table 8.2. Figs. 8.5 and 8.6 show the mean latencies for the unidirectional ring, the bidirectional ring, the partitioned indirect binary n-cube and six partitioned indirect binary n-tubes, as a function of the number of units connected by the network.

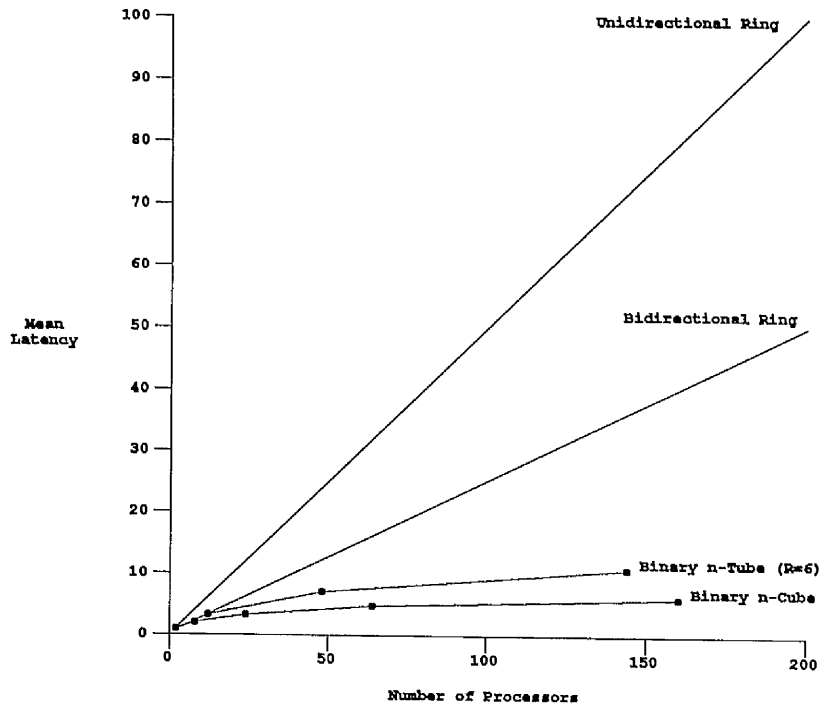


Fig. 8.5: Comparison of latencies for ring and partitioned indirect binary n-cube networks.

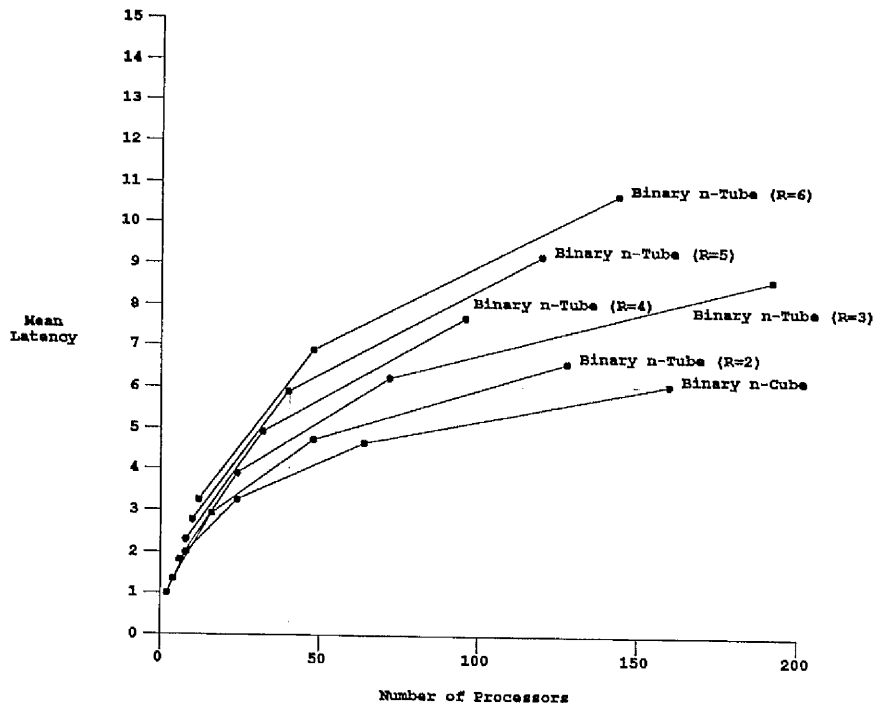


Fig. 8.6: Comparison of latencies for partitioned indirect binary n-tubes.

The mean latency of the partitioned indirect binary n-tube may be seen to be marginally higher than that of an equivalent sized partitioned indirect binary n-cube, but it is possible to construct many more different sizes of tube network.

The packet-routing algorithm for the partitioned indirect binary n-tube is similar to that for the partitioned indirect binary n-cube but, here, R opportunities to switch to any particular ring occur in any one circuit around the network. This gives the partitioned indirect binary n-tube an advantage over the partitioned indirect binary n-cube in that it may be shown that the network has a greater fault tolerance. The failure of any single connection or network control unit, provided suitable routing action is taken to avoid these, will not prevent the network from delivering packets to any other unit, since at least one alternative route always exists. Bidirectional n-tubes are also possible, but these are not investigated here.

8.3 Summary

A network to be known as the partitioned indirect binary n-tube has been introduced, and it has been shown that this may be a suitable interconnection structure for use in a shared-memory multiprocessor intended for image analysis applications. Several factors need to be investigated, however:

i) The network control units, briefly mentioned earlier, are critical parts of the scheme. It is necessary to show that these units could be implemented in an economical and efficient manner.

ii) In this form of machine, there may be some form of interaction between packets in the network. This may occur when two packets approaching a network control unit require to take the same route. Schemes for resolving such packet-routing clashes must be formulated and tested.

iii) When packet-routing clashes occur, one or other of the packets must take a route which is not an optimal route to its destination. This may (but will not necessarily) result in the packet being sent around the ring to approach its destination a second time. Whilst this does not affect the correctness of operation, it does increases the mean latency. The theoretical latencies quoted earlier are only correct under zero-traffic conditions. An investigation of non-zero traffic conditions is, therefore, necessary.

iv) From the theoretical results presented above, the partitioned indirect binary n-tube network appears to exhibit a relatively high latency, and this could seriously reduce machine performance. The possible effects of this latency must be investigated in detail, and methods of reducing the network latency must be examined. These methods fall into two categories; those which are intended to reduce the actual network latency, and those intended to avoid, wherever possible, the use of the network.

The structure of data packets, network routing algorithms and the implementation of network control units are discussed in chapter 9. An investigation of predicted network latency under non-zero traffic conditions, using a high-level simulator, is made in chapter 10, and a low-level simulator, designed for more detailed system simulation, is described in chapter 11. Details of low-level simulations and further results are presented in subsequent chapters.

Net. Anything reticulated or decussated at equal distances,
with interstices between the intersections.

'Dictionary of the English Language' — Samuel Johnson, LL.D.

Chapter 9:
Implementation Considerations for the Partitioned Indirect Binary n-Tube Machine

9.1 System Organisation

This chapter describes a possible implementation scheme for a shared-memory multiprocessor based on the packet-switched, partitioned indirect binary n-tube network. This machine comprises a number of processing units (PUs) and data memory units (DMUs), connected by a packet-switched network. The PUs and DMUs will be referred to collectively as functional units. Each functional unit is connected to a network control unit (NCU), which organises the routing of data packets along the network and the transfer of packets to and from the functional unit.

In the course of processing, the PUs generate memory requests in the form of request packets, and the NCUs organise the routing of these packets until they are delivered to the appropriate DMU. The DMU performs the required memory operation, and the result is returned by the NCU, in the same manner. A packet is returned for all types of access, not just for read operations. This is necessary to ensure correct program sequencing, as it is possible for packets to overtake one another in the network.

Each NCU is numbered, as shown in Fig. 9.1, and the functional unit associated with that NCU also takes this number. When depicted in this manner, the partitioned indirect binary n-tube topology may be seen to resemble 2^n interlinked rings, each composed of Rn units. The NCUs may also be said to form Rn 'ranks' of 2^n units. For any unit, the unit number may be seen to be a concatenation of the rank number and the ring number. The functional units, the packet format and the network control units are now described in more detail.

9.2 Processing Units

The PUs of the proposed machine are based on conventional microprocessors, since these are freely available at low cost, and no immediate advantage is to be gained by using a custom-built unit. A wide selection of processors are available, and most 16-bit microprocessors would be suitable. The properties

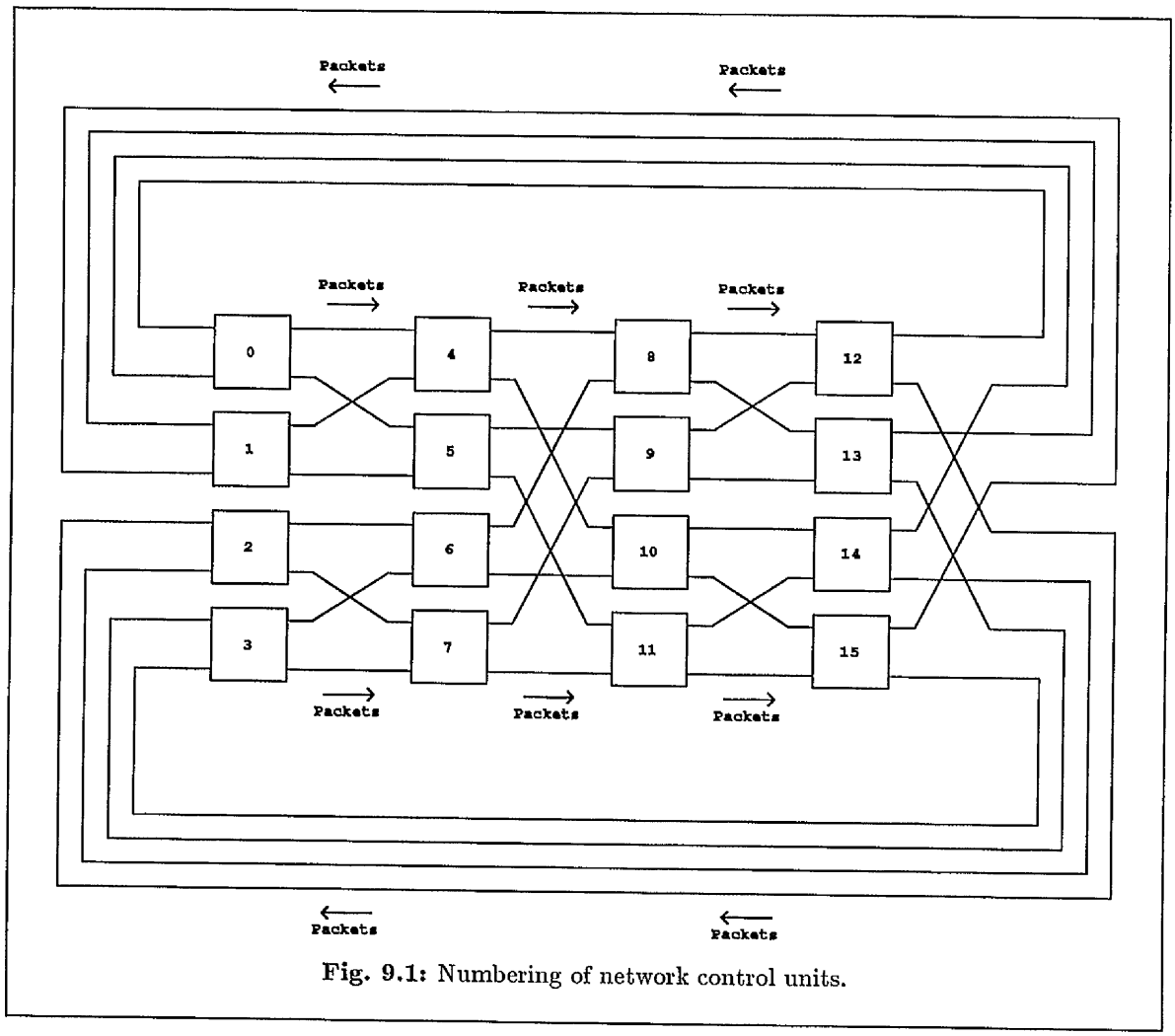


Fig. 9.1: Numbering of network control units.

required of the processor are high processing speed, a reasonable number of internal registers and a large addressing range.

The compiler and simulator systems described in later chapters are based on the Motorola MC68000 series processors [358, 359, 360, 361, 362]. The MC68000 series have an instruction rate of 1-2 million instructions per second, sixteen 32-bit registers (in addition to special-purpose registers) and a 32-bit addressing range, giving a maximum memory size of 4 Gigabytes. Instruction execution is overlapped with decoding of the next instruction, to increase processor throughput. Other microprocessors such as the Zilog Z8000 [363], the Intel iAPX 186 [364], the National Semiconductors NS32032 [365], or the Inmos TM424 transputer [301, 302, 304], were alternative choices.

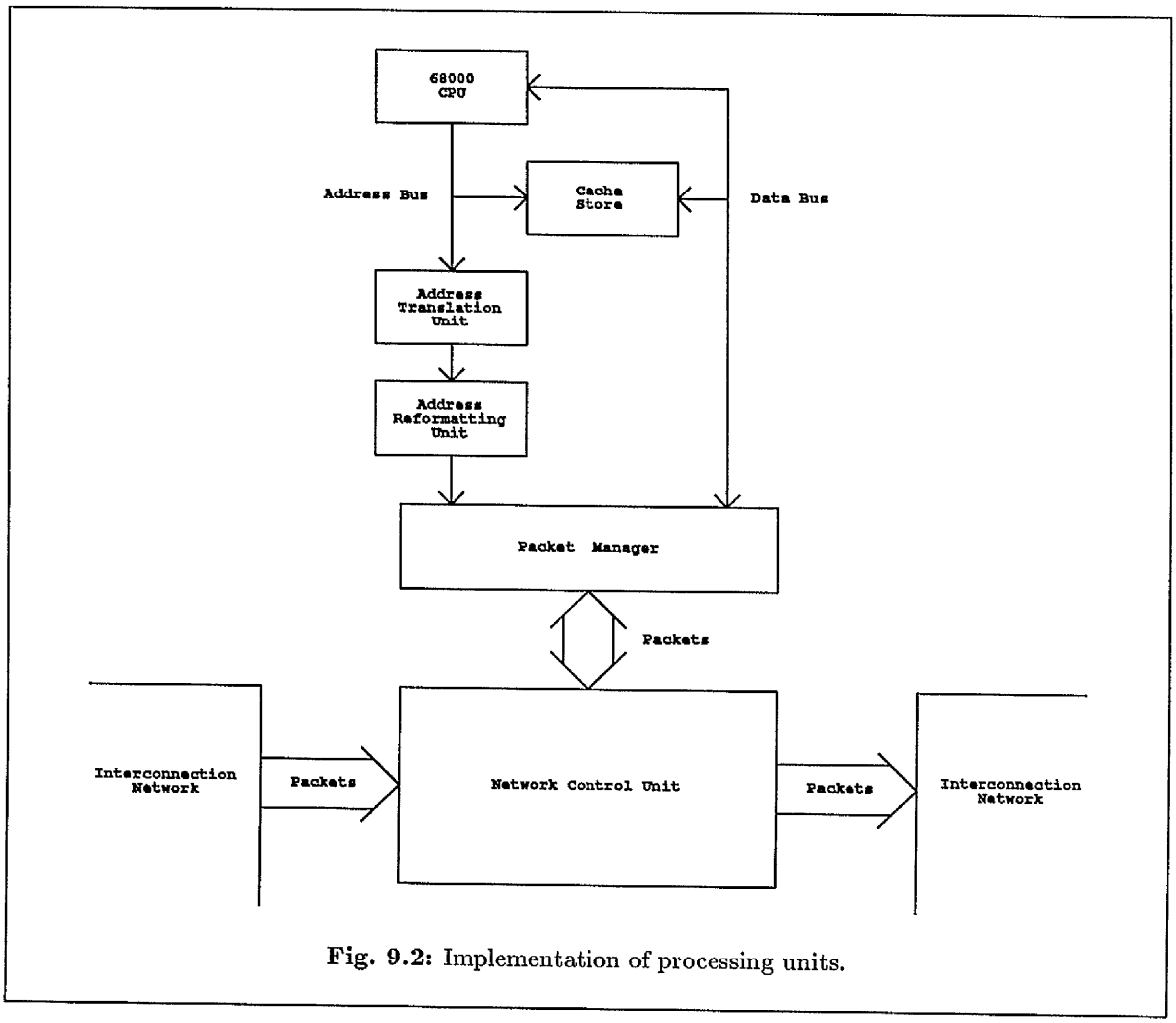


Fig. 9.2: Implementation of processing units.

In addition to the microprocessor, some other hardware is required to make up the processing unit, as shown in Fig. 9.2. In chapter 5, the DPU was defined to be primarily a processing unit, but it was stated that some form of cache store could be included in this. To reduce network traffic, a local program store could be provided. Similarly, some local private data storage could be useful in this respect. The effects of introducing these private cache stores, for instruction and local data storage, are described in chapter 14. Memory address translation, which is discussed below, must take place before the destination unit of the packet may be determined. To allow this, an address translation unit may also be incorporated into the processing unit. For reasons of efficiency of network transfers, also to be discussed later in this chapter, an address reformatting unit, which alters the format of unit addresses passed to the NCU, is also included in the processing unit. In addition to the sub-units described, the PU also requires a packet managing unit to construct and dismantle data packets.

9.3 Data Memory Units

Each DMU is a conventional random access store, with additional hardware to perform increment-in-memory (IIM) and decrement-in memory (DIM) operations. These operations are necessary for the efficient implementation of inter-process synchronisation, as described in chapter 12, and require a reply packet to be generated, yielding the modified value. These increment and decrement operations could be implemented using a simple binary counter, rather than a complete addition unit.

In the packet-switched network, data packets may be delayed in reaching their destination, due to interference from other packets in the network. Consequently, it is necessary to generate reply packets for normal write operations since, without this, a write operation may be delayed so that a subsequent read from the same memory location could yield the unmodified value. Read operations require a reply packet to be generated which contains the value read from memory.

To provide these functions, the DMU must contain an incrementer/decrementer and some additional timing circuitry, in addition to the hardware required to receive and transmit data packets.

9.3.1 Memory Map Organisation

The distribution of memory addresses across the available memory units must be considered. A straightforward memory mapping may be obtained by using the high-order bits of the address lines to determine the DMU unit number. This is not a good scheme, however, particularly for image analysis, since this would usually cause the entire of an image to reside in a single DMU. This would result in many accesses to this unit, with the inevitable results that memory requests would have to be deferred. A better scheme would be the one used in ATLAS [220], and later machines, to allow simultaneous multiple operand and instruction fetches. This system uses the low-order bits of the data address to determine the DMU number. Thus, images are distributed across all DMUs, permitting simultaneous access to different parts of the images. A problem occurs, however, when images are used which have a row length which is a multiple of the number of DMUs. In such cases, pixels which are in vertical alignment are stored in the same memory unit, and this may result in the occurrence of memory interference in certain forms of algorithm, particularly those which involve partitioning the image into horizontal strips to be processed.

To avoid this alignment problem, an address translation unit may be used to perform a pseudo-random hashing operation on the virtual address generated by the CPU, to yield a real memory address and, thus, a unit number. Such an address translation unit uses the k address bits as input to a pseudo-random function which, given a k-bit seed, generates a k-bit pseudo-random number. The function must be chosen carefully, so that all k-bit numbers generate unique results and, therefore, no two virtual addresses correspond to the same real location. All pseudo-random functions which generate k-bit numbers, and have a cycle length of 2^k are suitable, since every number in the range $0 \ldots 2^k - 1$ occurs in such sequences. A large number of pseudo-random sequences of the form

$$R_{n+1} = (C \times R_n + D) \bmod 2^k,$$

where C and D are constants, exhibit these properties, but the criteria for selection of suitable values for C and D are not obvious.

The result of using such a scheme is that contiguous virtual addresses occur at pseudo-random locations in the real store, and are therefore stored in pseudo-random DMUs. Patterns which may occur in accessing the virtual store will not then result in patterns in the real store accessing. The effectiveness of these schemes is discussed in chapters 14 and 15.

9.3.2 Memory-Accessing Clashes

In the normal course of processing, it is inevitable that, eventually, a memory unit will receive a request packet whilst it is involved in processing some other request. Two possible courses of action exist; one is to form a queue of requests, the other is to reject the new request, and force it to attempt the same access later. Although this latter option may appear to cause more latency delays, the network structure is such that any rejected packet, with no alterations, will take the simplest route back to the same destination, where it will re-try its request. This system effectively queues the memory requests in the network. Provided the rate of occurrence of memory-accessing clashes is low, this can be tolerated, and the expense of a special queueing unit in every memory unit may be avoided. The number of memory-accessing clashes which will occur in normal processing is difficult to predict, and can best be determined by simulation.

9.4 Network Packet Format

Each data packet passed to a network control unit from a processing unit must contain sufficient information to allow the NCUs to correctly route it to its destination, to define the memory request type and the memory location to be accessed, and also to allow the DMU to construct a reply packet. A suitable packet structure to do this contains:

i) the real global memory address concerned with this transfer (32 bits);

ii) the data to be transferred to, or from, the processing unit (16 bits);

iii) the memory access mode: read, write, increment-in-memory or decrement-in-memory (2 bits);

iv) the unit number of the source unit, which will become the destination number for the reply packet ($\log_2(Rn2^n)$ bits);

v) the unit number of the destination unit, to be used by the NCUs to route the data packet to its destination ($\log_2(Rn2^n)$ bits).

A number of schemes for network control unit implementation is now considered, and this simple packet structure is revised.

9.5 Network Control Units

Network control units perform two functions: data packet routing, and data packet insertion and removal from the interconnection network. Thus, the NCU may be divided into two parts, the exchange control sub-unit and the interface sub-unit (Fig. 9.3). The exchange control sub-unit performs the network routing, and the interface sub-unit handles the PU/DMU connection to the network.

9.5.1 Network Interface Sub-Units

The network interface sub-unit handles the injection of data packets into, and the extraction of packets from, the interconnection network. The exchange control sub-unit, described below, detects whether this unit is the destination of an incoming packet, and if so, the interface sub-unit must extract it from

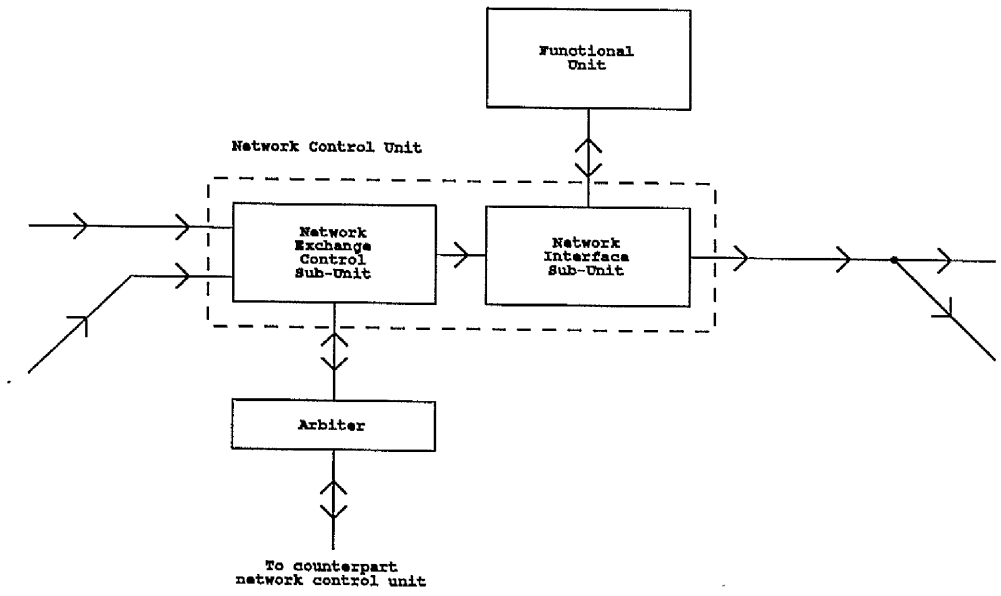


Fig. 9.3: The internal structure of network control units.

the network, and pass it to the packet manager of the attached functional unit. If the packet manager has an outgoing packet ready, and this network unit is free, then the interface sub-unit transfers this into the network.

It is important that this unit be capable of removing a packet from the network at the same time as placing an outgoing packet in the network, otherwise a deadlock situation could result if a heavily-used memory unit is unable to place its result packet in the network, due to new request packets circulating continuously past its NCU.

9.5.2 Network Exchange Control Sub-Units

The network exchange control sub-units route data packets through the network to their desired destination. As has been described, the machine may be considered to be a number interlinked rings which connect a number of 'ranks' of NCUs. This representation is useful in considering the routing requirements. Each exchange control sub-unit may allow the data packets to continue on their current ring or cause them to switch to another ring. In this, they are similar to the switch units employed in many other networks [366, 184, 183], but it should be noted that each NCU corresponds to only one half

of a switch unit. The exchange control sub-units must also determine when a packet has reached its destination node, so that the interface unit may be triggered to extract the packet.

The preferred routing algorithm for this unit has already been described in chapter 8. Packets move towards the ring on which their destination lies, and then circulate until the destination is reached. This may be implemented in a straightforward manner, but this could result in a slow hardware implementation if the simple packet format, described above, is used. Modified packet formats which allow more efficient NCU hardware to be used, are described later in this chapter. Before this detailed description of the exchange control sub-unit is given, the possibility of interaction between packets within the network is considered.

9.6 Packet-Routing Clashes

The packet-switched R-repeated partitioned indirect binary n-tube network is capable of holding $Rn2^n$ packets simultaneously. The maximum possible number of packets that may be generated at any one time is equal to the number of processing units in the system. There is a high probability, therefore, that only one packet will be present on the input side of any exchange unit, and this may be routed directly towards its destination, by this unit. In the case of two packets wishing to take the same route, a packet-routing clash occurs, and one of the packets must be sent away from its destination. This may result in this packet passing entirely around the network before it is able to reach its destination. In cases where a packet-routing clash occurs, information from the exchange control sub-unit must be combined with data from its counterpart on the exchanging ring, to determine the appropriate setting of this pair of NCUs (exchange or straight) for any particular case of input data packets. This is performed by the arbiter, the position of which is shown in Fig. 9.3.

A number of plausible strategies exist for arbitration of packet-routing clashes. It is important to consider each of these carefully, and to ensure that the selected strategy guarantees that all packets will eventually reach their destination. Such strategies are referred to here as 'correct strategies'.

i) Lowest/highest destination number takes priority

To allocate priority to the packet with the lowest, or highest, destination number guarantees that

the lowest-numbered (or highest-numbered) unit always has its packets routed directly to their destination, by an optimal route. At any given time, some packet has priority over all others, and must, therefore, reach its destination. The second lowest numbered packet then takes the place of the highest priority packet and, in this way, all packets will reach their destination. All processes must, therefore, terminate, and so this strategy is a correct one. It may not, however, be an optimal one, even though the 'worst case' described is an extremely unlikely set of occurrences.

ii) Straight connection for all clashes

This is, again, a correct strategy since, once on the correct ring, a packet takes priority over any other attempting to displace it. The only way in which a packet may be prevented from reaching its destination ring is by other packets which are themselves on their destination rings. These packets must eventually reach their destination units and, thus, all processes must eventually terminate.

iii) Exchange connection for all clashes

This is, perhaps surprisingly, not a correct strategy. It is possible, in this scheme, for two packets with destinations on the same ring, to enter an infinite loop in a partitioned indirect binary n-tube network. As an example, consider two packets, A and B, which clash just before packet A reaches its destination. The packets exchange, and A misses its destination. Just as packet B approaches its destination, the same two packets clash again and are again exchanged. Packet B now misses its destination, and the sequence repeats infinitely. This is only one example, and more complex repetitive sequences have been observed in simulations.

iv) Random, or pseudo-random allocation of priority

This cannot be a correct strategy, since there is no guarantee that the infinite sequence which occurred in case iii) cannot occur here.

v) Nearest to destination takes priority

The strategy of allowing the nearest packet to its destination to take priority, and randomly arbitrating equal priorities, may be shown to be correct, as follows. If some packet is deflected from its desired route, at some distance, d, from its destination, then the packet which deflected

it must be, in the worst case of equal priority, no further than $d-1$ units from its destination, after this exchange. If this packet is subsequently deflected, then the same argument applies, and some other packet will reach its destination in some lesser time. Thus, given any set of packets in the network, at least one must reach its destination, and so, eventually, all must reach their destination. This strategy would appear to be a sensible one, since a packet which is far from its destination has more opportunities to correct its course than one which is close to its destination. Using this scheme, packets are less likely to be deflected so that they miss their destination and, consequently, need to circumnavigate the network to reach it.

The last of these schemes would appear to be the most efficient, but this is best shown by simulation, the results of which are described in chapter 14. It is, however, necessary to show that this scheme may be implemented efficiently in hardware, since the performance of the network is thought to be an important factor in the overall machine performance.

9.6.1 Packet Formats for Improved Hardware Efficiency

The simple form of data packet, described earlier, includes, in each packet, the binary representation of its destination. Each NCU must compare this with its own unit number, which must be built into the NCU and, if the two are equal, the packet is extracted by the interface sub-unit. If this packet has some other destination, a routing calculation must be performed, arbitration with the counterpart on the exchanging ring must take place, and the packet may then be dispatched.

To implement the preferred packet-routing clash arbitration strategy, the distance of each packet's destination must be calculated, so that priority may be assigned to the packet with closer destination. For some NCU, whose number is i, the distance, l, to some other unit, j, is given by

$$l = (j - i) \bmod Rn,$$

where Rn is the number of ranks in the network. The arbiter structure shown in Fig. 9.4 evaluates l for each of the two packets in a packet-routing clash, and finds which is the smaller of the two. This is used to determine which of the two 'exchange request' signals is allowed to decide whether an exchange is to take place or not. However, since Rn is not necessarily a power of two, the required modulo Rn

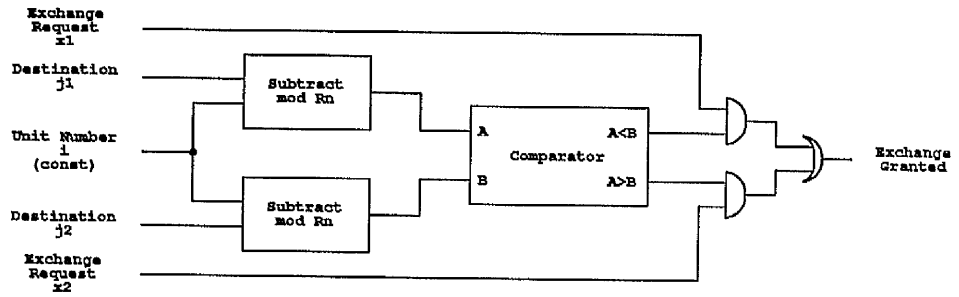


Fig. 9.4: An arbiter for simple packet structures.

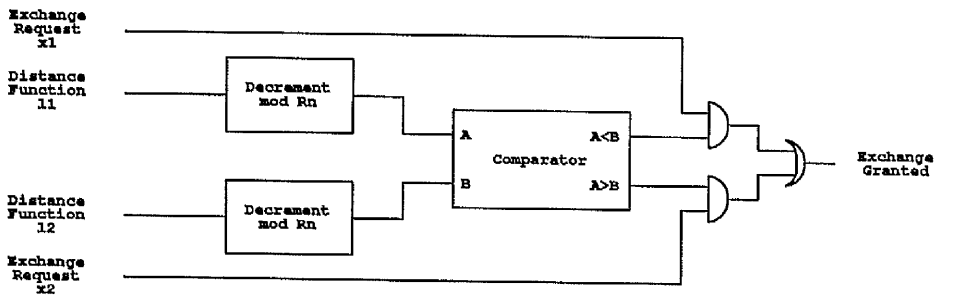


Fig. 9.5: An arbiter which uses incremental address calculation.

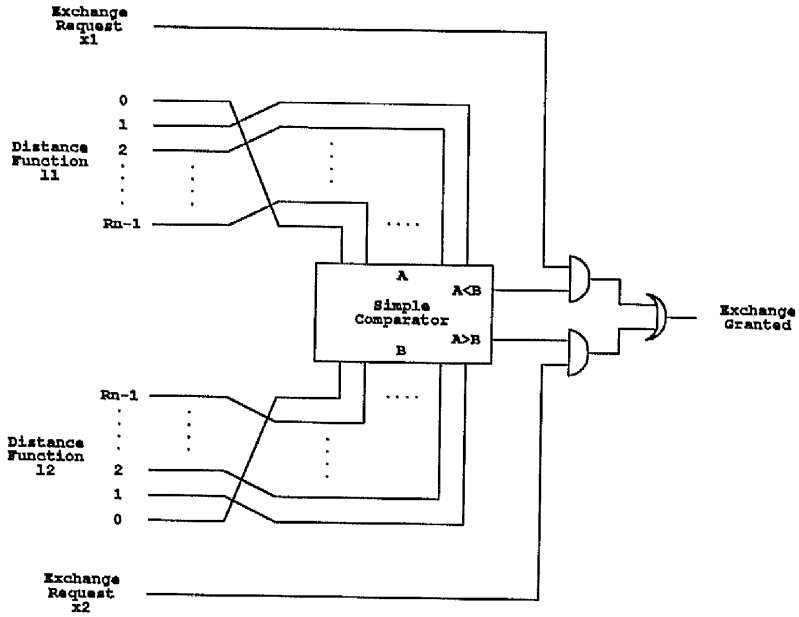


Fig. 9.6: An improved arbiter, with an Rn-bit numbering system.

subtraction is not straightforward, and may take an appreciable time. Since network speed is thought to be critical for efficient machine performance, this overhead is undesirable.

An alternative system, shown in Fig. 9.5, makes use of an address re-formatting unit, contained within the source unit. This calculates the value of l at the source unit and, at every network unit, this is decremented, modulo Rn. This value of l, when taken together with the destination ring number, contains sufficient information to uniquely define a destination address. The arrival of a packet at its destination may be detected when l reaches zero and, simultaneously, the ring number of the destination matches the ring number of the current NCU. Modulo Rn arithmetic must still be used to ensure correct calculation of destination distances if the packet is routed past its destination and must circulate the network to approach it again. The decrement operation is faster than a full subtraction, but modulo Rn arithmetic is still required, and may still result in an unacceptably slow network.

An improved scheme, shown in Fig. 9.6, uses an Rn-bit numbering system, where only one bit is asserted at any time, instead of the binary representation of the destination distance, l. As an example, destination distances of zero, one, two and three are represented in this notation by 0001, 0010, 0100 and 1000 respectively. Since the highest number to be represented is the number of ranks in the network, Rn, the number of wires required here is relatively small and, in any case, these numbers could be encoded for transmission between NCUs. Using this system, a decrement operation is reduced to a skewing of the wires to the right, and modulo Rn arithmetic is performed by wrapping the least significant line to the most significant place. Thus, the arbitration process is reduced to the detection of which of the two destination distance counters has the most significant asserted bit. This may be determined quickly, by simple combinatorial logic, and provides a fast method of arbitration. When the least significant (distance zero) bit is asserted, a packet may be seen to have reached the rank of units in which its destination lies. However, as the packet could be on any ring at this point, the current ring number must be checked against the destination ring number, to determine whether the final destination has been reached.

If the destination ring number of a packet is represented as a binary number, each unit must compare this with a stored version of its own ring number, to detect arrival of a packet at its destination. A comparison of one bit-position of the destination ring number is also required, to determine the

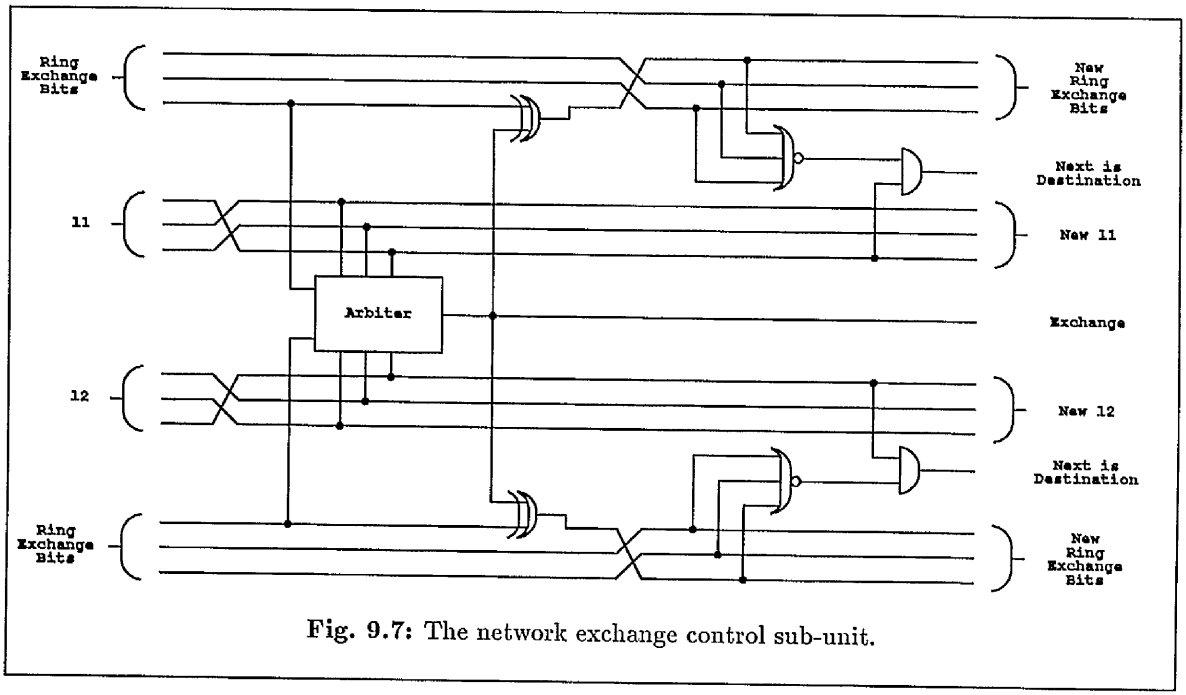


Fig. 9.7: The network exchange control sub-unit.

packet's optimal route. The selection of the bit to be checked depends on the exchange offered by the next stage of the network, and will differ between ranks of units.

A more efficient method of checking the ring numbers is to perform an exclusive-or operation on the source and destination ring numbers, at the source unit, to give an indication of all the ring-exchanges to be made by this packet. Whenever a packet moves between any two rings, the bit corresponding to the difference between the two is inverted. This is done whether the interchange was requested by this packet, or was due to diversion by some higher priority packet. In this way, the destination ring bits indicate, at any time, which ring interchanges remain to be made. Since the interchange of bits, in a regular partitioned indirect binary n-tube, is in a cyclic order of bit number, this code may be rotated at each rank and only the least significant bit tested. Fig. 9.7 shows a network exchange control sub-unit, for a system of eight rings, with three units on each, which uses this incremental method for ring-exchange checking.

When both incremental methods are used, it may be seen that the network control unit requires no representation of its own unit number to be built into it, since the arrival of a packet at its destination is indicated by a zero distance function, and all-zero ring-exchange bits. In this system, unit numbers

are only required during packet assembly in the address re-formatting unit and, elsewhere, all routing information is calculated on a relative basis. It would be possible to construct a system in which every functional unit contains an address re-formatting unit or, alternatively, the re-formatting unit in the processing unit could be used to calculate the appropriate relative return path, to be included in the outgoing packet. These incremental methods provide a fast and inexpensive way of implementing the exchange control sub-units.

9.7 Packet Hovering

In the network described, it is possible for packets to be deflected from their route by other packets with higher priority. The arbitration system for packet-routing clashes is selected to try to avoid deflecting packets such that they miss their destination, but this will inevitably, at some time, occur and these packets must travel around the network to reach their destination. An additional measure which may reduce this possibility is to allow packets to remain at their current network control unit, instead of proceeding on a non-optimal route, provided that no packet is waiting to move into the NCU which this packet currently occupies. After the higher-priority packet has proceeded, this packet is free to resume its optimal route. Such a system may improve the latency figures for the network, but would require careful design to avoid the possibility of lock-up, where parts of the network could become clogged with stationary packets.

9.8 Fault-Tolerance of the Network

In chapter 8, the partitioned indirect binary n-tube network was stated to exhibit a certain amount of fault tolerance, and the practical exploitation of this potential is now considered. Four different failure modes exist:

i) Processing unit failure

The failure of any processing unit may be overcome by isolating the faulty unit, altering the software to compensate for its absence, and restarting the machine. This may require re-compilation of the application program but, with a suitable system software structure, it is possible that this could be avoided.

ii) Memory unit failure

A memory unit failure requires similar action to a processor failure, and might also require re-compilation. In addition, it may be necessary to switch off any address translation mechanism which is in use, otherwise the missing memory locations would be distributed throughout the shared store.

iii) Network control unit failure

The failure of a network control unit could be dealt with by overriding the network-routing hardware in the two preceding NCUs, so that incoming packets are always routed away from the failed NCU. Once these packets arrive at an undamaged section of the network, the normal routing algorithm will cause them to be routed to their destination, exactly as if they were deflected by a routing clash with a high-priority packet. It may be necessary to implement a packet hovering scheme to ensure correct operation in such cases, since such failures could cause 'bottlenecks' in the network.

iv) Interconnection network failure

The failure of any connection between two network control units could be handled in a similar manner to the failure of a network control unit. In this case, however, only one network control unit must be set to deflect packets around the damaged link.

To implement these schemes, the only necessary modification to the existing hardware design is the provision of an override setting for the network-routing mechanism. This requires only minor modifications to the network control units, and the operating speed of the NCU should not be affected. A fault-tolerant system may, thus, be provided.

9.9 Summary

The implementation of a multiprocessor machine, based on the partitioned indirect binary n-tube, has been discussed, and the basic forms, and some variants, of the processing units and memory units have been described. The structure of network data packets for such a system, and their possible interaction, has been examined, and a number of methods of resolving network routing conflicts have been discussed,

one of which has been selected for further investigation. An implementation scheme for the network control units has been described in detail, and it has been shown that high-speed, fault-tolerant units may be economically constructed. The following chapters describe the simulation of this system, and its variants, at two levels of detail.

And now I see with eye serene
The very pulse of the machine;

'She Was a Phantom of Delight'
— William Wordsworth

Chapter 10:

Multiprocessor System Simulation

10.1 Levels of Simulation

The need for simulation of the partitioned indirect binary n-tube system has been shown in the preceding chapters, and two simulators developed for this purpose are described in this chapter and the following chapter. It is possible to simulate systems at many different levels:

i) Circuit-level simulation

This is the simulation of individual electronic components and their interaction. The detailed calculation involved in this limits the use of these systems to checking the internal structure of individual logic gates, and the prediction of such parameters as gate propogation delay and fan-out.

ii) Gate-level simulation

Here, the basic units of simulation are simple logical devices, such as gates or bistables. The model ignores internal detail, and includes only timing parameters such as propogation delays, set-up times, and hold times. Such simulators may also check user-imposed circuit design rules, but their use is usually limited to the modelling of small systems.

iii) Register-level simulation

This is the simulation of a machine, by the modelling of register contents and their transfers, without reference to propogation delays, or other physical parameters. Such systems may be used to simulate entire machines.

iv) Unit-level simulation

Here, entire functional units are treated as 'black boxes'. The model used to simulate the internal functions of these units is not related to the actual implementation of these units, and so may not perform in exactly the same manner.

It may be seen that, although detailed low-level simulation may be desirable, since its results are more accurate than high-level simulators, the large amount of calculation involved in low-level simulation may

be impractical. The simulation of the partitioned indirect binary n-tube system has been performed at only two levels, the unit level and the register level, since circuit-level engineering considerations are not of primary importance in this study.

10.1.1 A High-Level Machine Simulator

A unit-level simulator was designed to model the network interactions in a partitioned indirect binary n-tube system, and to measure the resulting latency in memory accesses for differing topologies, routing strategies and network traffic situations. The operation of this program is now described.

To calculate the mean latency for any network, given some activity level expressed as the proportion of occupied network control units, the probable latency between all possible pairs of units must be averaged. The computation involved in this may be reduced by making use of the symmetry of the network, and calculating only the mean latency from one fixed unit to all others. To evaluate the probable latency for some packet to travel from one unit to another, the position of the packet is modelled as a probability distribution across the network. The probability of the packet arriving at any particular network control unit, at any particular time, depends only on:

i) the probability of this packet having reached one of the two preceding units in the previous time interval;

ii) the routing which these units would then select;

iii) the probability of the occurrence, at this point, of a packet-routing clash in which the monitored packet would be diverted from its optimal route.

This third probability, in turn, depends on the level of network traffic, and is assumed here to be half of the probability that another packet will be present; that is, there is an equal probability that each of the packets will 'win' the conflict.

10.1.2 Results of High-Level Simulation

The results of the high-level simulation of partitioned indirect binary n-cube and n-tube systems, for several different sizes of network, are shown in Figs. 10.1 to 10.4. It may be seen that the mean latency does increase substantially with increasing network occupancy, particularly for wider networks, where more packet interaction is possible. Very large increases in latency occur only at occupancy levels of above 50%, and it should be remembered that a limit on network occupancy is imposed by the ratio of processing units to total number of units. Provided the network traffic level remains below a value of about 10% to 20% of total network capacity, depending on the parameters of the network selected, it appears that reasonable latencies may be obtained.

10.2 The Need for Low-Level Simulation

The model used to produce these results, however, makes certain assumptions which cannot be assumed to be valid without further investigation:

i) The network traffic distribution is assumed to be even, whereas it is, in practice, not necessarily so. The interaction of processes which access shared variables may result in a concentration of packets directed to a small number of memory units.

ii) The probability of a packet 'winning' at a packet-routing clash is assumed to be 0.5 at any point, but this may not be the case.

iii) The network traffic in the system is assumed, here, to be constant with time. In a real system, this is not likely to be so, and the number of packets in the network may vary considerably in different phases of a program. It is difficult to estimate a true mean level of network activity.

For these reasons, and in order to fully investigate the effects of different parallel algorithms on the memory access patterns and hence the machine performance, it is necessary to model the entire system at a lower level. A simulator program written to do this is described in the next chapter.

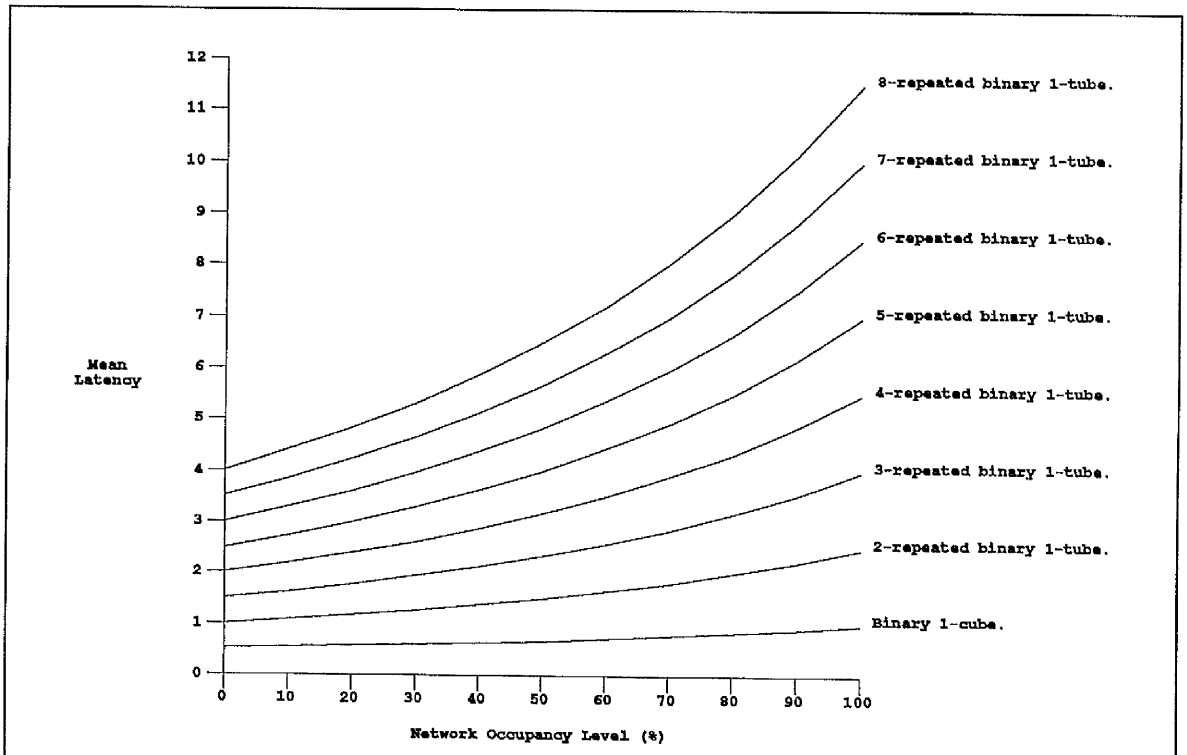


Fig. 10.1: Mean latencies for binary 1-tubes.

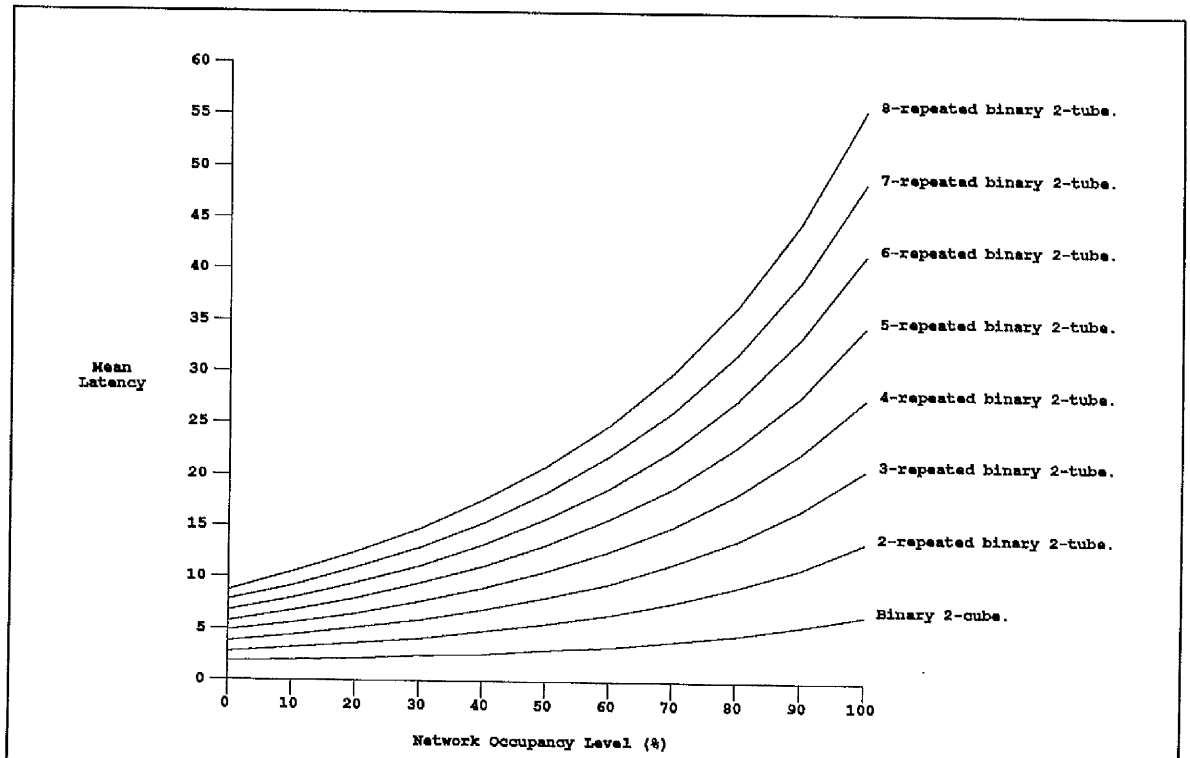


Fig. 10.2: Mean latencies for binary 2-tubes.

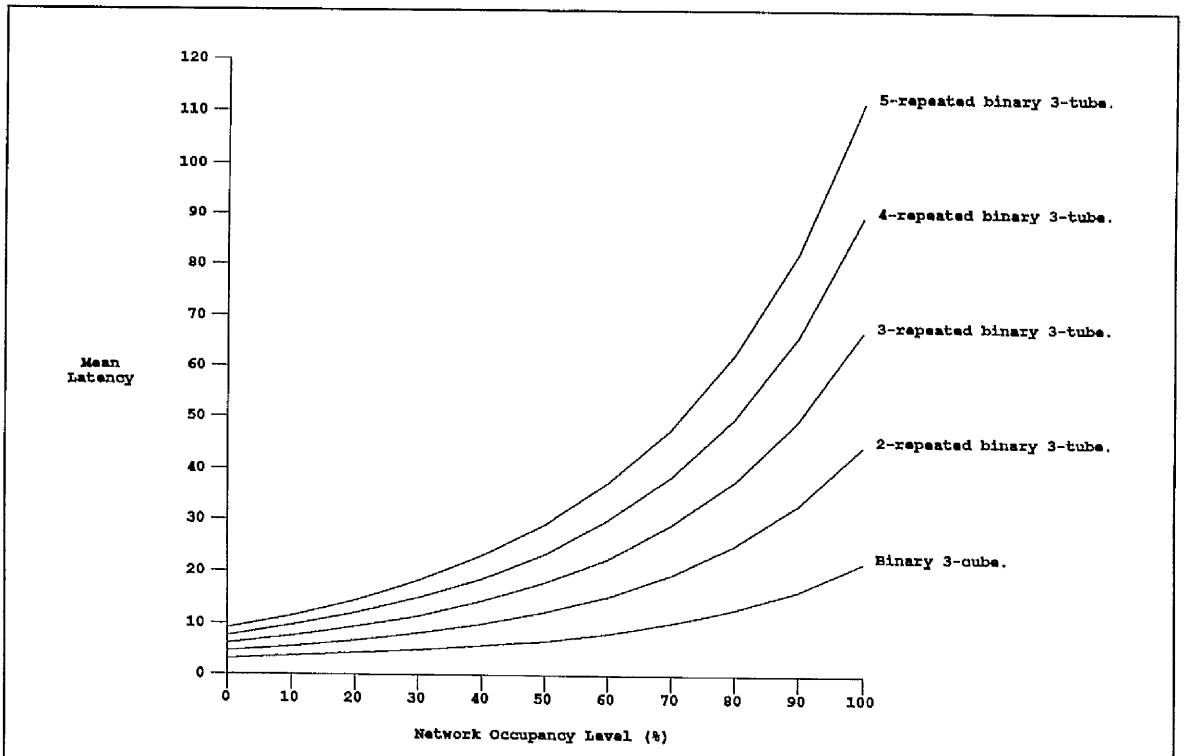


Fig. 10.3: Mean latencies for binary 3-tubes.

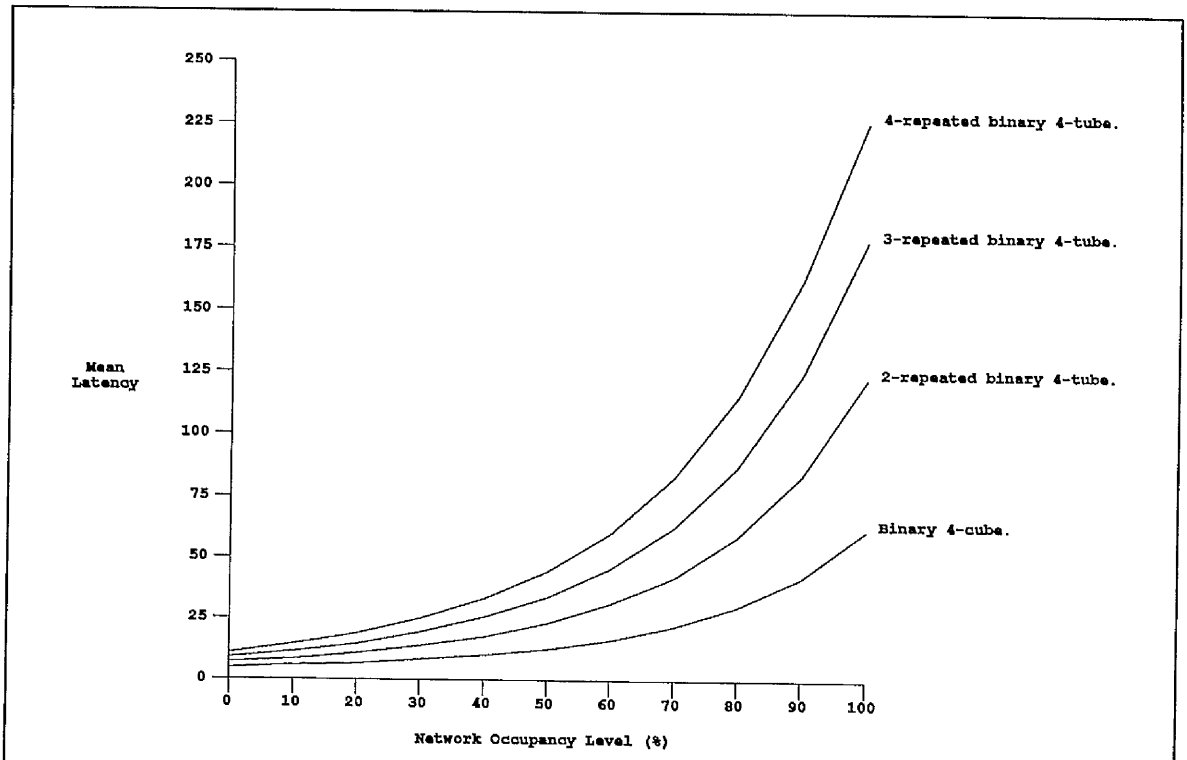


Fig. 10.4: Mean latencies for binary 4-tubes.

"An envious fever
Of pale and bloodless emulation."

Ulysses, in 'Troilus and Cressida'
— William Shakespeare

Chapter 11:

A Low-Level Simulator

11.1 The Structure of the Low-Level Machine Simulator

This chapter describes a program which was developed to perform low-level simulation of the shared-memory multiprocessor system described in chapters 8 and 9. In this simulator, the interconnection network and the internal functions of the processors are modelled at a register level, and the distributed memory is simulated in full.

The machine simulator is capable of modelling up to 128 network control units, with a maximum of 64 attached processing units, interconnected by one of a number of packet-switched networks. The currently implemented program supports the unidirectional ring network, the partitioned indirect binary n-cube, and the partitioned indirect binary n-tube. A distributed global memory of up to 32K 16-bit words is simulated. This size limit is imposed by the 16-bit integer range of the Magiscan 2, which makes larger arrays difficult to address. Although a 32K-word store is much smaller than any practical multiprocessor machine, it is sufficient for simulation since the constraints on simulator execution time mean that programs which use a larger memory space would require an impractically long simulation time.

The simulator is regulated by a single clock, which counts in network cycles. One network cycle is the time required for a data packet to pass through one NCU, and this is maximum temporal resolution of the system. All other temporal parameters are converted to integer multiples of network cycles, since all memory and processor units must remain in close synchronisation with the network. The simulator operates by repeatedly advancing the system clock by one network cycle at a time, and for each cycle, the following actions are performed:

i) Network management

Each data packet in the network is moved forward to the next network control unit, according to the topology of the network and the selected routing algorithm.

ii) Instruction execution

The model of the internal state of each functional unit is advanced by one network cycle.

iii) Result logging

Any events which have occurred in this network cycle, such as the arrival of a data packet at its destination or the decoding of a new instruction, are logged.

iv) User interface handling

The keyboard is scanned, and any commands issued by the user are executed.

Each of these actions is examined in detail in the remainder of this chapter.

11.2 Network Management

The data packets transmitted by the network are based on those described in chapter 9, and comprise a source unit number, destination unit number, a memory address, a data word, and access mode information. The incremental routing calculation methods described in chapter 9 are not simulated, as these are merely mechanisms to increase the speed of the network control units and do not affect their function. To simulate the actions of the network for one network cycle, the following steps are performed:

i) Any packet which has reached its destination is removed from the network, and passed to the appropriate unit handling routine.

ii) Any packet which is waiting to move from a packet manager onto the network is moved, if the appropriate network control unit is free.

iii) Any packet-routing clashes are resolved, using one of the algorithms discussed in chapter 9, and the immediate destination of all packets is determined.

iv) Each packet in the network is moved to the next network control unit, in accordance with the route selected in step iii).

The network topology is determined by the user, before simulation commences.

11.3 Instruction Execution

At the commencement of each new instruction, an instruction number is read from the memory location pointed to by the processor's program counter. This is used as an index into the program list, to obtain the new instruction. This indirect system is used because instructions are not packed into the compact form used by the MC68000. Instead, the individual fields of the instruction are stored in a record structure, to avoid the necessity of first packing them, and later unpacking them.

The instruction is decoded into a set of primitive operations, known as microcycles. The number of network cycles that are required in order to execute each microcycle is determined at this time. A large instruction set, with many addressing modes, may easily be modelled by relatively few microcycles. The most complex MC68000 instruction implemented in the simulator requires only eight microcycles. This model of the MC68000 is not strictly accurate; in the MC68000, the execution time for operations such as multiply or divide depends on the data involved, whereas, here, these operations take a constant time. Since these instructions are relatively infrequent, the use of an average value was felt to be appropriate. Each processor maintains a list of outstanding microcycles, and a new instruction fetch is performed when this list becomes empty. Each microcycle takes one of three forms; mill, memory request or execute:

i) Mill (MILL N)

This form of microcycle has one parameter, N, and causes a processor to pause for N network cycles. This may be used to model a local memory access, or the computation time involved in performing an instruction.

ii) Memory request (MREQ M, R)

This type of microcycle, when executed, causes an memory access packet to be generated, of request type R, to act on memory location M. The memory address is calculated when the instruction is decoded, taking into account any register changes which are to be made by this instruction.

iii) Execute (XUTE)

This causes an instruction to be performed on the registers of this processor. This form of

microcycle is executed in zero network cycles, and the execution time of the instruction is modelled by a separately-generated mill microcycle.

As an example, the two-word instruction

```
282 SUB (32000), D1
283     32000
```

where 282 is the address of the instruction, in a non-local program store, might be decoded as

```
MREQ    282,Rd   ; Instruction fetch
MREQ    283,Rd   ; Operand address fetch
MREQ    32000,Rd; Operand fetch
MILL    2        ; Instruction execution
XUTE             ; Alter register values
```

No time is required to account for instruction decoding, since the MC68000 overlaps instruction execution with the decoding of the next instruction. This does mean that the exact sequence of operations in the simulator differs from those which would be observed in an actual MC68000, but this difference was felt to be minor.

11.4 User Interface

The user interface allows the configuration of the simulated machine to be altered, and the actions of the simulated program to be interactively observed in detail, whilst simulation takes place.

11.4.1 The Display

When the simulator program is running, the Magiscan screen is divided into three sections, as shown in Plates 11.1 to 11.3. The upper part of the screen displays the current state of the simulated machine, and shows the current simulated time, the source and destination of any packets in the network, the state of each processor, their program counters, the percentage of elapsed time spent executing each instruction class (defined in chapter 12), and the hardware efficiencies (defined later in this chapter) for each instruction class. The overall machine efficiency, memory utilisation, network utilisation, and average distribution and efficiency for each instruction class, are also displayed here. Slightly below the

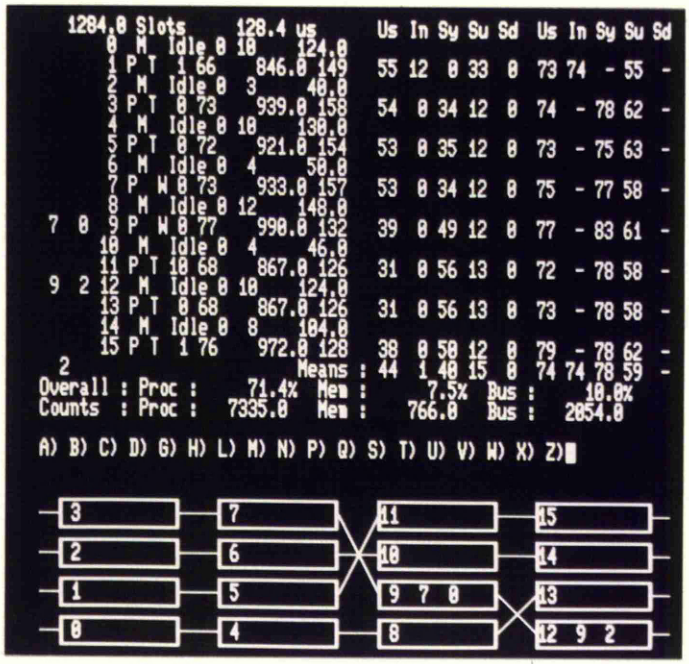


Plate 11.1: The low-level simulator, in display mode 1.

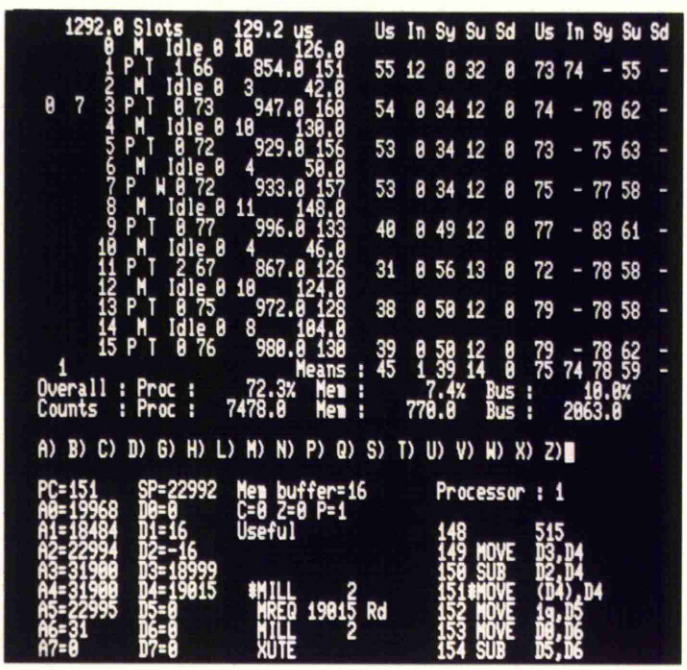


Plate 11.2: The low-level simulator, in display mode 2.

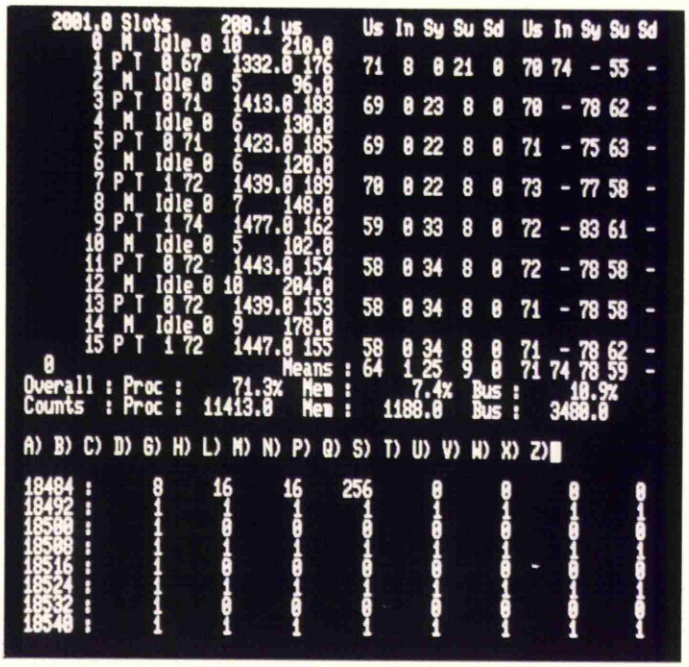


Plate 11.3: The low-level simulator, in display mode 3.

centre of the screen is the command line, on which is displayed a list of currently available options, or a suitable prompt for numeric or textual input.

The lower part of the screen may operate in one of four modes. In mode 0, no information is displayed here, to increase display speed. Mode 1 (Plate 11.1) gives a graphic representation of the simulated machine structure, showing the network topology, the sources and destinations of any packets present, and the last routing decisions made. Mode 2 (Plate 11.2) shows the internal state of a selected processor, and displays the processor's registers, flags, memory buffer, all microcycles of the current instruction, and a scrolling window into a mnemonically disassembled program listing, with the current instruction indicated. Mode 3 (Plate 11.2) provides a 64-word memory window which may start from any location, set by a separate option.

During processing, the screen is not updated on every network cycle, as this would be very time-consuming. On every 100th cycle after starting any operation, the time display is updated and, at an alterable interval (for which the default is 1000 network cycles), the whole screen is updated. In addition, the display is updated at the end of every user command, or at a user interrupt.

11.4.2 Organisational Commands

A wide range of user commands are provided, using a multiple menu system. Commands may be typed during simulation, in which case the current activity is halted, and the new command is executed. The available options include commands to allow the user to:

i) examine any part of the logged results, described below;

ii) display or modify parameters of the simulated machine;

iii) list the program in a mnemonic disassembler notation;

iv) display the contents of all packets on the network;

v) list or alter any section of the simulated memory;

vi) display or change any simulated machine register;

vii) execute any number of network cycles.

A particularly useful command continues execution until the currently selected processor moves to a new instruction. In conjunction with display modes 1 and 2, this provides a convenient method to step

through a program, viewing the resulting register or memory changes. Other commands are available which are concerned with program and result file handling, and simulator de-bugging.

11.4.3 Simulated Machine Options

The low-level simulator is capable of modelling a large number of different variants of the basic machine structure described in the preceding chapters. Any of the following parameters of the machine may be manually altered by the user, or by loading a system configuration file, which contains a full set of parameters to define a machine for simulation.

i) Network topology

The choice of network is limited, in the current implementation, to the simple unidirectional ring, the partitioned indirect binary n-cube, or the partitioned binary n-tube.

ii) Number of network control units

This defines the total number of network control units in the system.

iii) Number of rings

This parameter is only relevant to the partitioned binary n-tube topology. It defines the number of rings in the machine, and, together with the number of NCUs, determines the length of each ring.

iv) Number of processors

The number of processing units, and the positions of these in the network, must be specified.

v) Number of memory units

The number of memory units, and their positions in the network, is also be specified. The total number of processors and memory units may be less than the number of network control units, as some of these may have no attached functional unit.

vi) Memory board size

The size of each memory unit may be altered and this, together with the number of memory units, defines the total size of the global memory.

vii) Network cycle time

This defines the time, measured in nanoseconds, required for each network cycle.

viii) Processor cycle time

This defines the cycle time of the processor clock, in nanoseconds. From this, the time to execute the internal parts of each instruction may be calculated and converted into multiples of the network cycle time.

ix) Mirror read time

This is the time taken for a processor to read a unique identification number, which must be provided on each processor board in order that each processor may execute only the part of the program which is appropriate to it. This operation is not performed often, and so this is not a critical parameter.

x) Memory access time

This is the time, measured in nanoseconds, taken by a memory unit to perform a memory read or write operation, and generate a return packet. This time is rounded up to a multiple of the network cycle time for simulation, to allow for synchronisation with the network control units.

xi) Memory operation time

This is the time taken to perform an increment-in-memory or decrement-in-memory operation in a memory unit. This is at least twice as long as the memory access time, since both a read and a write operation must take place internally.

xii) Memory address translation system

This determines which scheme is used for mapping the virtual addresses, generated by the processors, into real memory addresses and network unit numbers. Two schemes are implemented: a direct mapping, with consecutive addresses in the same memory unit, and an interleaved mapping, with consecutive addresses in adjacent memory units.

xiii) Packet-routing clash arbitration system

Four of the five arbitration methods described in chapter 9 are implemented as alternative packet-routing clash arbitration strategies. These are the 'always straight' strategy, the 'always exchange'

strategy, the 'pseudo-random' strategy, implemented here as a simple alternating system, and the 'nearest to destination' strategy.

xiv) Packet hovering

This is a boolean variable which determines whether a packet is allowed to wait at a network unit until its route is clear, provided no other packet is about to move to its present location.

xv) Local instruction store

This boolean variable defines whether processors have a local store for instruction storage. It is assumed that, if present, this is pre-loaded with a copy of the program.

These parameters are set before simulation commences, and remain unaltered for the duration of the simulation. System configuration files, which contain a full set of these parameters, are also used by the MIMD-Pascal compiler (described in the next chapter), which places the name of the configuration file used during compilation into the output code file. This is detected by the simulator, when a code file is loaded, and the appropriate configuration file is also loaded, although this may be manually overridden by the operator.

11.4.4 Batch Mode

The simulator may operate in a batch mode, to allow it to be used unattended for long periods, in particular, for overnight runs. Before batch mode may be selected, the user must specify a list of files to execute, which configuration files to use, and names of result files to be generated. These may be entered either by typing them from the keyboard, or by reading from some specified file. Whilst in this mode, the display is only updated at long intervals, but pressing any key causes it to be updated immediately. Batch mode may be interrupted, at any time, to allow the user to examine the simulated machine state, after which batch mode may be resumed.

11.5 Result logging

Information is collected throughout the execution of the program and is deposited in a disc file at the end of the run. The information which is gathered comprises :

i) Total simulated run time

This is the simulated execution time, measured from the commencement of the first instruction, to the shut-down of the last processor, and is expressed both in network cycles and in microseconds.

ii) Time used for each instruction class

Each instruction is tagged with a marker to indicate the part of the program with which it is concerned. Five classes of instruction are defined in chapter 12: system start-up; system shut-down; system synchronisation; user initialisation; and user program. The percentage of processing time spent executing instructions of each class is recorded.

iii) Hardware efficiency

A figure known as the hardware efficiency is calculated for each of these instruction classes. For any sequence of n instructions, with elapsed processing times $T_1, \ldots, T_n$, and total memory latencies $L_1, \ldots, L_n$, the hardware efficiency, E, is given by

$$E = \frac{\sum_{i=1}^{n} T_i - \sum_{i=1}^{n} L_i}{\sum_{i=1}^{n} T_i}.$$

This is measure of the performance of the hardware, for some given number of processors under the specified conditions, and is largely independent of the efficiency of the software in completing the task involved.

iv) Time distribution between checkpoints

Named pairs of checkpoints, one designated 'enter' and one designated 'exit' may be inserted in the source code of the simulated program, to mark some region of interest. A timer is associated with each pair of checkpoints and, whenever the instruction corresponding to the 'enter' checkpoint is decoded, the timer is started, and it is stopped at the 'exit' checkpoint. This provides a

mechanism to monitor the distribution of processor activity across the program. A separate set of timers is maintained for each processor, and these are used to calculate activity distributions for each individual processor, and the mean activity distribution taken over all processors. Pairs of checkpoints are automatically inserted, by the MIMD-Pascal compiler, at the beginning and end of each program block or procedure.

v) Hardware efficiency between checkpoints

The total network latency which occurs between each pair of checkpoints is recorded, and this may be used to calculate the mean hardware efficiency for instructions in this section of the program. This is calculated for each individual processor, and a mean figure for all processors is also produced.

vi) Number of memory accesses

The number of packet transfers originating from each functional unit is recorded. This may be used as a measure of network activity generated by a processing unit, or as a measure of the amount of use which a memory unit receives.

vii) Mean latency

The mean latency for all packet transfers from each functional unit is recorded, and an overall mean is calculated.

viii) Number of packet-routing clashes

The total number of packet-routing clashes which occur during simulation is recorded.

ix) Number of memory-accessing clashes

The total number of memory-accessing clashes which occur during simulation is also recorded.

x) Shut-down time for each processor

The time at which each processor executes its final 'stop' instruction is recorded, as not all processors will finish execution simultaneously. The mean shut-down time, and the standard deviation are calculated from these times.

The result file is also tagged with all of the the simulator and compiler options which were active for this particular program run.

11.6 Summary

A flexible low-level simulator program has been described, which models ring-based, shared-memory multiprocessors. The following chapters describe a compiler which generates suitable code for this simulator, a number of test programs, and the results of simulation of these programs.

"Whereas, by exciting emulation
and comparisons of superiority,
you lay the foundation of lasting mischief;"

Attributed to Samuel Johnson, *in*
'The Life of Samuel Johnson, LL.D.'
— James Boswell

1

Parallel Computer Architectures and Algorithms for Medical Image Analysis

by

Granville Vincent Moore, B.Sc..

Volume 2

A thesis submitted to
the Victoria University of Manchester,
for the degree of Doctor of Philosophy
in the Faculty of Medicine.

Department of Medical Biophysics,
Faculty of Medicine,
University of Manchester.

October, 1987.

Abstract

Computer manipulation of digitised images is a rapidly expanding field of computer science, which has many biomedical applications. This project examines some medical applications of computerised image analysis, and studies, in detail, two applications in clinical ophthalmology. The first of these is the automatic evaluation of optic disc cup volume, for early diagnosis of glaucoma. A partially implemented model-based, iterative approach is described for this. The second ophthalmic application is the automatic detection and measurement of corneal endothelial cells, and a program is described which performs this task. From the measured performance of these programs, a need for faster image processing hardware is apparent, and a number of machine structures are studied for this purpose. To aid comparison of machines, a machine taxonomy is developed. This includes a matrix-based representation scheme for interconnection networks, from which quantitative characteristics of networks are derived. A parallel machine structure, suitable for image analysis applications, is proposed. Two machine simulators which operate at a unit level and register level are described, and simulation results are presented. A high-level language, developed for use with the low-level simulator, is described, and the implementation of a compiler for this language is also described.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this, or any other, university or institution of learning.

Acknowledgements

The author would like to thank all those who assisted in the completion of the work described in this thesis, and in the production of the thesis itself. In particular:

All of the members of the Wolfson Image Analysis Unit and the staff of Visual Machines Limited; in particular, Dr. R. N. Dixon, Dr. J. Graham and Mr. D. Pycock, for their advice on image analysis problems and assistance in de-bugging programs on the Magiscan 2;

Mr. A. Ridgeway, of the Manchester Royal Eye Hospital, for assistance in relation to the ophthalmic aspects of the project;

The Department of Computer Science, at the University of Manchester, for the use of the MU6 research computer, to provide some of the results presented in the thesis, and for the use of typesetting facilities to produce the thesis;

Mr. M. I. Wolczko of the Department of Computer Science for assistance in typesetting the thesis.

The author would also like to thank his family and friends, for the support without which this project could not have been completed. Finally, the author is greatly indebted to the project supervisor, Dr. C. J. Taylor, for his invaluable help throughout the project, in the writing of the thesis, and, in particular, for reading the manuscript.

Look on my works, ye Mighty,
and despair!

'Ozymandias'
— Percy Bysshe Shelley.

Contents: Volume 1

Contents: Volume 2

List of Illustrations

Chapter 13
Test Programs for Simulation

Chapter 14
Results of Low-Level Simulation

List of Plates

Chapter 11
A Low-Level Simulator

Chapter 12:

A Compiler for Parallel Machine Simulation

12.1 Choice of Programming Language

To simulate the execution of any task, the low-level simulator described in the previous chapter requires a suitable set of machine-code programs; one for each processor in the machine. These programs must, in the general case, interact closely and it is desirable that all should be generated from a single source program. Four classes of programming language were considered for use with the low-level simulator:

i) assembly languages;

ii) functional languages;

iii) automatically partitioned imperative languages;

iv) manually partitioned imperative languages.

In this chapter, each of these classes is examined, and a compiler, developed for use with the low-level simulator, is described.

12.1.1 Assembly Language

The use of assembly language for multi-processor programming has the advantage of simplicity in the system implementation, but the programming task involved in writing applications programs, even for simple test purposes, is much greater than if a high-level language were employed. It is possible, using an assembly language, to make efficient use of parallelism by maintaining closely-controlled synchronisation between processors, and so good machine performance may be obtained. However, a time-consuming process, such as writing assembly code, could not be regularly used, in the long term, for a production system. Consequently, simulations using hand-coded assembler programs would be likely to give misleading results. For this reason, the use of a simple assembler was rejected.

12.1.2 Functional Languages

Functional languages are a class of languages, including applicative and single-assignment languages [368], which are based on the application of functions to arguments. They are history insensitive, that is, no state changes occur during function evaluation and, therefore, any call of a function with the same arguments always yields the same result. Pure functional languages [369, 370, 371, 372] and reduction languages [373] are based on the lambda calculus [374], and do not involve the use of named variables for data storage. These languages are well suited for formal program correctness proofs [370, 375]. Parallelism may be easily exploited by parallel evaluation of the arguments of functions, although some programming styles which involve the manipulation of potentially infinite lists make this hazardous [377]. The implementation of functional languages on conventional machine architectures is rather inefficient [377, 340, 341, 368], but they appear to be suitable for use with data flow architectures [378, 379, 357, 380, 381], since they are easily translated into the graph structures frequently used to program this type of machine [382, 343].

Functional languages suffer from a major problem concerned with large data structures. In a conventional language, these may be easily and efficiently updated, by changing only part of the data structure. The history-insensitivity constraint of functional languages prevents any form of data modification, and any change to part of a data structure involves making a new data item, identical to the first, except for the required alterations. This is extremely inefficient for large data structures [347], and has led to the development of special hardware and software constructs to support structured data [349, 372]. From this, functional languages do not appear to be well-suited to image analysis applications.

12.1.3 Automatically Partitioned Imperative Languages

In systems which use automatically partitioned imperative languages, the allocation of tasks to the available processors is done automatically by the compiler. This involves the detection of data dependencies, and analysis of control flow within the program [384, 385, 386]. Although the automatic partitioning of programs is highly desirable, such systems are extremely difficult and time-consuming to implement, and this was deemed to be beyond the scope of this project.

12.1.4 Manually Partitioned Imperative Languages

Special constructs may be introduced into a conventional imperative programming language, to allow parallelism to be explicitly expressed. Such systems are more difficult to use for developing applications programs than automatically partitioned systems, but the system implementation is considerably simpler. Languages such as this are frequently used to program homogeneous SIMD systems, using simple parallel constructs in which a statement may be executed by all available processors, using different data. Examples of such languages are PascalPL [387] and DAP-FORTRAN [388, 390]. In a system with multiple instruction streams, however, this expression of parallelism at a statement level may result in an unacceptable amount of inter-process synchronisation.

Concurrent Pascal [391, 392] is a manually-partitioned imperative language based on the concepts of processes and monitors. A process, in this context, is a sequential section of code which operates on a set of locally-declared private variables. All shared variables are declared within structures known as monitors. Each monitor also contains a set of monitor procedures, which are the only mechanism by which this shared data may be accessed. The monitor procedures are, themselves, globally accessible, and shared data must always be accessed through these. Monitors and processes are declared statically in the Concurrent Pascal program, and may not be dynamically created. These constructs provide a method of ensuring that all manipulation of shared data is made in a controlled manner, and exclusive access to data is achieved by providing exclusive access to monitor procedures. The use of monitors would, however, be likely to be inefficient in image analysis applications, since the accessing of shared image data which occurs in such programs would require frequent calls to the monitor procedures, and this is a time-consuming process.

12.1.5 MIMD-Pascal

It was decided to use a manually partitioned imperative language as the programming language for the multiprocessor simulator system. A compiler for a language based on Pascal [393, 394, 395], to be known as MIMD-Pascal, was written. This is a conventional block-structured language, with constructs which allow multiple instruction stream programming, with shared data and code. In this language, parallelism is expressed at a block level. The MIMD-Pascal source code is compiled into a set of assembly-language

instruction streams, one for each simulated processor, by an optimising compiler. Optimisation is performed because machine performance may be greatly affected by the ratio of memory accesses to instructions executed, and this is generally greater in unoptimised compiled code than in hand-written, or machine-optimised, code.

12.2 The Compiler Target Language

The compiler takes an MIMD-Pascal source program and creates a text file containing a number of fields of integers, corresponding to Motorola MC68000 instructions [359, 360, 361, 362]. This object file is passed to the simulator, which produces files of results (described previously in chapter 11). Each instruction in the compiled object file consists of a number of numeric fields. These are not packed to give true MC68000 instructions, as doing so would waste time in both the compiler and the simulator.

The system configuration file, which must be present at both compile-time and simulation-time, describes the required number of processors and memory units, their parameters, and the network topology. Options are provided, by means of pragmatic comments in the MIMD-Pascal source code, which cause the output code to contain full mnemonic listings, with automatically-generated comments, in addition to the numeric instruction fields.

12.3 A Description of MIMD-Pascal

The compiler source language is based on a subset of Pascal, with extensions to support multiple instruction streams. These take the form of constructs known as tasks, which are similar in structure to processes in Concurrent Pascal, but one task is allocated to each available processor, and runs to completion on that processor.

12.3.1 Block Structure

The block structure of MIMD-Pascal is shown in Fig. 12.1. The global declaration sections contain declarations for shared constants, types and variables which are common to all tasks. All variables which are to be accessed by more than one processor are declared here. These are referred to as

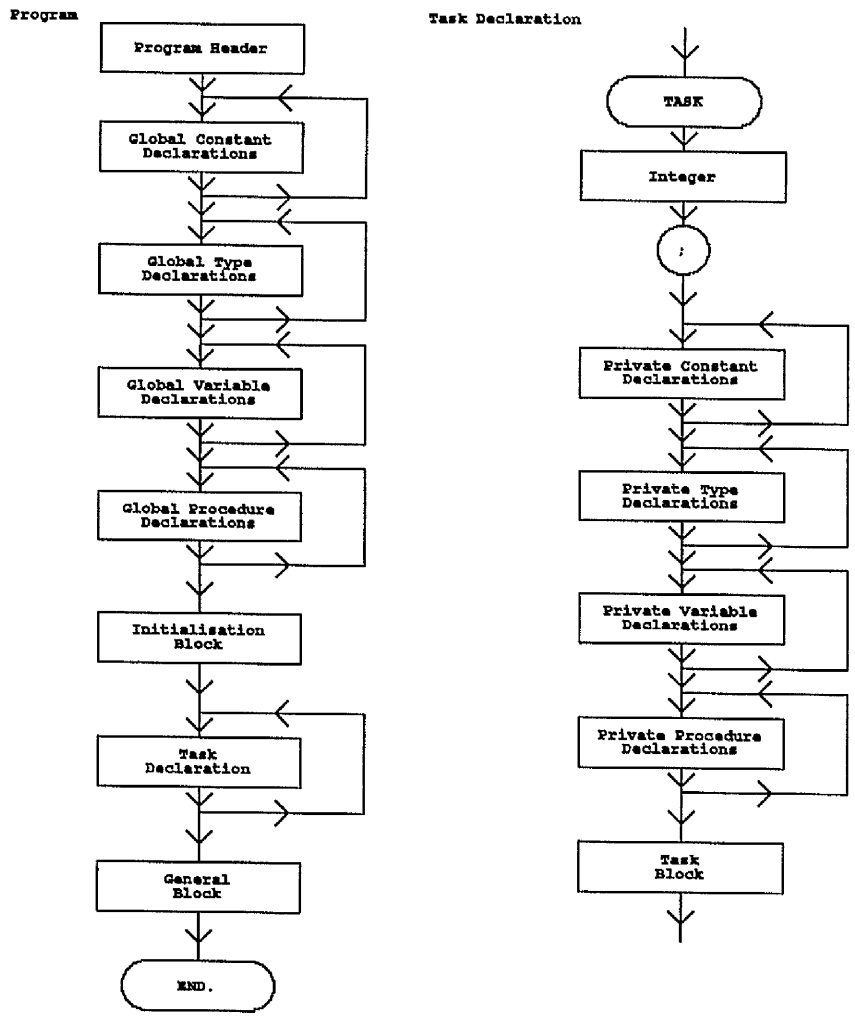


Fig. 12.1: The block structure of MIMD-Pascal.

globally-accessible variables, or global variables. Global procedures are defined in exactly the same way as standard Pascal procedures, and may be called from any section of the program. The use of global procedures avoids unnecessary duplication of code for common sub-sections of programs.

The initialisation block is executed sequentially, by a single processor, before any other task is started. This provides a facility to initialise shared variables and semaphores (discussed later in this chapter). The task blocks contain the statements which are to be executed by individual processors. Each task may contain local declarations and procedures which, in accordance with Pascal scope rules, are not accessible by other tasks. The general block contains code to be split, by the compiler, amongst

the remaining unallocated processing units. In the current implementation, this feature is not supported.

12.3.2 Data Structures

The current implementation of MIMD-Pascal supports integer variables and constants, boolean variables and constants, one-dimensional arrays, and records. Arrays may be of any type, including array types, so that the one-dimensional constraint is not restrictive. Records may have any number of fields, of any type. Within the compiler, the comma (,) is used as a structuring operator, with the result that a record type is type-compatible with a list of other variables, of appropriate type, separated by commas. For example, given the declarations:

```
VAR R   :RECORD
            A   :INTEGER;
            B   :BOOLEAN;
            C   :ARRAY[1..10] OF INTEGER;
        END;

    X   :INTEGER;
    Y   :BOOLEAN;
    Z   :ARRAY[1..10] OF INTEGER;
```

then

```
R:=X,Y,Z
```

is a legal statement. In a similar manner, constants of record types may also be created. This violates the strong typing constraints of Pascal, but was felt to be a useful feature; for example, in image analysis, it is frequently necessary to assign separate values to X and Y co-ordinates, whereas the natural structure to represent a point is a record of two integers. A similar type-compatibility occurs in the passing of procedure parameters, where a variable may be passed as either a record or a list of its individual constituent fields.

12.3.3 Control Structures

A minimal subset of Pascal control structures was implemented in the MIMD-Pascal compiler. This comprises REPEAT loops, FOR loops, IF-THEN-ELSE statements and PROCEDURE calls, with parameters passed by value. These constructs operate in the manner defined in British Standard BS6192

(ISO 7185) [394]. It was not considered worthwhile, for the evaluation of the simulated system, to implement other control constructs, such as the WHILE loop, or CASE statement, since these may be easily replaced in test programs by REPEAT, or IF-THEN-ELSE statements.

12.3.4 Input and Output

Two procedures, READ and WRITE, which operate on integers only, are provided in MIMD-Pascal, and these cause the simulator to display results, or prompts for input, in the lower display window of the console. In a simulated system, input and output operations are rarely required, since the value of any variable may be examined directly by the user at any time.

12.4 Process Synchronisation

Process synchronisation is a major problem in multiprocessor machines, and requires the provision of special mechanisms to avoid corruption of shared variables in situations where more than one processor attempt to modify these variables simultaneously. The problem may be seen in an example where two processors, A and B, both wish to increment a variable X. One possible sequence of events is:

i) Processor A reads X from the shared memory, and places it in a register.

ii) Processor B reads X from the shared memory, and places it in a register.

iii) Processor A increments the copy of X in its register.

iv) Processor B increments the copy of X in its register.

v) Processor A writes the register contents back to the shared memory.

vi) Processor B writes the register contents back to the shared memory.

This sequence results in the variable X having been incremented only once, not twice as intended. To prevent this interference, a mechanism must be provided which allows a processor to have exclusive access to a variable. These are provided in MIMD-Pascal, in the form of the procedures INCREMENT, DECREMENT, WAIT, and SIGNAL. Each of these takes one integer parameter, which is passed by name, so that altered values may be returned.

INCREMENT causes an integer variable to be incremented in an indivisible operation, performed in the memory unit itself by means of an increment-in-memory instruction. The operation is

not susceptible to interference by any other processes which may attempt to access the same variable. DECREMENT operates in a similar manner. These operations are similar to Gottlieb's 'AltFetchand-Add' operations [396], but require simpler arithmetic functions in the memory unit. WAIT and SIGNAL are the semaphore primitives defined by Dijkstra [172, 397]. SIGNAL is, in fact, identical to INCREMENT. WAIT tests the variable concerned and, if it is greater than zero, the variable is decremented in a similar manner to DECREMENT. If the variable is less than or equal to zero, the process calling WAIT is suspended until the variable is greater than zero, when it is decremented and the process is allowed to continue. The implementation of these procedures is discussed later in this chapter.

If an integer variable, known as a semaphore, is used as the argument of a pair of WAIT and SIGNAL operations which bracket some linear section of program code, then this code is said to be protected. If the semaphore is initialised to some value, N, before any processor is activated, then at most N processors may execute the protected code at any one time. Code which must be protected in this manner, for correct program execution, is known as critical code. Gottlieb [396, 398] and Deo [400] discuss the necessity of avoiding critical sections of code and, in particular, job queues. Gottlieb [398] describes a method of job queue management in which waiting time may be reduced. Primitives equivalent to INCREMENT and DECREMENT are used on pointers to the queue, and 'known minimum' and 'known maximum' values for these pointers are maintained. Although waiting time may be reduced, this scheme is complex, and still contains some critical sections.

12.5 Run-Time Stack Organisation

The simulated machine memory is divided into shared program space, system space, global variable space, and one private stack for each processor. The system space is occupied by semaphores concerned with start-up and shut-down synchronisation, and by the static display structures [401, 402] for each processor. Since the first textual level of the program contains the shared global data, the first element of each display structure points to the global variable space. The size of this global storage area is determined at compile-time. Local variables for each task are stored on the private stacks but, although these stacks are designated private, they are stored in part of the shared memory.

12.6 Compiler Implementation

The MIMD-Pascal compiler may be divided into six parts:

i) the lexical analyser;

ii) the parser;

iii) the graph builder;

iv) the graph manipulator;

v) the code generator;

vi) the linker.

This division is illustrated in Fig. 12.2. The sections interact by passing lexical 'tokens' from one section to the next, and also via two shared data structures, the name list and the directed acyclic graph (DAG) [401, 402]. The name list contains some information about identifiers, but all information concerning data types is held on the DAG, and is pointed to by the name list entries. The DAG also holds representations of each program block in turn, as non-recursive graph structures, which may be easily manipulated by the compiler. This data structure is described in detail, later in this chapter.

The lexical analyser is of a conventional type [403, 402], and changes the input file of characters into a stream of lexical tokens. These are passed to the table-driven parser [403, 402], which checks the stream of tokens for syntactic correctness. Errors are reported, but no error recovery is attempted. This would introduce additional complexity into the compiler, which is not intended to be a generally usable compiler, but a merely a tool for generating suitable object files for the low-level simulator. Programs are likely to be relatively short and correcting one error at a time is not excessively time-consuming in such cases. The parser passes a stream of syntactically-checked tokens to the graph builder, which maintains the name list, re-arranges the input stream, and constructs the DAG.

When a complete block has been assembled on the DAG, this is optimised by the graph manipulator, which simplifies and re-arranges expressions, eliminates common sub-expressions, and could, although this is not fully implemented, re-arrange code to remove loop-invariant operations. After optimisation, the DAG is dismantled by the code generator, which produces a file of target machine code instructions. A number of register descriptors are used by the code generator to allow the re-use of

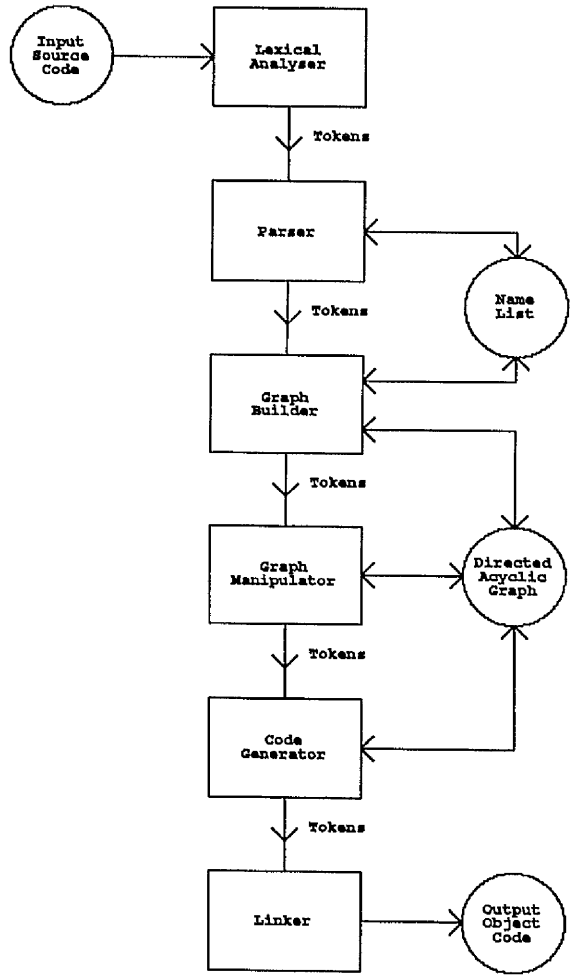


Fig. 12.2: The structure of the MIMD-Pascal compiler.

values already held in registers, wherever possible. Control structures are implemented by inserting one or more built-in sections of hand-written code for each structure. A list of forward references is constructed during this phase, and these are filled by the linker, which makes a single pass through the output file, to complete the code file.

12.6.1 The Structure of Lexical Tokens

Lexical tokens are passed from one section of the compiler to the next, and form the main method of communication between different sections of the compiler. These tokens are used to transmit parts of programs, and also to invoke actions in the next stage of the compiler. In this manner, each section

drives the following one by sending appropriate control tokens. Each lexical token comprises three fields:

i) The string field

This field contains the string from the input file, which originally generated this token. The string contains no essential information, but is retained in all tokens for ease of compiler de-bugging.

ii) The symbol field

The symbol within each token defines the basic form of this token. The symbol is represented by an enumerated type, and symbols exist for

a) each reserved word (*program*, *begin*, *end*, *for*, *if*, etc.);

b) each punctuation symbol(:=, ;, [,], etc.);

c) each arithmetic operator (+, −, *, /, etc.);

d) constant symbols (*signed_integer*, *unsigned_integer*, *boolean_constant*, etc.);

e) a read from, or a write to, a variable (*rd_contents*, *wr_contents*, etc.);

f) each different form of identifier (*constant_identifier*, *type_identifier*, *variable_identifier*, *procedure_identifier*, etc.);

g) non-terminal symbols, used by the parser in recognising the structure of the input program (*procedure_declaration*, *identifier_list*, *block*, *statement*, *expression*, etc.);

h) pseudo-symbols, used for control purposes within the compiler. Some pseudo-symbols are generated by the parser, to mark the end of structures where no other unique lexical token is generated. An example of such a situation is the 'END' at the end of a procedure definition, or at the end of a record declaration. Different action is to be taken by the graph constructor in each case, and so different tokens (*end_procedure* and *end_record*) are generated by the parser.

iii) The value field

Each lexical token contains an integer value, but this is only significant for some classes of token.

For identifier tokens, this field holds a pointer to the name list and, for constant tokens, the token value is the value of the constant.

12.7 Directed Graph Structure and Manipulation

The DAG is the largest data structure in the compiler, and is used to store a representation of each program block in turn, and also to store all information concerning data types. Each node of the DAG contains the following items:

i) A token

This is a lexical token, as described previously, and the symbol within this token defines the form of the node. The value field of the token contains any appropriate integer, such as a constant value, or a pointer to the name list.

ii) Left and right child pointers

Two pointers indicate the left and right 'children' of this node. In some cases, depending on the form of the node, only one child exists. No pointers to 'parent' nodes are stored since, in a graph structure, a node may have more than one parent.

iii) A data type

Each node contains a data type, which is evaluated according to the types of its children, to allow type-checking of the program. Nodes relating to structured data have the special type '*structure*', and two *structure* nodes are type-compatible if both pairs of children are type-compatible.

iv) A reference count

A count of pointers to each node is maintained, and if, for some node, this reaches zero as a result of DAG manipulation, the space occupied by this node may be recovered and re-used.

The order of construction of the DAG is such that left sub-trees are constructed; then right sub-trees; then the two are linked. This is related to the construction of tree structures from arithmetic expressions in reverse polish notation [402], but is expanded, here, to apply to all forms of node. This process is recursive, and the DAG is dismantled in the same order. The DAG is used to store two types of information, declarative and imperative, and these are discussed separately.

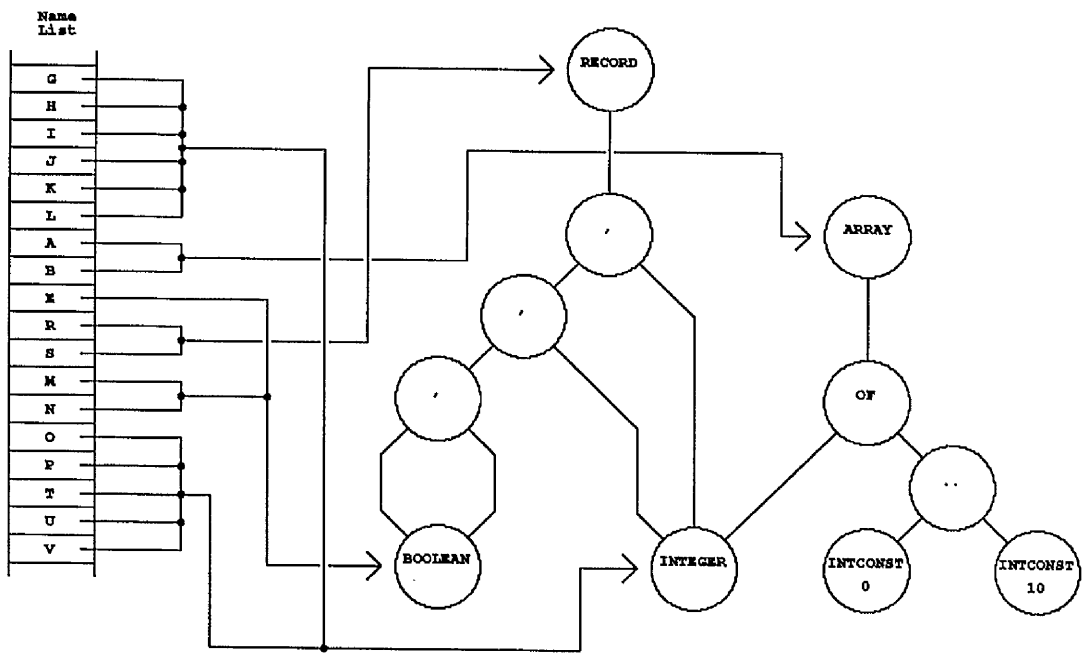


Fig. 12.3: Name list and DAG structures for variable declarations.

12.7.1 Declarative Information

The declarative information which is stored on the DAG takes the form of a number of sub-graphs, each of which contains a single type declaration. All name list entries of a particular type point to a single representation of that type on the DAG, and sub-graphs may be shared between types. Fig. 12.3 shows the DAG and name list pointers for the following declarations:

```
VAR G,H,I,
    J,K,L    :INTEGER
    A,B      :ARRAY[1..10] OF INTEGER;
    E        :BOOLEAN;
    R,S      :RECORD
                  M,N :BOOLEAN;
                  O,P :INTEGER;
              END;
    T,U,V    :INTEGER;
```

It may be seen that all variables of the type INTEGER point to a single node, including the INTEGER fields within the record declaration. This sharing of nodes for common sub-expressions may result in graphs with complex connectivity. Procedure names are entered into the name list when they are

declared; that is, when the procedure header is encountered. This means that the procedure is visible from within its own body and, thus, recursion is allowed.

12.7.2 Imperatives

A complete representation of each program block is assembled on the DAG, so that this may be manipulated and, thus, optimised. To allow this, each program construct is represented by a corresponding DAG structure.

12.7.2.1 Expressions and Assignments

Expressions are represented on the DAG as simple tree-like structures, where leaf nodes represent operands, and interior nodes represent operators. In general, an expression may have a number of different possible representations. Fig. 12.4 shows three possible forms of the expression $A + B * C - D$. The third of these is preferred, since trees which are deeper towards the left may be evaluated with less storage of temporary results, when evaluated in a left-to-right manner. Figs. 12.5, 12.6 and 12.7 show the DAG representation for the following program block:

```
TASK 1;
    BEGIN
    I:=0;
    REPEAT
        A[I]:=I;
        I:=I+1;
    UNTIL I>10;
    FOR I:=0 TO 10 DO
        B[I]:=I;
    H:=-1;  I:=42;  J:=55   K:=19;
    L:=H+I*J-K;
    I:=14+4*I-3;
    SIGNAL(G);
    END;
```

These DAG representations have, in fact, been optimised but, before the nature of these optimisation is discussed in detail, the graph is, itself, described. The form of a simple assignment statement may be seen at the top left of Fig. 12.5. The expression, in this case the constant zero, is placed on the left

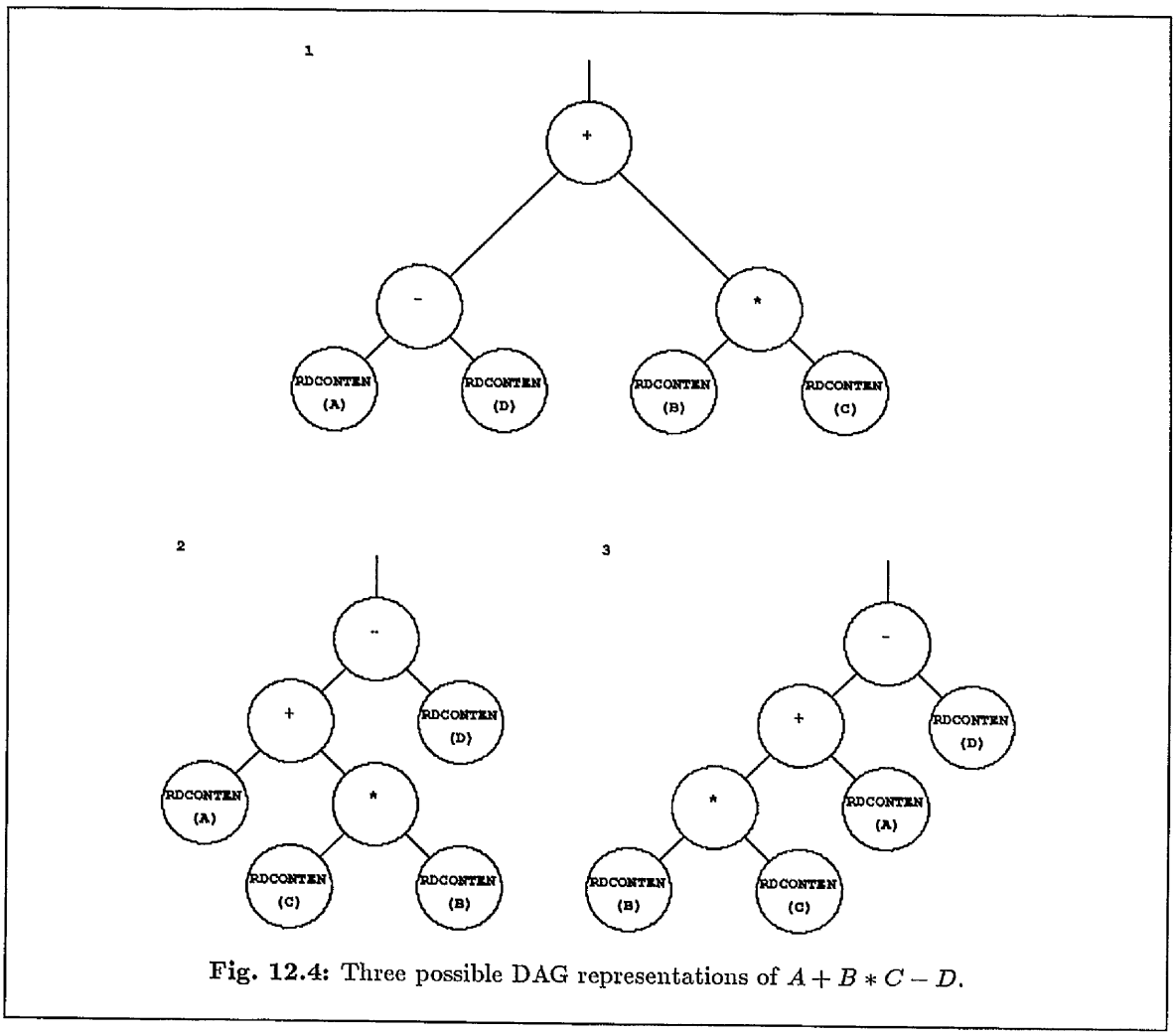


Fig. 12.4: Three possible DAG representations of $A + B * C - D$.

hand side of the *becomes* (:=) node and the address of its destination is placed on the right hand side. This arrangement causes the expression to be evaluated before its destination address is required.

Any sub-graph which is the same as some other sub-graph is removed, and replaced by a pointer to the now-shared structure. This may be seen in the sharing of the *rd_contents* node at the bottom of Fig. 12.5, which represents the contents of the variable I. Nodes which represent reads from variables may only be shared provided no *wr_contents* (write to a variable), relating to the same variable, occurs between them. This ensures that, once data is updated, any copy of it which may remain in a register from an earlier calculation is no longer used. For this reason, a second, unshared, read from I occurs to the right of Fig. 12.5. *Wr_contents* nodes are never shared.

Most elimination of common sub-expressions takes place during DAG construction, but this does

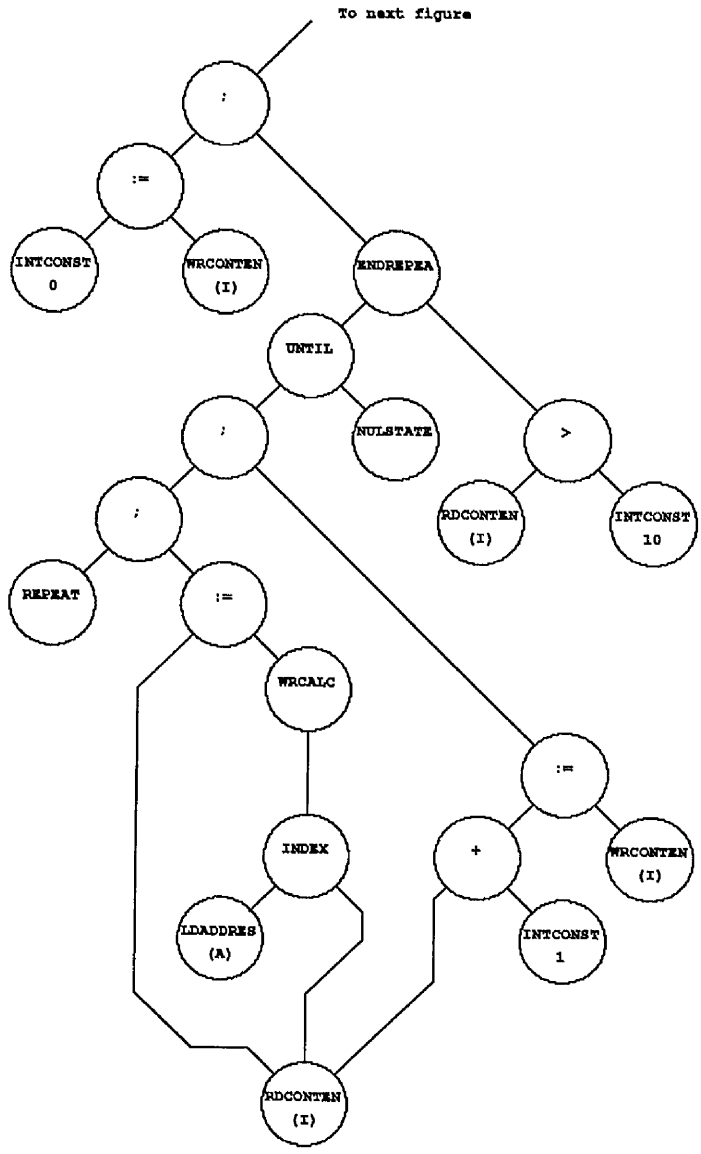


Fig. 12.5: Typical DAG structures (I).

not necessarily detect all possible cases, since the simple algorithm used does not attempt to re-arrange expressions to check for equivalence. Further elimination of common sub-expressions takes place during the optimisation phase.

Array indexing is also shown in Fig. 12.5. This is performed by the use of an *index* symbol, which represents an addition operation which operates on an address and some other data type, with appropriate scaling according to the size of the data item. The *wr_calc* symbol performs a write to this

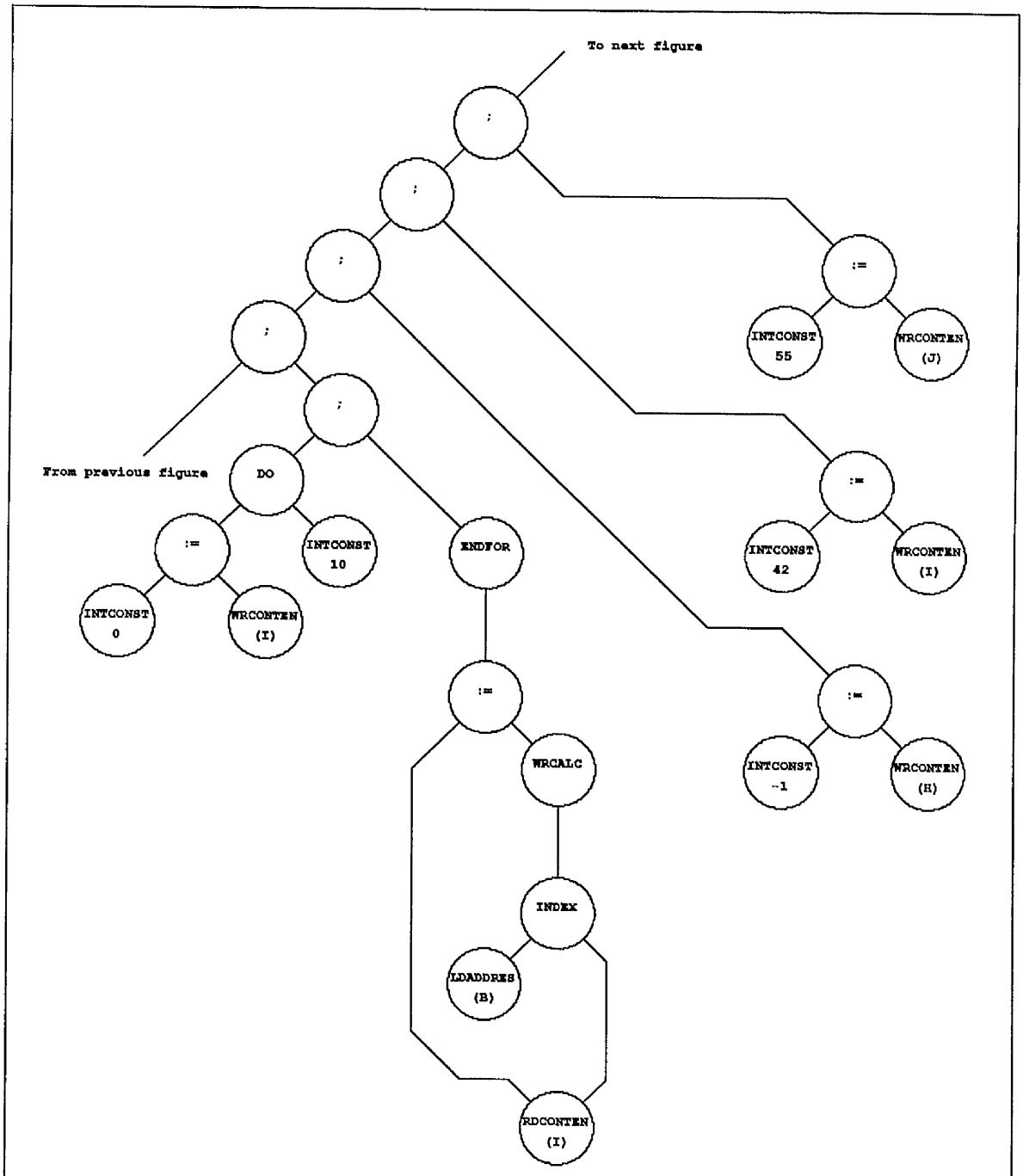


Fig. 12.6: Typical DAG structures (II).

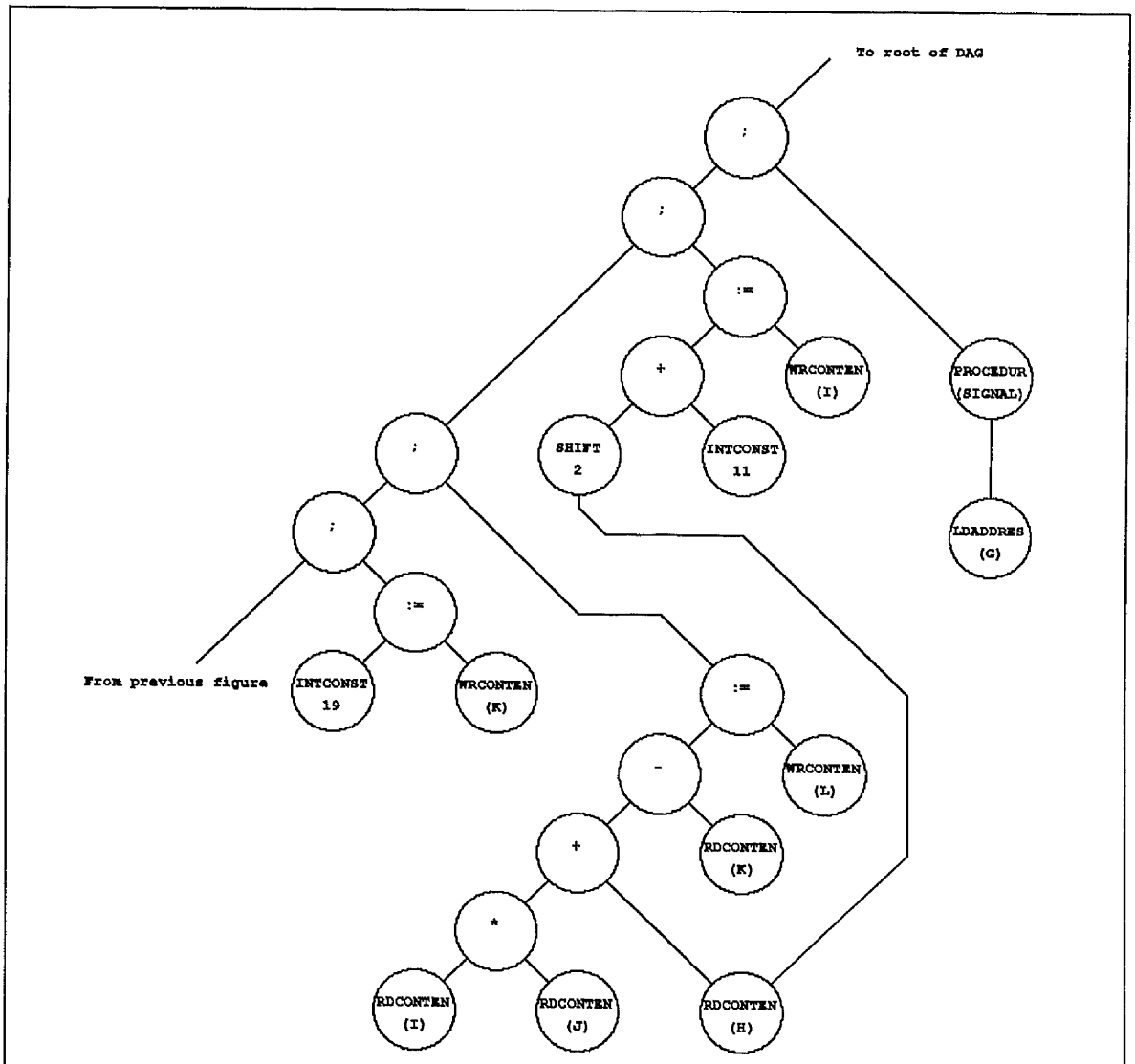


Fig. 12.7: Typical DAG structures (III).

calculated address, and a *rd_calc* symbol is used to when a similar read operation is required.

12.7.2.2 Control Constructs

Statements are linked together by semicolon nodes, which imply that the statement on the left subgraph is to be executed, followed by that on the right. All control constructs are represented by special DAG structures, whose order of evaluation is also left-to-right. The REPEAT-UNTIL loop, shown in Fig. 12.5, consists of a *repeat* node which serves to mark the start of the loop, linked in the normal manner to other statements, until an *end_repeat* node is encountered. Here, an expression occurs on

the right sub-graph, and this is evaluated as the termination expression for the loop. The *until* node acts in the same way as as a semicolon, to link the last two statements in the loop, since a semicolon is not required before an UNTIL, in Pascal. As a redundant semicolon does occur in the example, a null statement has been generated. The structure of FOR loops is shown in Fig. 12.6, and Fig. 12.7 shows a number of assignments, and a procedure call to the standard procedure SIGNAL. This uses a *ld_address* node to load the address, rather than the value, of its parameter so that the procedure may return an altered value. Other control structures are represented in similar ways.

12.7.3 Optimisation

When a complete block has been assembled on the DAG, graph manipulation is performed. This is done according to the optimisation level, which may be set by the programmer, using pragmatic comments. Three different forms of optimisation are defined:

i) re-arrangement and reduction of expressions, to improve code efficiency;

ii) re-use of register values, wherever possible;

iii) re-arrangement of loops, and evaluation of loop-invariant expressions outside loops [405, 386].

The first of these is performed by the graph manipulator, and the second, by the code generator. A system to move loop-invariant code was partially implemented, but this is non-trivial and was not completed. The optimisation performed by the graph manipulator may be divided into four sections.

i) Expression re-arrangement

Wherever possible, expressions are re-arranged so that the sub-graph to the left of any node is deeper than that to the right. This is done so that run-time expression evaluation requires less storage of temporary results. To perform this re-arrangement, addition, and multiplication may be directly reversed, and subtraction and division may be reversed, and the operator changed to a reverse subtraction, or reverse division. This is shown in Fig. 12.8 for the expression $A/(B - C * D)$, which would otherwise require the storage of two partial results.

ii) Constant folding

If an arithmetic node has two children which are constants, the expression is evaluated at compile-time, and the sub-graph is replaced by an appropriate constant node. This applies to all integer

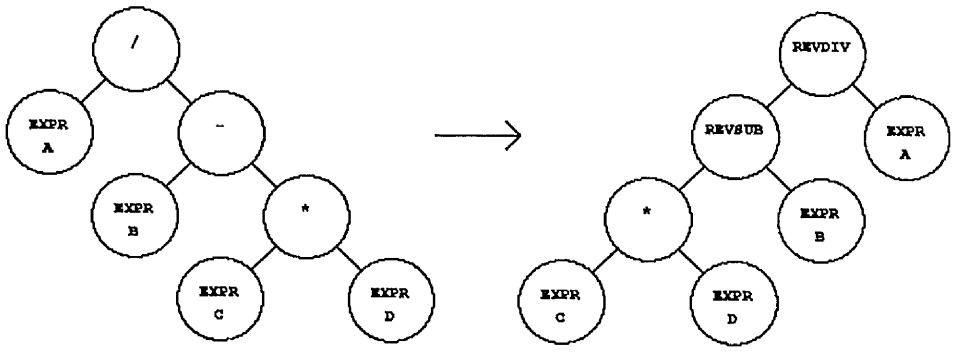


Fig. 12.8: Expression re-arrangement.

and boolean operations. If an arithmetic node has one child node which is a constant and one which is an expression, the two are arranged so that the constant is on the right hand side. More complex expressions may be manipulated to move constants towards the right of the graph. Fig. 12.9 depicts the interchange of operands in addition operations, for the expressions $(A+P)+B$ and $(A+P)+(B+Q)$. This is performed in a similar manner for multiplication. For subtraction and division, it is necessary to make alterations to the operators used, as shown in Fig. 12.10 for the expression $(A/P)*(B/Q)$. Repeated application of these rules ensure that any constants in an expression migrate to the right hand side of the expression, where they are evaluated.

iii) Operator strength reduction

A number of special constructs are recognised and converted into simpler forms, as shown in Table 12.1. Some of these optimisations are described by Hanson [406].

iv) Elimination of remaining common sub-expressions

Multiple occurrences of sub-expressions, which may be revealed by other graph manipulations, are replaced by shared sub-expressions.

These manipulations are repeated, until no more alterable expressions can be found on the DAG, at which time the code generator is invoked. Owing to difficulty involved in the direct evaluation of certain constructs, the code generator is unable to cope with the unoptimised DAG, and so, unfortunately, optimisation may not be switched off.

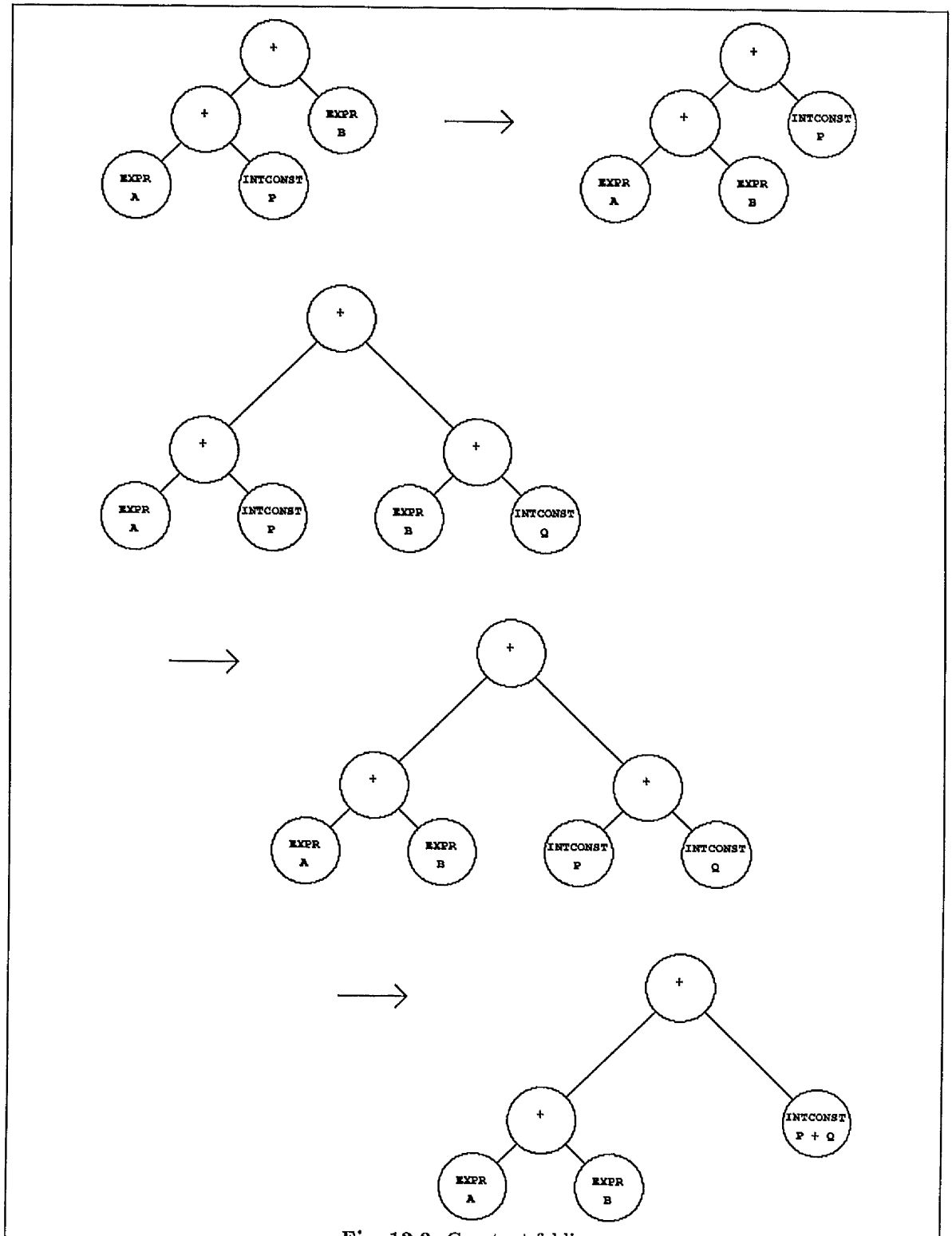


Fig. 12.9: Constant folding.

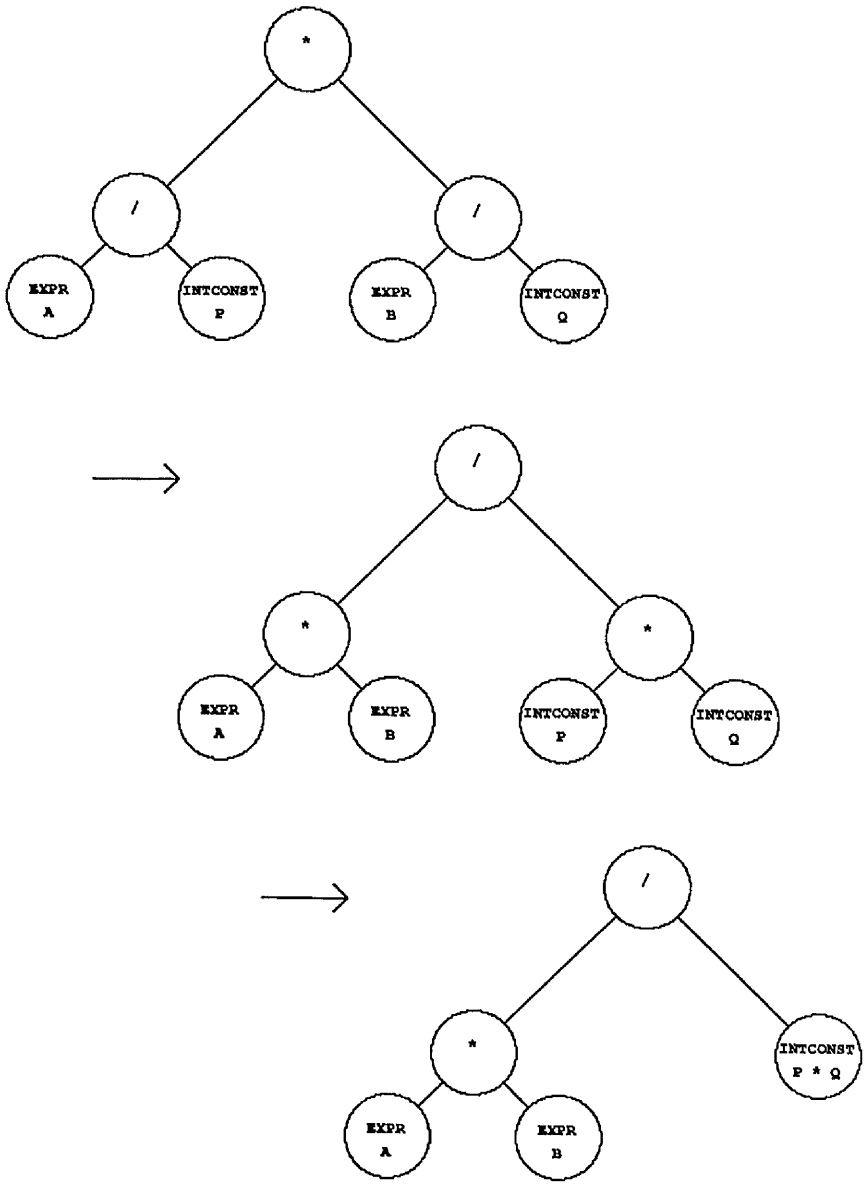


Fig. 12.10: Constant folding for non-commutative operators.

Table 12.1 Operator Strength Reduction	
Operation	Reduced-Strength Operation
Addition of zero	Operation removed
Addition of negative constant	Subtraction of positive constant
Addition of two identical expressions	Shift left by 1 bit position
Subtraction of zero	Operation removed
Subtraction of negative constant	Addition of positive constant
Subtraction of two identical expressions	Constant zero
Multiplication by zero	Constant zero
Multiplication by one	Operation removed
Multiplication by power of two	Shift left
Division by zero	Error
Division by one	Operation removed
Division by power of two	Shift right
Division of two identical expressions	Constant one
Shift by zero bit positions	Operation removed

12.8 Code Generation

Code generation takes place once all possible optimisations have been made. The DAG is traced, in a left-to-right manner, and instructions are generated according to the form of each node. Common sub-expressions are detected, by checking for the use of shared nodes in the DAG, and provision is made to re-use values already loaded to registers, wherever possible. A set of register descriptors are maintained, which keep a record of the current contents of each register. This is necessary, since it may be necessary to overwrite the contents of a register, before the opportunity to re-use it occurs, and this case must be detected. Each instruction produced by the code generator is tagged, according to the purpose of the instruction, so that some estimate can be made of the amount of time spent, by the simulated machine, on each category. The categories of instructions are:

i) Initialisation instructions

These are instructions concerned with execution of the initialisation block of the program.

ii) Synchronisation instructions

Instructions which are marked as 'synchronisation' include those within the system synchronisation procedures, WAIT and SIGNAL, and also the instructions in the main program which are concerned with loading the parameter addresses for calls to these procedures.

iii) Start-up instructions

All instructions, other than those in categories i) and ii), which are to be executed before a processor enters its TASK block are marked as start-up instructions.

iv) Shut-down instructions

All instructions which are to be executed after a processor leaves its TASK block are marked as shut-down instructions.

v) Useful instructions

All other instructions are marked as 'useful', since they are generated by the program, rather than by the system.

The code generator uses short instructions wherever possible, to reduce code length, and execution time.

12.9 System Procedures

The implementation of some of the system procedures is now described. These take the form of hand-written machine code which is implanted in the output code. Built-in procedures are required to perform synchronisation operations, and to start up and shut down the machine in an orderly fashion.

12.9.1 Synchronisation Procedures

The system procedures INCREMENT and DECREMENT may be easily implemented by use of increment-in-memory (IIM) and decrement-in-memory (DIM) instructions. Since only a single instruction is required, a procedure call is not generated in the output code; instead, the appropriate instruction is inserted directly. The system procedure, SIGNAL, calls a machine-code routine which increments the semaphore in memory, using an increment-in-memory instruction. A procedure call is used in this case, so that the SIGNAL procedure may be easily modified, at some later date, to improve the efficiency of the WAIT and SIGNAL mechanism.

A straightforward implementation of the WAIT procedure is shown in Fig. 12.11. The semaphore is decremented, and the value obtained is tested. If this is greater than or equal to zero, then the

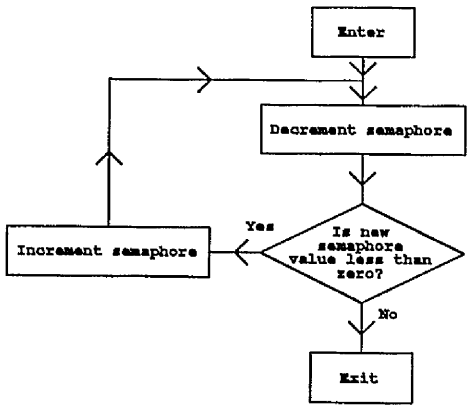


Fig. 12.11: The simple WAIT algorithm.

WAIT operation has succeeded, and the procedure is exited. If the decremented value is less than zero, the WAIT has temporarily failed, the semaphore is incremented, and the decrement operation is immediately re-tried. The use of this simple algorithm may, however, result in total deadlock of the simulated machine. This situation may arise whenever three or more processes perform a WAIT operation on the same semaphore at the same, or almost the same, time. Two of the three processors fail to enter the critical section, and remain in the WAIT procedure. They follow identical courses of action, and remain closely synchronised in a loop. When, at some later time, the semaphore is incremented by the processor which was successful in the initial WAIT, each of the remaining processors prevents the other from observing the semaphore in its new, incremented, state. One possible sequence of such events, for three processors A, B, and C, operating on a semaphore which is initially '1', is:

i) Processor A calls WAIT on the semaphore;

a DIM instruction is decoded in A;

the DIM request from processor A arrives at the DMU;

the semaphore is decremented to 0, and a reply packet is dispatched to A;

processor A enters the critical section.

ii) Processor B calls WAIT on the semaphore;

a DIM instruction is decoded in B;

the DIM request from processor B arrives at the DMU;

the semaphore is decremented to -1, and a reply packet is dispatched to B.

iii) Processor A exits the critical section, and calls SIGNAL;

an IIM instruction is decoded in A;

the IIM request from processor A arrives at the DMU;

the semaphore is incremented to 0, and a reply packet is dispatched to A;

processor A takes no further part in the proceedings.

iv) Processor C calls WAIT on the semaphore;

a DIM instruction is decoded in C;

the DIM request from processor C arrives at the DMU;

the semaphore is decremented to -1, and a reply packet is dispatched to C.

v) Processor B receives the value -1, and an IIM instruction is decoded in B;

the IIM request from processor B arrives at the DMU;

the semaphore is incremented to 0, and a reply packet is dispatched to B;

processor B re-tries its wait operation, and a DIM instruction is decoded in B;

the DIM request from processor B arrives at the DMU;

the semaphore is decremented to -1, and a reply packet is dispatched to B.

vi) Processor C receives the value -1, and an IIM instruction is decoded in C;

the IIM request from processor C arrives at the DMU;

the semaphore is incremented to 0, and a reply packet is dispatched to C;

processor C re-tries its wait operation, and a DIM instruction is decoded in C;

the DIM request from processor C arrives at the DMU;

the semaphore is decremented to -1, and a reply packet is dispatched to C.

vii) The whole process repeats from step v).

Although this example requires some unlikely timing sequences, many other similar sequences are possible. When more than two processors are actively involved in the loop, the timing requirements are relaxed, and such loops have occurred in simulation. When many processors execute WAIT on a single

semaphore, partial deadlocks, where the loop is eventually interrupted by some other processor, are common. Partial deadlocks were frequently observed in simulation, but it was not realised, until many simulation results had been obtained, that a total deadlock could occur. To avoid total deadlock, it seems that the best approach would be to alter the hardware decrement-in-memory instruction, so that a decrement is only performed if the semaphore is greater than zero. For correct operation, in such a system, the unaltered semaphore value must be returned. Since total deadlock was not suspected, more sophisticated WAIT algorithms were developed, to attempt to reduce the time for which a waiting process kept the value of the semaphore below its 'true' value. This does not prevent the occurrence of deadlock, but reduces the probability of it.

In the simple WAIT algorithm, the time between the DIM instruction and the IIM instruction was as short as possible, and no further reduction could be made in this. To reduce the probability that any processor might read the decremented value, a delay was introduced after the IIM instruction, which reduces the proportion of time for which the DIM is effective. A fixed delay could have been used, but a processor which failed to enter a critical section, when it is the only process attempting to do so, would then to have to wait for some considerable time before it re-tried its semaphore. For this reason, a delay was introduced which was doubled in length at every failed test operation. This was found to be unsatisfactory, since some processors tended to develop extremely long delay periods, when forced to wait for a frequently-used semaphore. To prevent this, a limit was imposed on the WAIT count, above which no increase in delay was made. This improved WAIT algorithm, shown in Fig. 12.12, successfully avoided deadlock in all but one case of simulation.

12.9.2 System Start-up

To co-ordinate the start-up process, one processor must be designated 'head processor'. This can not be done by planting different start-up code for each processor, since all processors commence execution at instruction zero of the global store, even though they may have a copy of the program in a local store. This problem is solved by providing a fixed memory location, known as a 'mirror', from which a processor may read its unit number, and so may determine if it is to be the head processor. Since the machine configuration is known at compile-time, code may be planted so that any given processor becomes the head processor. In the implemented version, this is the lowest numbered processor.

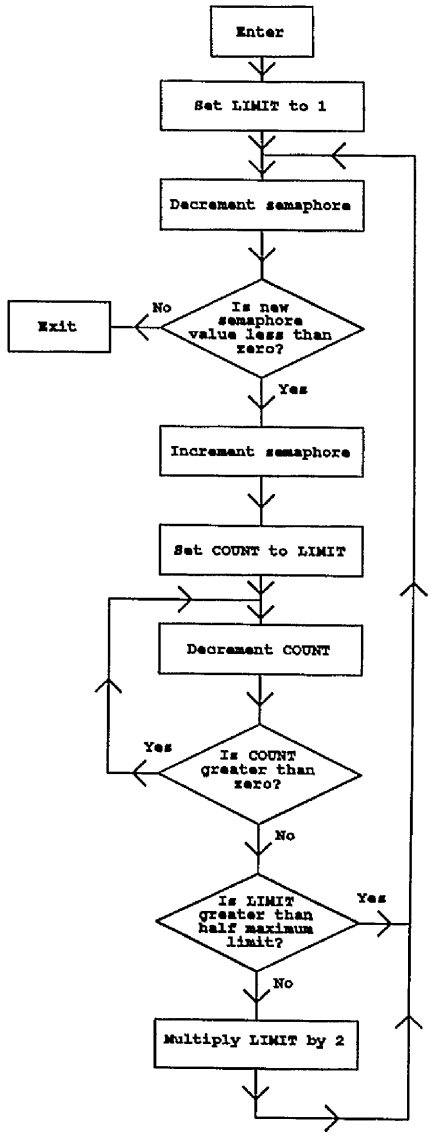


Fig. 12.12: The improved WAIT algorithm.

Two sets of semaphores are used to synchronise the start-up and shut-down operations. These are referred to as the 'running' semaphores and the 'finished' semaphores. A pair of these exists for each processor in the system. The system start-up routines perform the following operations:

i) The mirror location is read, and tested to see if this processor has been designated 'head processor'.

ii) The stack pointers and static procedure displays are initialised. Tables of these addresses are created by the compiler, and are appended to the program code file. Each processor reads

the appropriate values, using its mirror value as an index into the tables. Each processor also initialises pointers to the system data space and the global data space.

iii) All processors, other than the head processor, enter a 'suspended' state, to allow the head processor to execute the initialisation block of the program. To do this, each processor initialises its 'running' semaphore to zero and performs a WAIT operation on it. The 'finished' semaphores are also initialised to zero at this time, by the head processor.

iv) After executing the initialisation block, the head processor increments each of the 'running' semaphores, to allow all processors to proceed with their tasks.

A synchronisation problem could possibly occur in this system if, for any reason, some processor was delayed in initialising its 'running' semaphore to zero, until after the head processor performed a SIGNAL operation on it. This processor would clear the semaphore, and wait indefinitely for it to be incremented. The delay required for this to happen is very long, and is extremely unlikely to occur but, for a practical system, some suitable method of dealing with this situation would need to be developed. If all scheduling could be done at run-time, this would not be important, since a missing processor would not then affect correct program execution.

12.9.3 System Shut-down

After each processor, other than the head processor, has completed the execution of its TASK, it performs a SIGNAL operation on its 'finished' semaphore, which was initialised to zero on start-up. The processor then performs a WAIT on its 'running' semaphore, which causes it, once more, to enter a suspended state. Instead of this, the head processor performs a WAIT operation on all of the 'finished' semaphores in turn. When all of these WAITs have succeeded, all processors must have finished their tasks, and are, therefore, waiting for their 'running' semaphores to be incremented. At this point, the head processor is free to load some new program, before parallel execution is resumed.

12.10 Summary

A number of schemes for multiprocessor programming have been examined, and a parallel language, based on Pascal, has been described. The structure of a compiler for this language has also been

described. The next two chapters describe how this compiler has been used, in conjunction with the low-level multiprocessor simulator (described in chapter 11), for the evaluation of machine structures based on the partitioned indirect binary n-cube network, and the partitioned indirect binary n-tube network.

"When you're lying awake with a dismal headache,
and repose is tabooed by anxiety,
I conceive you may use any language you choose
to indulge in without impropriety."

The Chancellor, in 'Iolanthe' — Sir W. S. Gilbert

Chapter 13:

Test Programs for Simulation

13.1 Tasks for Simulation

In chapter 7, it was pointed out that the performance of a machine may depend heavily on the characteristics of the program which it running. To investigate the performance of simulated systems under various conditions, three differing processing tasks were used. These tasks were chosen to have different memory accessing patterns, and different synchronisation requirements. These tasks are:

i) Linear filtering

This task is to convolve a two-dimensional image with a two-dimensional mask, and to place the result in a new image. This task involves parallelism at the pixel level, it is simple to partition amongst several processors at compile-time, and no inter-process communication is required. The images and masks used in most simulations were 16×16 and 3×3 arrays of integers, respectively.

ii) Region filling

This task is to modify the value of all pixels which lie within a number of connected regions of a binary image, in such a manner that only those pixels which are connected to specified starting points are to be changed. The image contains several groups of pixels, or regions, which initially have pixel values of 1, on a background of 0s. This is a task similar to some found in area measurement, and region colouring. Parallelism exists, here, at a feature level, but this cannot be exploited easily by partitioning the task at compile time. Consequently, some form of run-time scheduler is required, and this is discussed later in this chapter. The images used in simulation were 128×128 binary images containing eight identical regions, each approximately 80 pixels in area. One of these regions is shown in Fig. 13.1.

iii) Integer sorting

The task, here, is to sort a list of integers into ascending order. In simulations, lists of 64 and 256 integers were sorted, using a parallel algorithm.

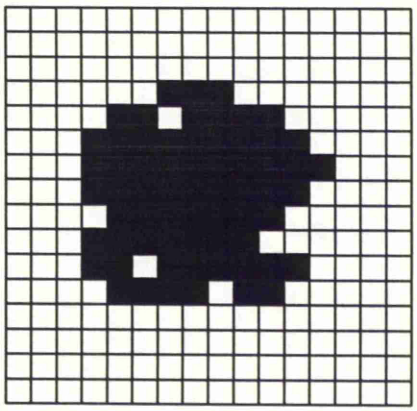


Fig. 13.1: An object for the region filling task.

13.2 Algorithms for Simulation

Several algorithms exist to perform each of the above tasks. These algorithms vary, both in their efficiency when executed serially, and in the amount of parallelism which is available. In general, there is a tradeoff situation, where the more efficient algorithms exhibit little parallelism, and the highly parallel algorithms tend to be inefficient. The algorithms which were used in simulations are described in the remainder of this chapter, and MIMD-Pascal listings of the basic forms of these programs are given in appendix 3.

13.2.1 Algorithms for Linear Filtering

Only one algorithm was considered for the linear filtering task, as it is both efficient and highly parallel. This algorithm divides the image into regions, and allocates one region to each available processor. The processors perform a space-domain convolution operation on each pixel in their region, and place the result in the destination image. To perform the convolution, each processor needs to read pixel values from adjacent regions and, to permit this, the image must be globally accessible. Task partitioning may be done either at compile-time or at run-time, but requires the number of available processors to be known, to determine the region sizes. In the simulated system, partitioning was performed at run-time, although the number of available processors was also known at compile-time.

13.2.2 Algorithms for Region Filling

Two basic methods were considered for the region filling task. These are the wildfire algorithm and the scan-line algorithm. Some variants of these are also discussed. The wildfire algorithm is highly parallel, but is relatively inefficient, whereas the scan-line algorithm has less parallelism, but is more efficient.

13.2.2.1 The Wildfire Algorithm

The wildfire algorithm is a method of filling an arbitrarily-shaped connected region with a specified pixel value. A start point is 'ignited' by placing its co-ordinates on a 'job' list. All available processors repeatedly attempt to remove points from the list, mark the indicated pixel appropriately, and to ignite all appropriate neighbours by placing their co-ordinates on the list. The algorithm terminates when all processors are inactive and the job list is empty.

The main problem encountered with the wildfire algorithm is that access to the job list must be exclusive, to prevent its corruption by a mechanism similar to that described in chapter 12. This restriction imposes a theoretical limit on the maximum speedup factor obtainable. If the time taken to execute one entire iteration of the algorithm is t_i, and exclusive list access is required for t_x within this, then, since only one processor may be in the critical section at any time, at most $\frac{t_i}{t_x}$ processors may be usefully employed. This algorithm performs quite poorly in this respect, since the amount of useful work done, for each list access, is small.

13.2.2.2 The Scan-line Algorithm

The scan-line algorithm attempts to increase the theoretical maximum speedup factor, by increasing the amount of work performed for each job list access. This may be achieved by operating on a larger region, and this is done using a region-filling algorithm developed for computer graphics applications [408, 409]. Each seed point, taken from the job list, is used as a starting point to draw a horizontal line to the left and right extremities of the region. The horizontal lines directly above and below this line are examined, and new seed points are generated at the extreme left of any disconnected section of these lines. In this manner, any convex polygon may be filled with a maximum of two seed points on the job list at any

time. Concavities in the region may cause some scan-lines to be split into a number of sections, and one extra seed point will be generated for each of these. Using this algorithm, the job lists remain quite short, and this is an advantage over the wildfire algorithm, in which the job lists may become very long.

In the version of the scan-line algorithm used for simulation, the available parallelism is increased by seeding the scan-lines directly above and below the current point, if any other processors are idle. This allows other processors to commence work on adjacent lines immediately, rather than having to wait for seed points to be generated normally. The scan-line algorithm reduces the frequency with which the processors require access to the global job list and, by increasing $\frac{t_i}{t_x}$, increases the theoretical maximum speedup. However, since fewer seed points are generated by this algorithm than by the wildfire algorithm, the available parallelism may be reduced.

13.2.2.3 Other Region-Filling Algorithms

An adaptive algorithm was investigated, which changes between the wildfire and scan-line algorithms according to the number of idle processors in the system. This attempts to maintain the amount of generated parallelism at a level which may be usefully exploited, and to use the more efficient algorithm wherever possible. No quantitative results were, however, obtained using this algorithm. Several other algorithms, including one which attempted to generate 'runners' to ignite distant parts of the region, and thus increase the task parallelism, were implemented, but not investigated in detail. In general, the additional overheads involved in such algorithms seem to outweigh any increase in efficiency, although the increase in speed could be significant if larger numbers of processors were involved.

13.2.3 The Integer Sorting Algorithm

The algorithm used for the sorting task is based on the 'quicksort' algorithm [410, 411]. An arbitrary value is selected from the list of integers, and the list is partitioned into two smaller lists, containing values larger than, and smaller than, the selected value. These two sub-lists may be similarly sorted, in parallel. This algorithm was implemented, with a job list and run-time scheduler, in a similar manner to the region-filling algorithm. For the sorting task, the job list contains pointers to the list of integers which are to be sorted. In practice, the choice of the partition value must be made with care, to allow

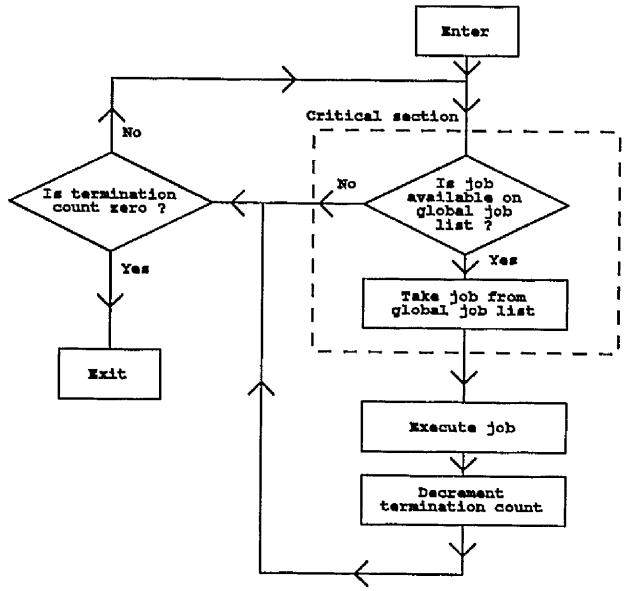


Fig. 13.2: The simple run-time job scheduler.

the list to be split into approximately equal parts and, thus, allow the maximum parallelism to be developed. It was found that selecting the value located in the middle of the list was better than using a value from one end of the list, since high or low values tend to migrate to the ends of lists, and so frequently produce partitioned lists with only one element.

13.3 Run-Time Job Schedulers

The algorithm which controls the selection of appropriate processor activity and, in particular, determines when a processor may attempt to obtain new jobs from the job list, is known as the run-time job scheduler. Such a scheduler is required for tasks which may not be divided amongst the available processors at compile-time. A number of different scheduling algorithms were investigated, to attempt to reduce conflict in accessing the job list. These were implemented in MIMD-Pascal, and listings of these are given in appendix 3.

13.3.1 A Simple Run-Time Job Scheduler

A simple run-time scheduler is shown in Fig. 13.2. In this simple scheduler, the first action is to attempt

to enter the critical section, where a job may be taken from the job list. It may be necessary to wait at this point, until the critical section is clear. Once access has been obtained to the job list, a job may, or may not, be available on it. In either case, access to the job list is relinquished immediately. If a job has been found, the appropriate procedures are executed; if not, then the job list must be re-tried, provided some jobs exist within the system. The termination count is used to keep track of the number of jobs in the system, at any time. This is initialised to be the number of jobs placed on the job list during the initialisation phase of the program and, as execution progresses, the termination count is decremented as each job is completed, and incremented whenever a new job is created. The program terminates when this count reaches zero.

13.3.2 A Run-Time Job Scheduler with Controlled Waiting

Using the simple scheduler, repeated attempts to access the job list will inevitably result in 'saturated accessing' of the job list, where several processors are waiting to enter the critical section at any time. This may mean that a process which is attempting to place new jobs on the list may be unable to gain access to the list to do so. A modified scheduler, referred to as a pre-waiting scheduler, is shown in Fig. 13.3. This scheduler only attempts to enter the critical section when the job list appears to have jobs on it. To do this, the size of the job list is read before any attempt is made to enter the critical section. The value obtained in such a read operation is not guaranteed to yield a correct result, since some other processor may be in the process of changing this variable, and may have a more recent value held in a processor register. This is not, however, significant, since the only consequence of obtaining an incorrect value is that the processor may make a decision to access the job list at a time when it will find no jobs on it, or that it may decide to defer an access when jobs were, in fact, available. Provided that the probability of a correct decision is high, program efficiency should improve. A scheme to formally allow multiple access to critical sections has been proposed by Wettstein [412], but it was not felt necessary to implement such a comprehensive scheme for this application.

13.3.3 Run-Time Job Schedulers with Private Job Lists

A potential bottleneck still occurs in the pre-waiting scheduler, since all jobs must still be obtained from a single job list. Although interference is reduced, the basic requirement of exclusive access to the job

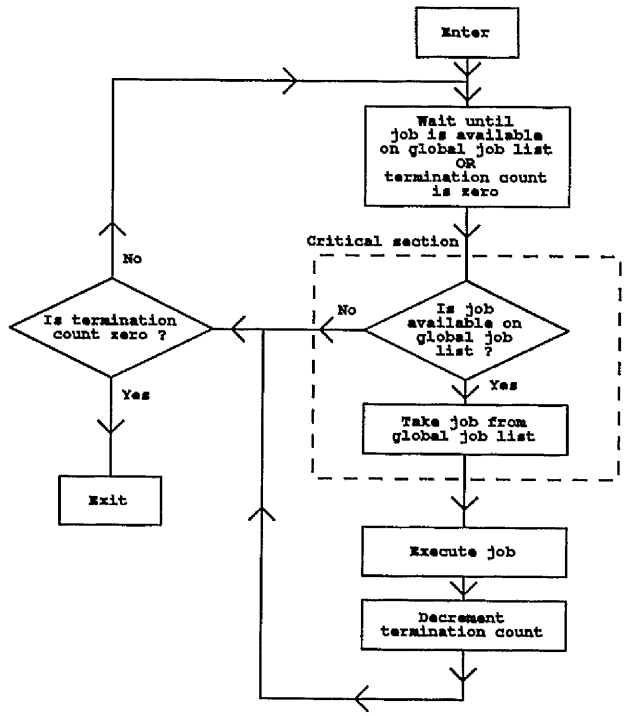


Fig. 13.3: The run-time job scheduler with controlled waiting.

list has not been removed. It is possible to reduce the number of accesses made to the global list, by using a job scheduler in which a private job list is maintained for each processor, as shown in Fig. 13.4. In this system, a processor operates as much as possible on its own job list, and only takes jobs from the global job list when its private list becomes empty.

Straightforward application of this algorithm may, however, result in all new jobs occurring in a small number of private lists, whilst the others become empty and thus processors become idle. A variant of the private-list algorithm, shown in Fig. 13.5, prevents this by maintaining a count of processors currently attempting to read from the global job list. If the number of these idle processors rises above the number of jobs on the global job list, then half of a private job list is moved to the the global list. This dynamic job re-allocation imposes an increased scheduling overhead, but gives a more even distribution of jobs between the processors in the machine.

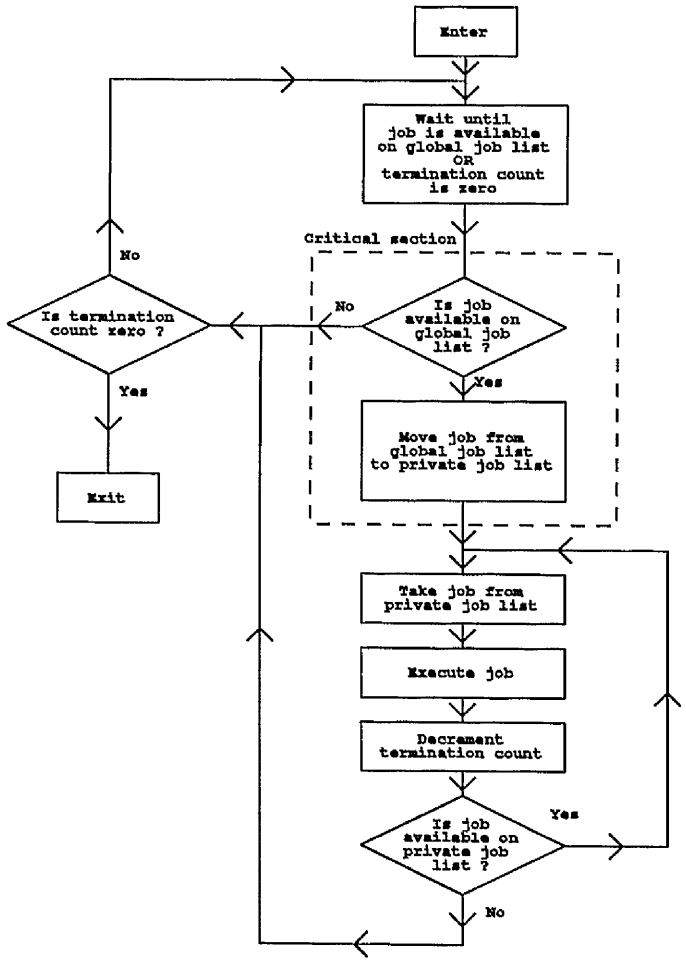


Fig. 13.4: The run-time job scheduler with private job lists.

13.4 Summary

A number of programs to perform three different tasks have been described, for the purpose of simulation using the low-level simulator program. Two of these programs require the use of a run-time scheduling system, of which several versions have been described. The next chapter describes the results of the simulation of these programs.

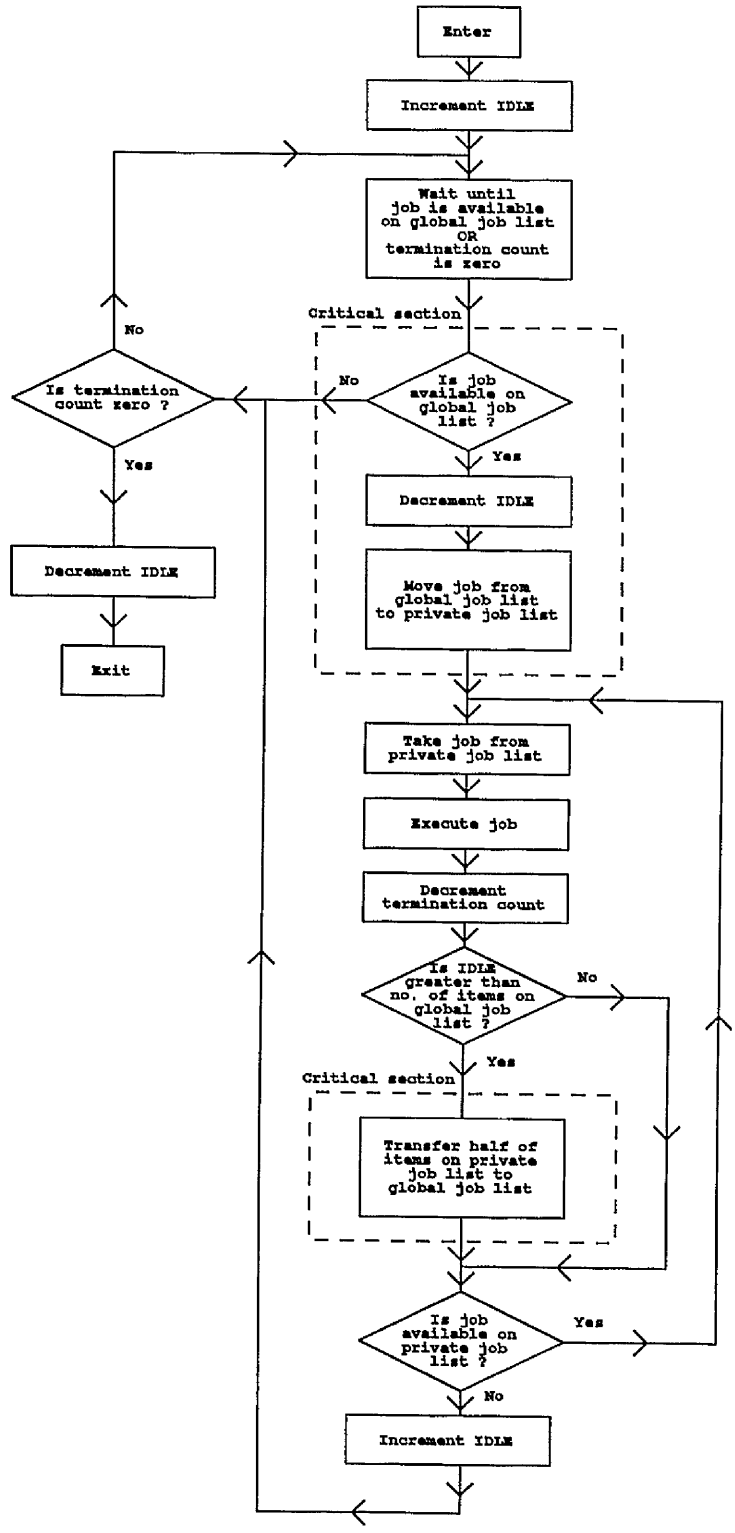


Fig. 13.5: The run-time job scheduler with private job lists, and job transfer.

Chapter 14:

Results of Low-Level Simulation

14.1 Machine Configurations for Simulation

In this chapter, the results of low-level simulation are presented for a range of different machine configurations and test programs. To obtain meaningful results from such a series of experiments, it is necessary to vary only a single parameter at any time, and to observe the resulting changes. Ideally, each of the test programs should have been simulated using each possible hardware configuration, but the time required to do this would be prohibitive. To obviate this problem, a standard machine configuration was defined, from which variations could be made, and differences in behaviour observed. This standard machine configuration may be divided into a software configuration and a hardware configuration. The standard software configuration defines the software used in a series of experiments in which different machine hardware configurations were simulated. In a similar manner, the standard hardware configuration was used for a number of tests in which several aspects of the software configuration were varied.

14.1.1 Software Configuration for Hardware Tests

The task chosen for tests of hardware variants was the linear filtering operation. This was selected because little inter-processor communication is required, and the performance of the task is not dependent on the behaviour of a run-time scheduler. This means that the results concerning hardware parameters are obtained from the execution of genuine program code, rather than the run-time scheduler, and so the problems of system software implementation may be separated from those of hardware design. In addition, the run-time of the linear filtering program is relatively short, compared with the other programs (about 30 minutes per single program execution). In this configuration, all compiler optimisations are enabled, other than the re-location of loop-invariant code, and the improved WAIT algorithm, described in chapter 12, is used.

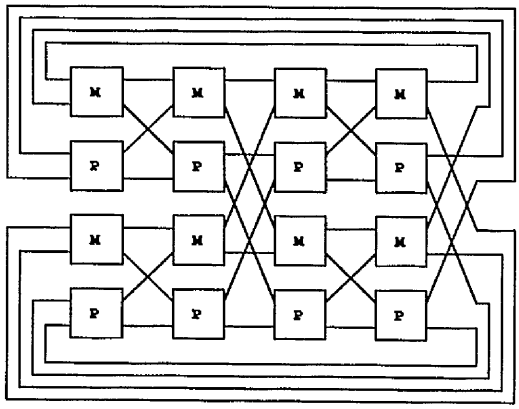


Fig. 14.1: Unit placements for the standard machine configuration.

14.1.2 Hardware Configuration for Software Tests

The hardware configuration which was selected for software tests is based on the 2-repeated partitioned indirect binary 2-tube. This particular n-tube network was chosen because it is capable of supporting up to sixteen functional units, which allows a configuration containing eight memory units and eight processors to be used. Eight processors was felt to be the minimum number which could give a realistic impression of the amount of processor interaction and the speedup characteristics which would be encountered in a real machine. The large number of simulation runs which were required meant that machine configurations larger than this could not be used for all simulations, although machines with up to 64 functional units were used in some individual simulation runs. The eight processors and eight memory units were connected to the interconnection network in the pattern shown in Fig. 14.1. Each piece of software under test was simulated using different numbers of processors, from one to eight, to produce a series of performance graphs. In the cases where less than eight processors were in use, these were attached to the lowest numbered network control units (using the numbering scheme described in chapter 9) which were consistent with the arrangement shown in Fig. 14.1.

In this standard configuration, the processing units have a 100ns cycle time and are equipped with a local program store. The memory units have interleaved addresses, and an access time of 200ns for read and write operations, and 500ns for increment-in-memory and decrement-in-memory operations. The interconnection network has a cycle time of 100ns, and uses the 'nearest-to-destination' arbitration

strategy, described previously in chapter 9, for packet-routing clashes. Packet hovering is not allowed.

14.2 Organisation of Results

Simulations were performed in a series of numbered runs. In each run, a specified machine configuration was used, and the machine was simulated using different numbers of processors from one, to the maximum possible in that configuration. All simulation runs are described in appendix 4, and samples of system configuration files and raw result files are given in appendices 5and 6respectively. From the result files, graphs of different parameters were plotted against the number of processors in the simulated machine, and these are compared and discussed in this chapter. Eight different forms of graph are used:

i) Total completion time

The total execution time for a program is the elapsed time from the start of the first instruction to the shut-down of the last processor. This gives an impression of the absolute speed of an algorithm or machine configuration.

ii) Mean completion time

The mean completion time of a program is the mean of the completion times, for each processor in the machine. This figure is always less than the total completion time, and does not show the extent to which the shut-down times of processors may be skewed. For this reason, mean completion times are only used where a large number of runs were required and, thus, short run-lengths had to be used. In such cases it is believed that the mean completion time more accurately predicts the expected performance of a machine working on a larger problem than the total completion time would do.

iii) Speedup factor

The speedup factor, for some particular number of processors, is defined to be the ratio of the total completion time of a program using that number of processors, to the total completion time for the same program, using only one processor in the same machine configuration. The graphs of speedup factors give a measure of the relative performance of algorithms, or machine configurations, as the number of processors increase. They are often easier to interpret than

completion-time graphs, since an ideal machine would exhibit a linear speedup factor, with a gradient of unity. These graphs may also give more of an impression of the likely performance of larger machines than those simulated, since the speedup graphs are easier to extrapolate than graphs of completion time. For these reasons, graphs of both completion time and speedup factor are presented for most program runs.

iv) Mean speedup factor

The mean speedup factor is a speedup factor calculated from mean completion times, rather than total completion times.

v) Hardware efficiency

The hardware efficiency was defined in chapter 11, as the ratio of the total time spent processing instructions or performing memory accesses, to the total completion time. The values used in the graphs are averaged over all processors in the machine, weighted according to the total completion time of each processor. This gives a mean figure for the time during which all processors are active.

vi) Number of packet-routing clashes

This is the total number of packet-routing clashes which occurred during the execution of a program.

vii) Number of memory-accessing clashes

This is the total number of memory-accessing clashes which occurred during the execution of a program. Any packet which is forced to repeat its request more than once is counted at every attempt to re-try its access.

viii) Scheduling overhead

This is the percentage of the total elapsed processing time for which the run-time scheduler was executed, rather than the task itself. The scheduling time includes the time taken to place jobs on, and remove them from, the job lists.

The results are divided into hardware tests, where the same software is used for each run, and software tests, where the hardware configuration is the same in each case.

14.3 Hardware Tests

A number of parameters of the simulated machine hardware were varied from those in the standard hardware configuration, and the programs defined by the standard software configuration were executed. These hardware tests include variants of interconnection networks and functional unit design, and may be grouped into:

i) comparison of ring and partitioned indirect binary n-tube networks;

ii) variation of partitioned indirect binary n-tube length;

iii) arbitration strategies for packet-routing clashes;

iv) speed of network control units;

v) functional unit placement patterns;

vi) effects of local instruction stores;

vii) address translation mechanisms.

These groups are discussed individually in the following sections.

14.3.1 Comparison of Ring and Binary n-Tube Based Machines

A number of simulations were performed to investigate the difference in performance of machines which contain the same numbers of functional units. These machines are based on the unidirectional ring network and the 2-repeated partitioned indirect binary 2-tube network. The results of these simulations are depicted in Figs. 14.2, 14.3 and 14.4, which show total completion times, speedup factors and hardware efficiency values respectively. It may be seen that, as may be expected, the machine using the simple ring network requires a significantly longer execution time. This may be attributed to a lower hardware efficiency which is, in turn, caused by high network latency. The increase in speedup factor, using the ring network, is approximately linear, but the rate of increase is less than that obtained using the the 2-tube. From this, it may be inferred that the relative performance of the ring network will not improve significantly for larger machines, and so the performance of the machines based on partitioned indirect binary n-tube networks machine will be better than that of ring-based machines.

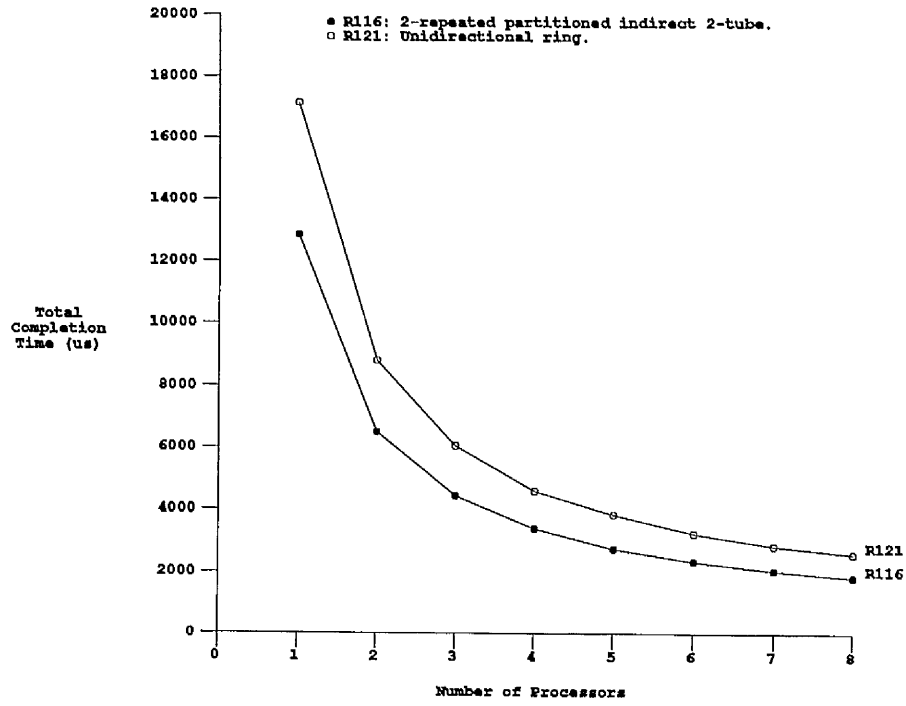


Fig. 14.2: Total completion times for the ring and 2-repeated partitioned 2-tube networks.

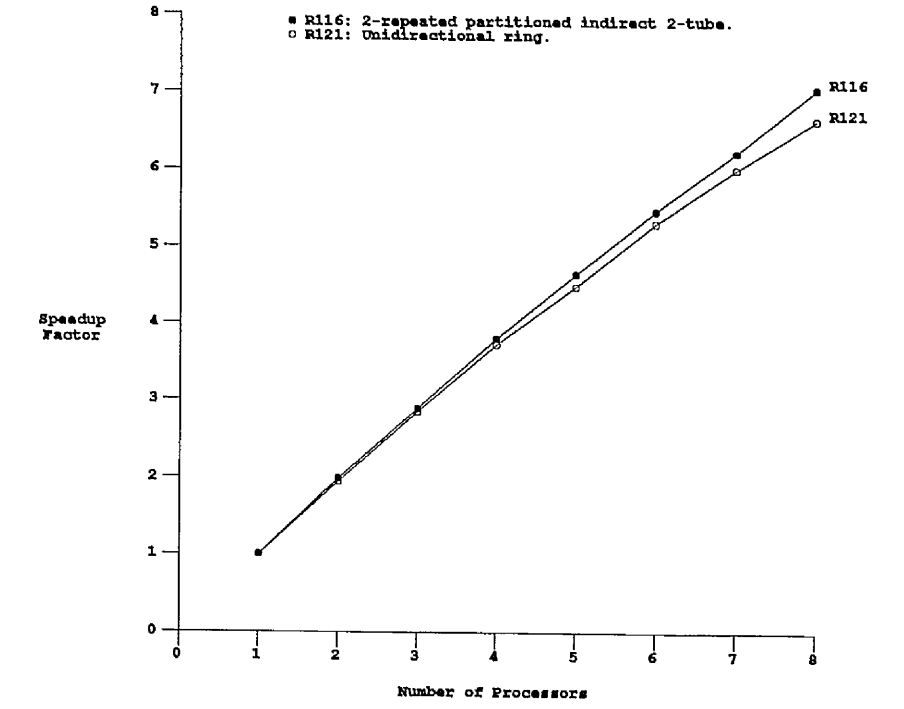


Fig. 14.3: Speedup factors for the ring and 2-repeated partitioned 2-tube networks.

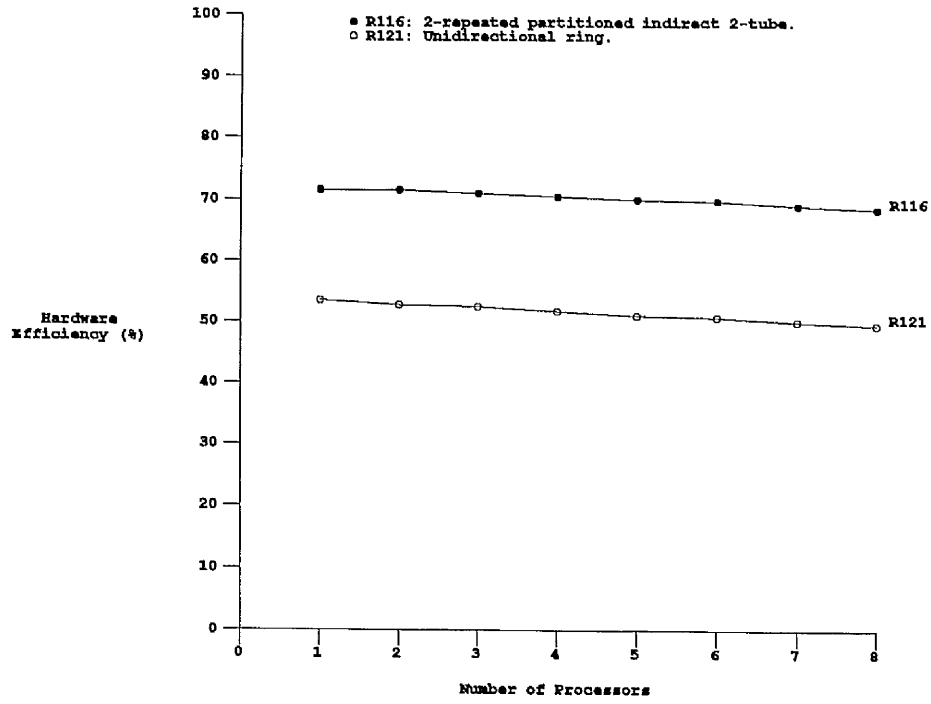


Fig. 14.4: Hardware efficiencies for the ring and 2-repeated partitioned 2-tube networks.

14.3.2 Variation of Partitioned Indirect Binary n-Tube Length

A number of simulations were performed to investigate the effect of tube length in machines based on the partitioned indirect binary n-tube. Figs. 14.5 and 14.6 show mean completion times and mean speedup factors for the linear-filtering task executed on machines based on the partitioned indirect binary 2-cube and on partitioned indirect binary 2-tubes of up to eight repeated sections. These values are shown in Figs. 14.7 and 14.8 for the partitioned indirect binary 3-cube, 4-cube and 2-repeated 3-tube. All of the simulated configurations show good speedup, and the levelling off of the speedup factor which may be seen in Figs. 14.6 and 14.8 is mainly due to the start-up and shut-down overheads, which may take up to 25% of the total execution time. This happens because the simulated execution time becomes very short when 32 processors are used, as the image is divided into very small sections. The hardware efficiency for these machine configurations is plotted against the number of processors in Figs. 14.9 and 14.10. It may be seen that, as might be expected, the hardware efficiency of the tube networks is slightly less than that of the cube networks but, perhaps surprisingly, the hardware efficiency increases

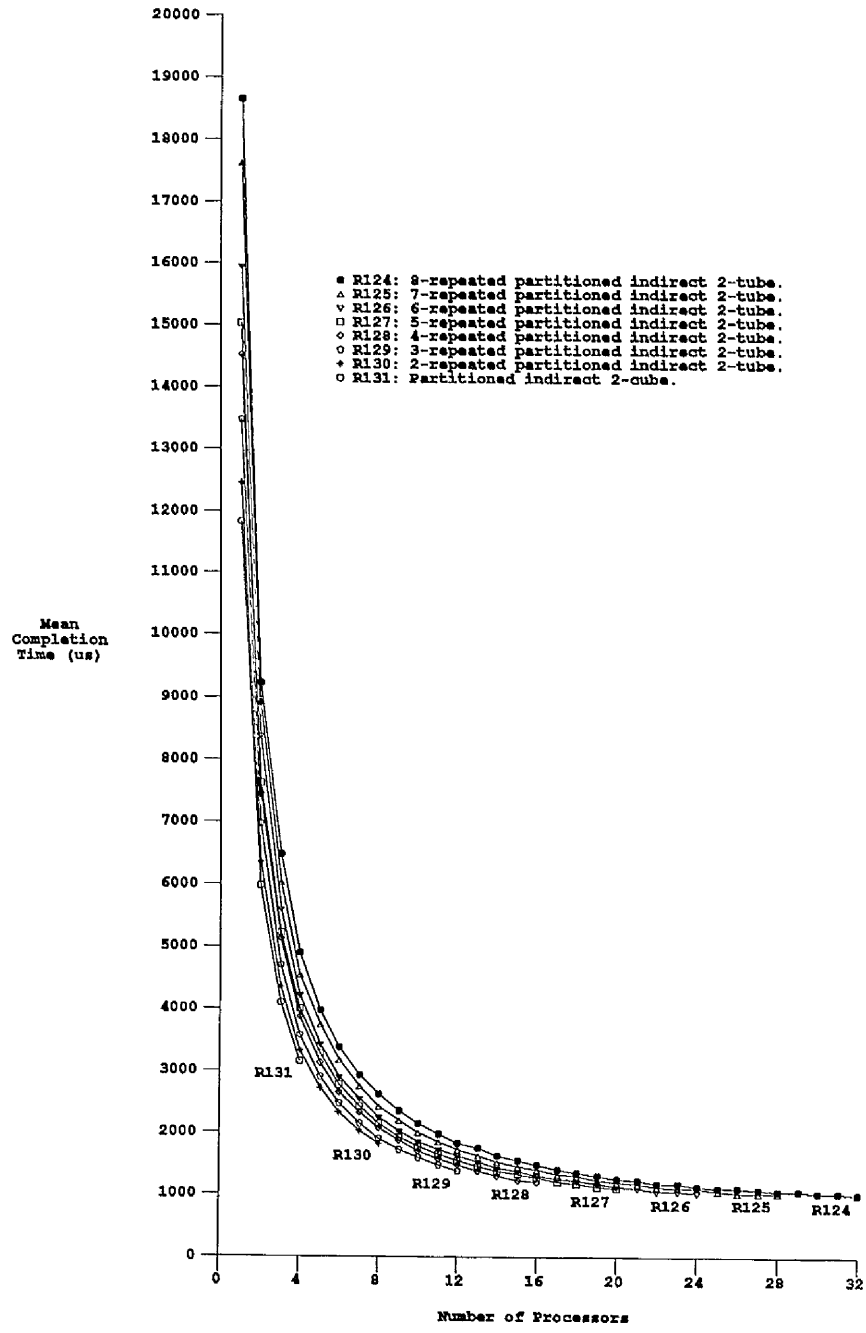


Fig. 14.5: Mean completion times for differing-length tube networks.

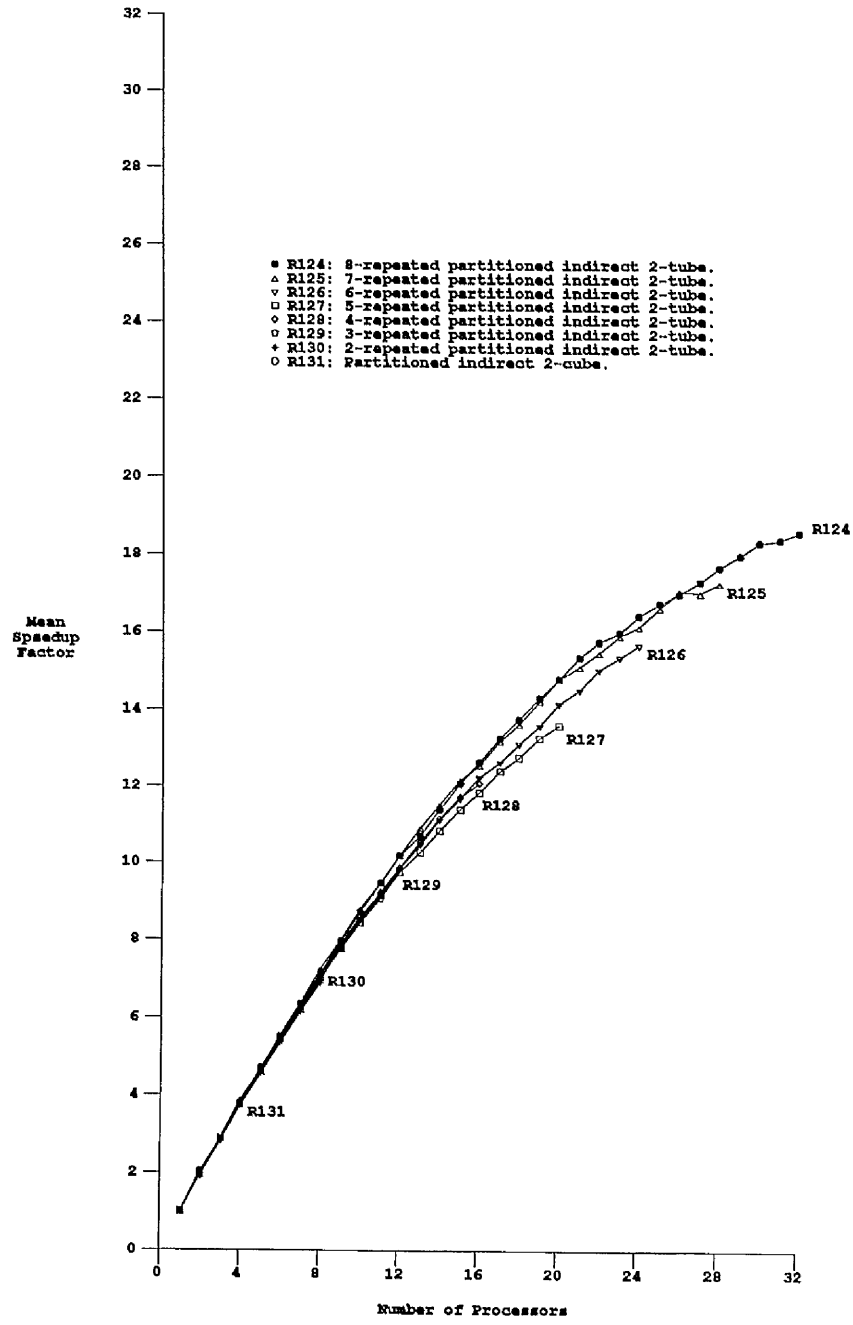


Fig. 14.6: Mean speedup factors for differing-length tube networks.

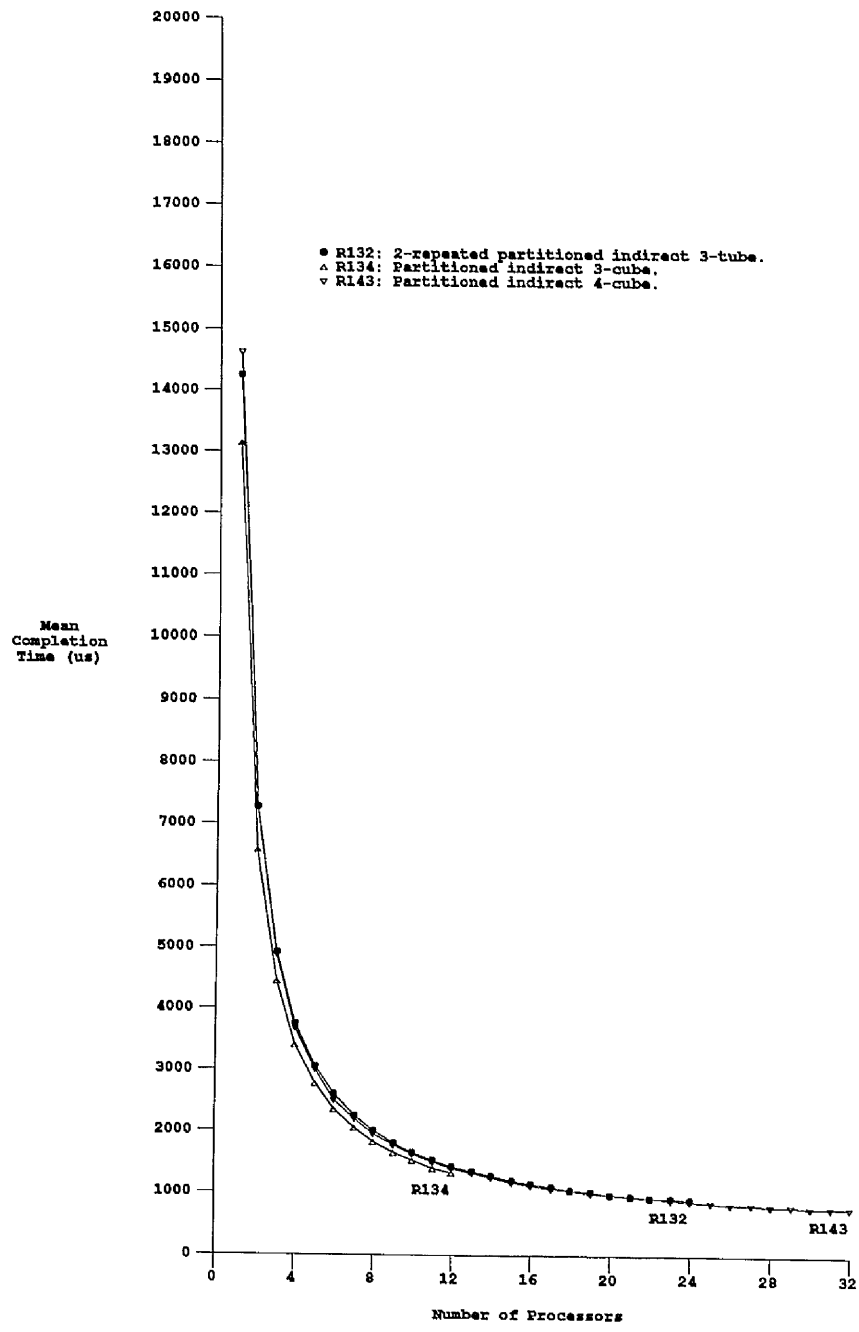


Fig. 14.7: Mean completion times for differing-length tube and cube networks.

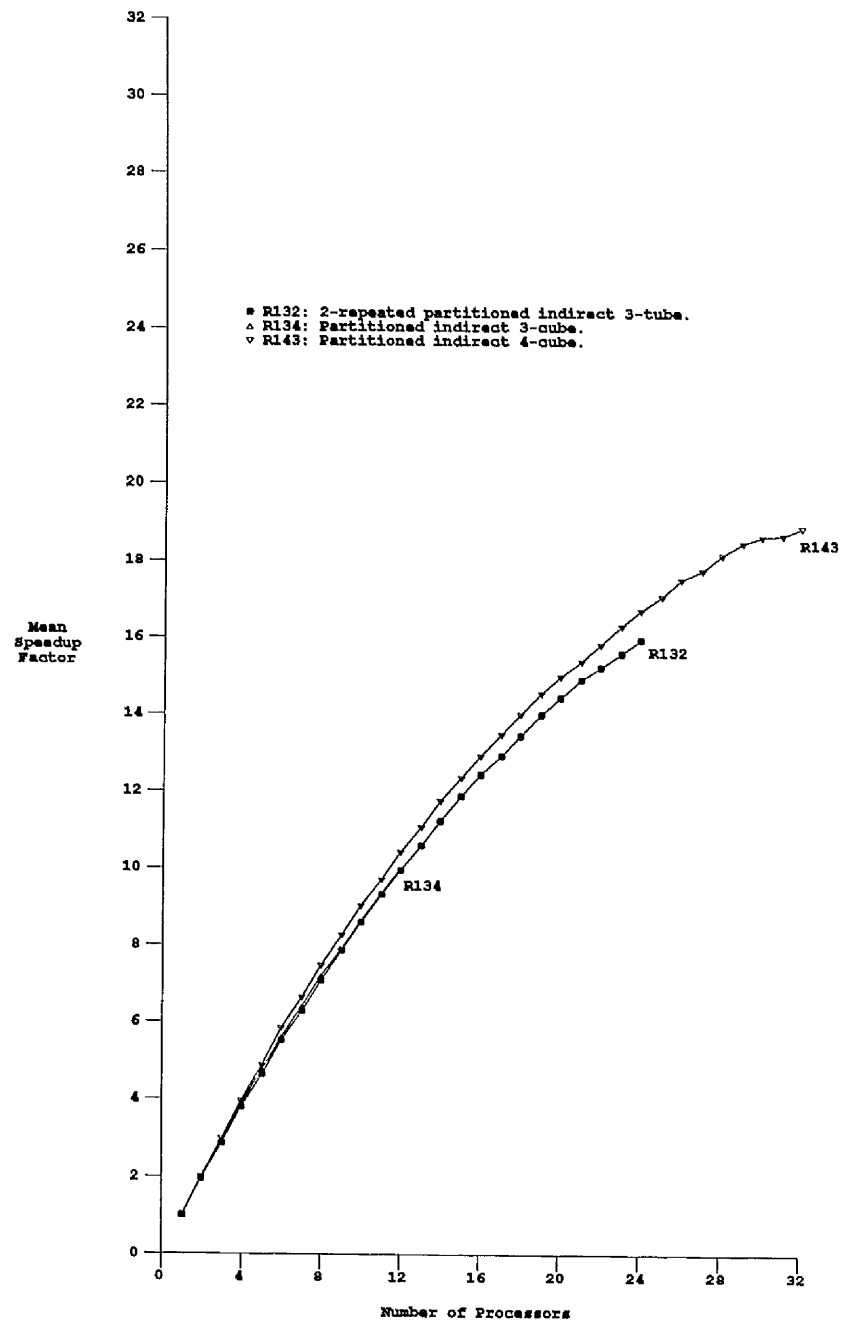


Fig. 14.8: Mean speedup factors for differing-length tube and cube networks.

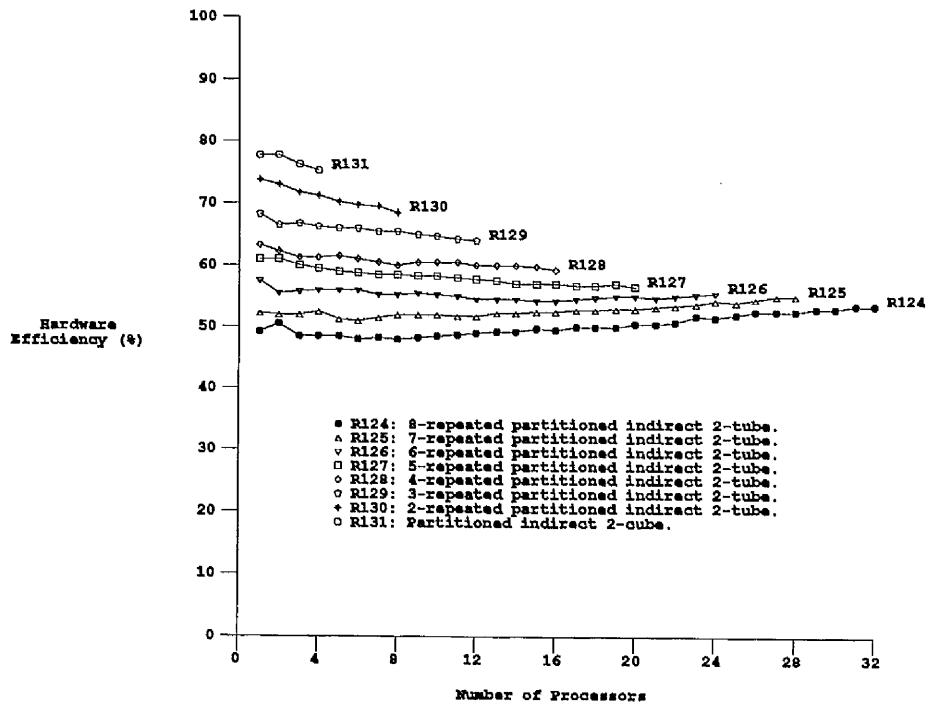


Fig. 14.9: Hardware efficiencies for linear filtering on differing-length tube networks.

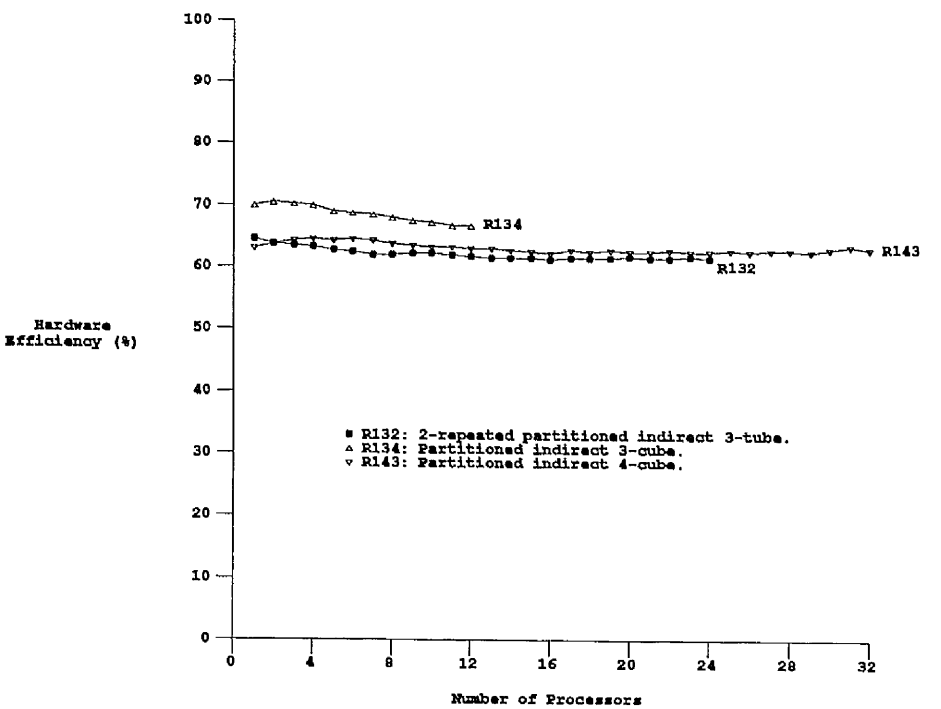


Fig. 14.10: Hardware efficiencies for linear filtering on differing tube and cube networks.

slightly with tube length. This effect is caused by the pattern of memory accesses, in tubes with differing populations. In long tubes with few processors, these processors were concentrated at one end of the tube. Memory access patterns, even with interleaved store addresses, result in some memory units receiving many more accesses than others. These correspond to the fixed addresses of loop counters, and other frequently-used variables. If, as is the case in this simulation, the processors are located very close to the appropriate memory locations, it may require two full circuits of the network to make a memory request, and return a reply packet. These effects of unit placement patterns are discussed later in this chapter.

The results obtained for the partitioned indirect binary 4-cube and the 8-repeated partitioned indirect binary 2-tube network are worthy of particular examination, since these are relatively large networks which support equal numbers of functional units. Some results of low-level and high-level simulation of these machines are shown in Table 14.1, in which the low-level simulation results relate to fully-occupied (32 processor) machines. The theoretical maximum latency for the 8-repeated 2-tube network (derived in appendix 1) may be seen to be over twice that for the 4-cube (also derived in appendix 1), and the results from high-level simulation predict that the mean latency for the tube network should be about twice that of the cube network. The low-level simulation shows, however, that despite the higher actual network occupancy level and the higher number of packet-routing clashes in the tube simulation, the ratio of the mean latencies is lower than expected. This may be explained by the fact that if, in a cube network, a packet-routing clash occurs, one packet must always travel around the entire network again to reach its destination. In a tube network, the probability of a clash resulting in a complete circumnavigation is lower, and so this reduces the observed mean latency.

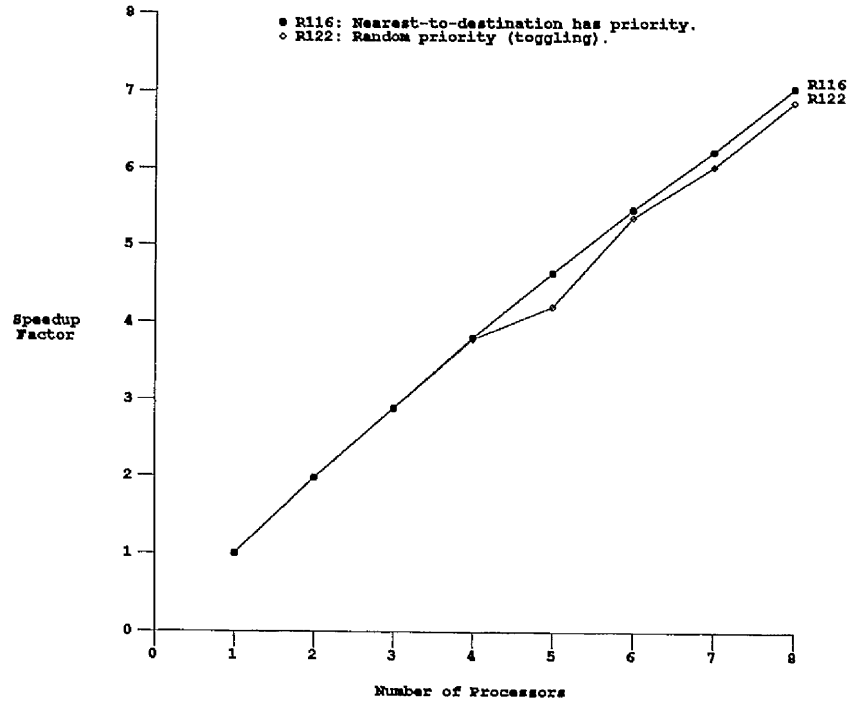


Fig. 14.11: Speedup factors for differing packet-routing arbitration schemes.

Table 14.1
A Comparison of Partitioned Indirect n-Cube and n-Tube Machine Topologies

	R124	R143
Network Topology	8-Repeated Tube	4-Cube
Theoretical L_{max}	18	8
High-level simulation		
L_{mean} at 0% network occupancy	8.750	4.563
L_{mean} at 10% network occupancy	10.426	5.386
L_{mean} at 20% network occupancy	12.389	6.467
Low-level simulation		
Network occupancy	17.5	13.4
L_{mean}	10.0	5.9
Packet-routing clashes	7755	3088

14.3.3 Arbitration Strategies for Packet-Routing Clashes

Two different packet-routing clash arbitration strategies were simulated, and the speedup factors for these are shown in Fig. 14.11. The two strategies are the 'nearest-to-destination' priority scheme, and a pseudo-random scheme, implemented by a simple toggling action, between exchange and straight connection, at each packet-routing clash. The speedup factors may be compared directly, since both configurations have exactly the same total execution time for the single processor case, as no packet-routing clashes can occur in a single processor system. The hardware efficiency and numbers of packet-routing clashes are shown in Figs. 14.12 and 14.13. The 'nearest-to-destination' priority system may be seen to have a slight advantage over the pseudo-random scheme, and it is likely that this advantage would be much greater in cases where more packet-routing clashes take place. It should be noted that the toggling strategy is not usable in practice, as it may permit a hardware lock-up where two or more packets repeatedly deflect each other from their destinations. This strategy is implemented only as an approximation to the random arbitration strategy, against which the preferred strategy may be compared.

14.3.4 Speed of Network Control Units

It was indicated in chapters 8 and 9 that the speed of the network control units was thought to be critical for high machine efficiency, since the network latency time is directly proportional to this. Figs. 14.14, 14.15, and 14.16 show the effects of varying the time allowed for each network cycle. It may be seen that this does, indeed, have a profound effect, but the results are somewhat unexpected. A surprising anomaly may be seen in Fig. 14.14, where the simulated machine with a network-cycle time of 200ns (R119) is seen to be faster than one with a network-cycle time of 150ns (R118). This occurs because the memory-access time is held constant, at a figure of 200ns. When the network-cycle time is 150ns, a memory access requires two network cycles, but if this is increased to 200ns, a memory access may be made in a single network cycle, and so the overall machine speed is increased. Such anomalies occur only when the network-cycle time is high in comparison with the memory-access time, and it is clear, from the completion times for these configurations, that a short network-cycle time is essential. A similar effect may be seen in the hardware-efficiency graphs (Fig. 14.16), where the grouping of the curves is

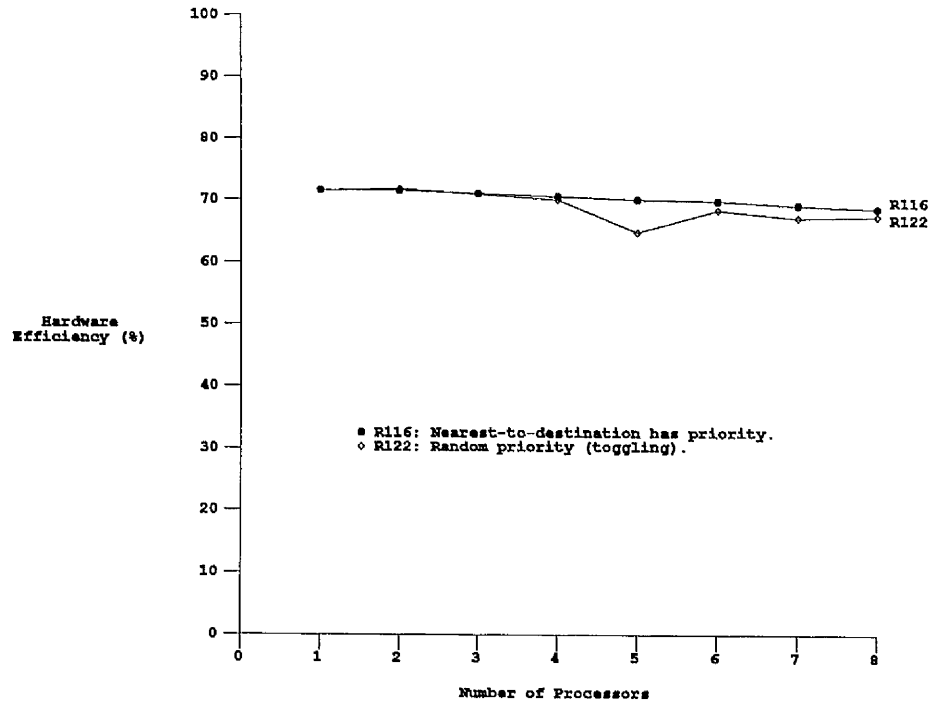


Fig. 14.12: Hardware efficiencies for differing packet-routing arbitration schemes.

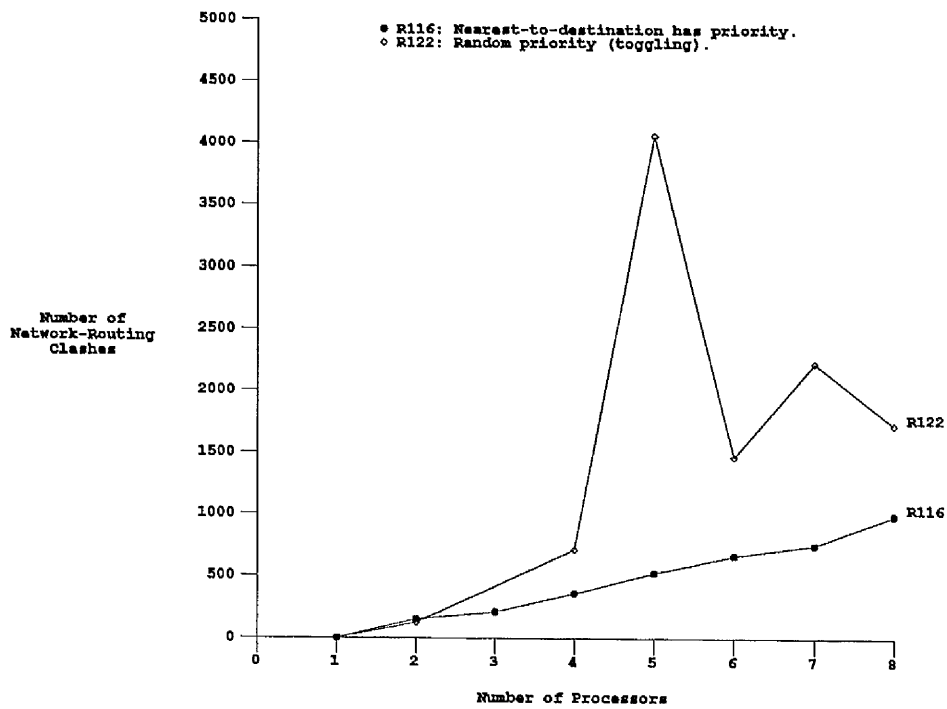


Fig. 14.13: Numbers of packet-routing clashes for different packet-routing arbitration schemes.

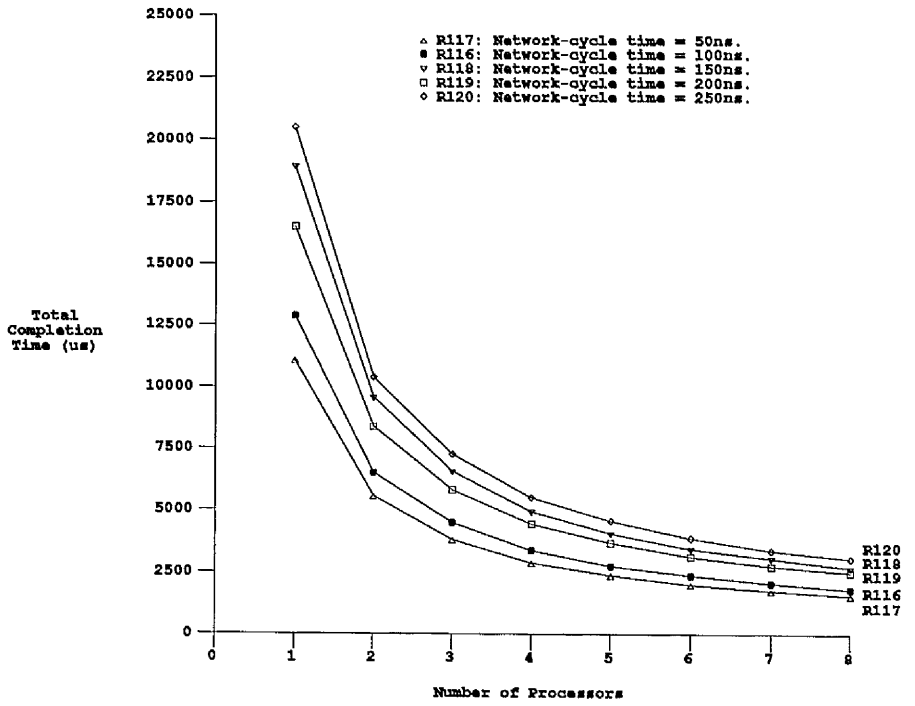


Fig. 14.14: Total completion times for varying network-cycle times.

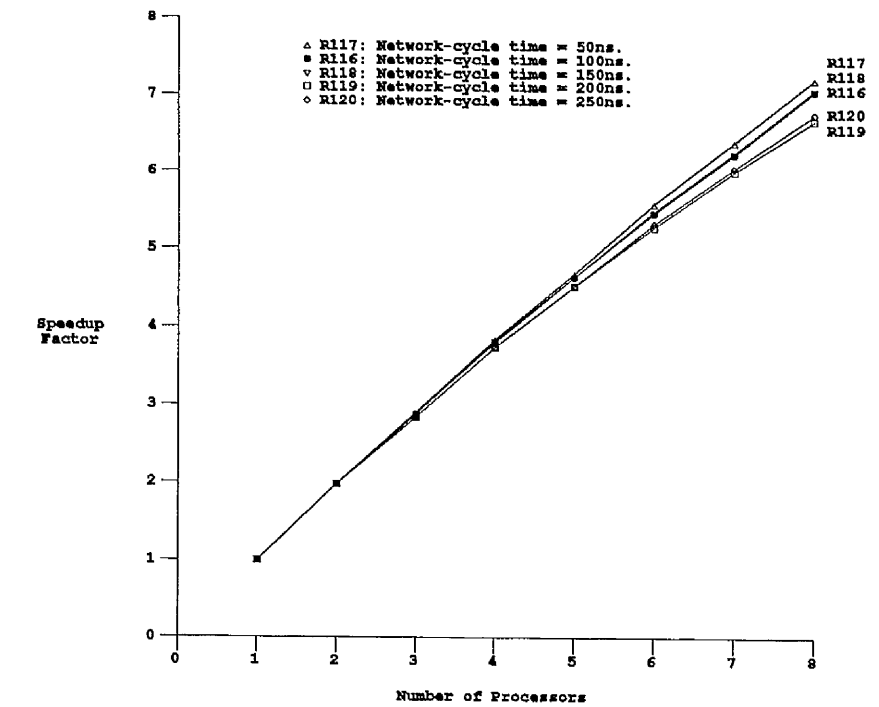


Fig. 14.15: Speedup factors for varying network-cycle times.

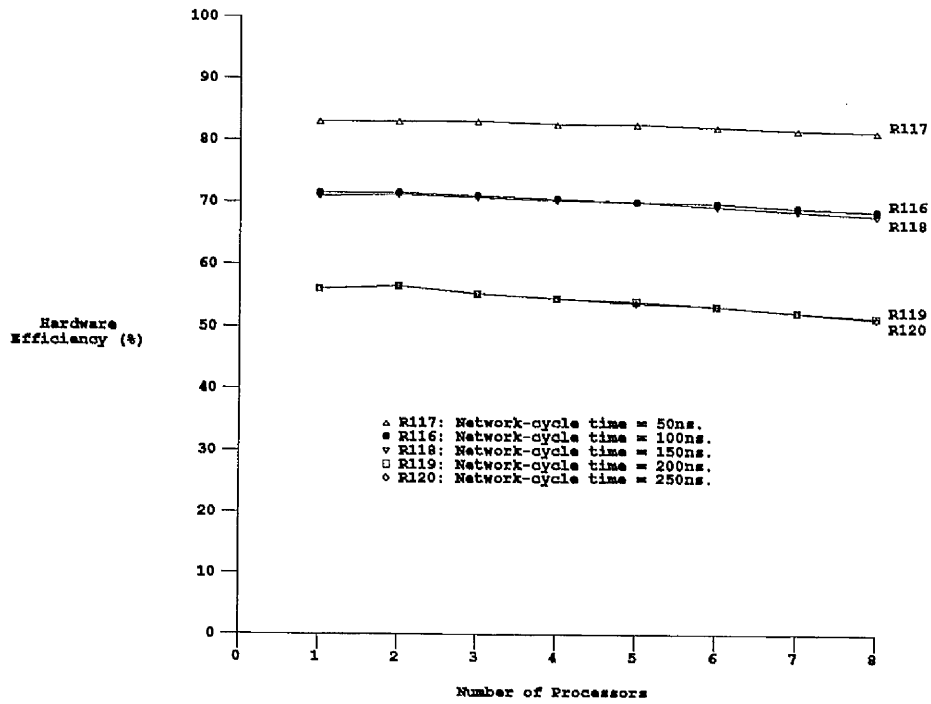


Fig. 14.16: Hardware efficiencies for varying network-cycle times.

related to the number of processor cycles (100ns) which correspond to one network cycle. The speedup factor (Fig. 14.15) may be seen to be marginally affected by variations in the network-cycle time and this occurs because, for cases with longer network cycles, the network occupancy level was found to rise slightly, causing more packet interaction and, thus, higher mean latency.

14.3.5 Functional Unit Placement Patterns

To investigate the effects of the patterns of placement of processors and memory units on machine performance, several different placement patterns were simulated. These patterns are illustrated in Fig. 14.17. The standard machine configuration (used for simulation run R116), has alternate rings occupied entirely by processors or memory units. This is referred to as a layered horizontal split. The central horizontal split (R147) is similar, but has two rings occupied by processors, then two rings occupied by memory units. Since there is no distinction between rings, other than the numbering of their functional units, and the network is symmetric in cross section, these two configurations may be expected to, and do in practice, show similar results. The layered vertical split (R149) is configured

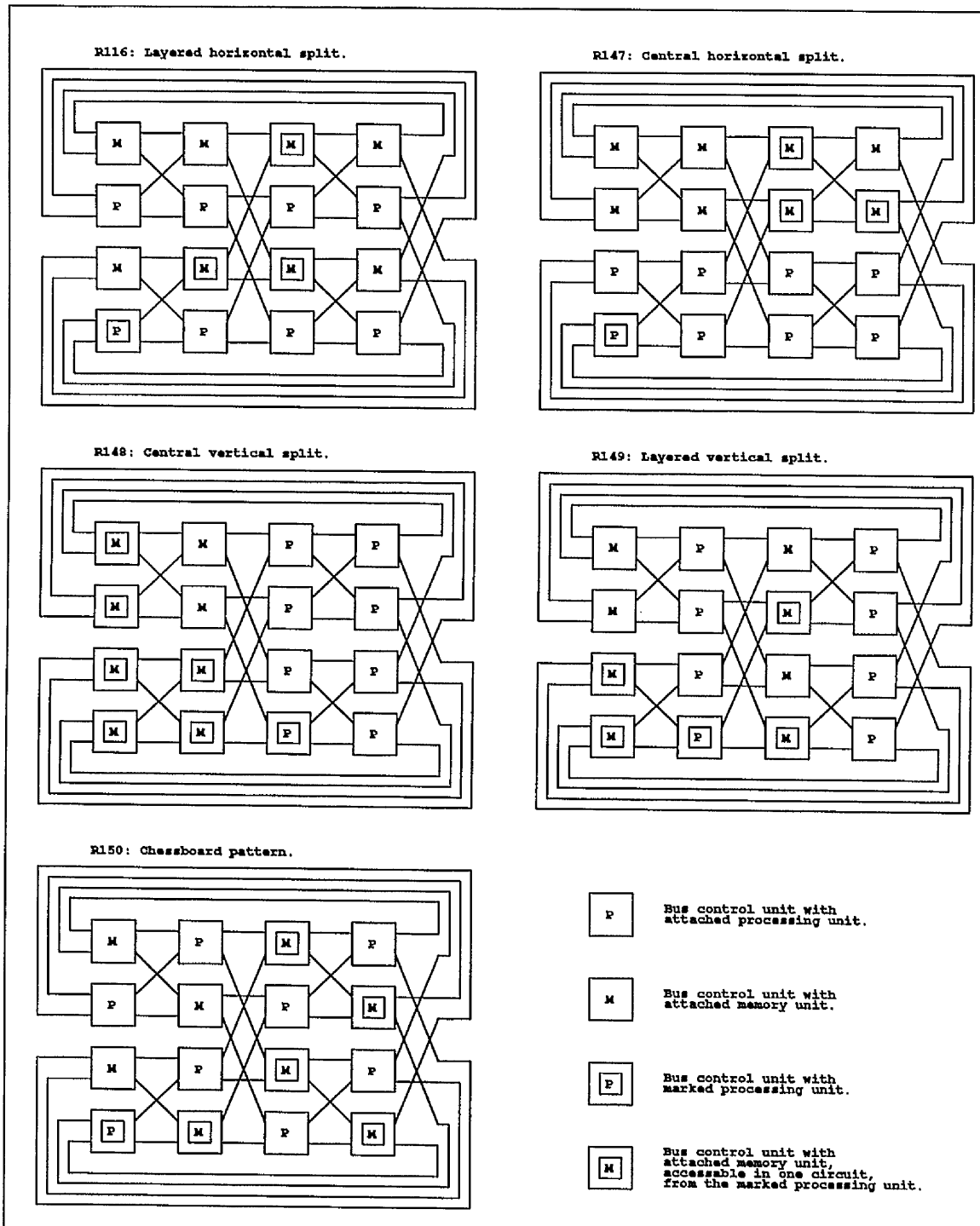


Fig. 14.17: Functional unit placement patterns.

to have alternate ranks completely occupied by processors or memory units, and the central vertical split (R149) has two ranks of memory units, then two ranks of processing units. A 'chessboard' pattern (R150) was also used, in which two processors and two memory units occupy each ring, skewed relative to each other, so that two processors and two memory units appear on each rank.

It may be seen, from the total completion times shown in Fig. 14.18, that the placement pattern does have a slight effect on machine performance, and this is caused by differences in the hardware efficiency, which is shown in Fig. 14.19. This may be explained by considering the number of memory units, in each arrangement, which a processor may access in such a way that a reply packet is generated and routed so that the packets travel, in total, only once around the network. Such memory units are indicated in Fig. 14.17. For memory accesses directed to other memory units, the outward and return packets must perform a total of two complete circuits around the network. The number of memory units which may be accessed in one circuit affects the overall mean latency and, thus, the hardware efficiency of the machine. For the layered horizontal split (R116), and the central horizontal split (R147), only three memory units may be reached in this way. The layered vertical split (R149) permits four units may be reached; the chessboard pattern (R150) allows five; and the central vertical split (R148), six. The graphs of completion time and hardware efficiency may be seen to reflect this order.

A secondary cause of these variations in completion time may be the numbers of packet-routing clashes (shown in Fig. 14.20) which occur in the various configurations. The two configurations which have entire rings composed of a single type of unit (R116 and R147) show the worst results, in this respect. This may be due to the routing strategy used, which causes packets to move to the ring on which their destination lies as soon as possible, and remain there until they are accepted by their destination unit. This means that, when packets accumulate on the network due to memory conflicts, packets which are destined for other memory units will have a higher chance of interacting with the queued packets, since these queued packets circulate along rings which are fully occupied by memory units only.

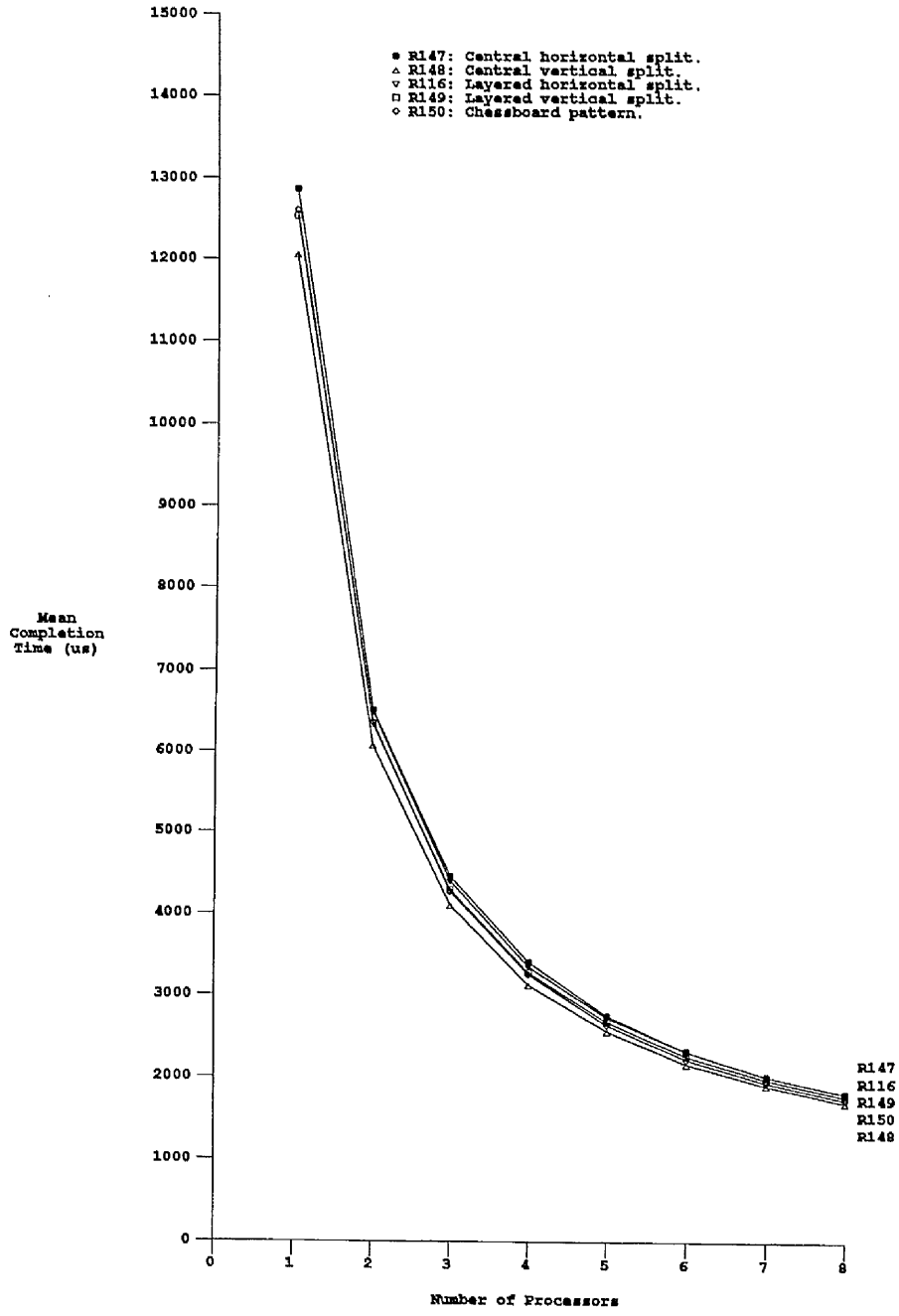


Fig. 14.18: Mean completion times for differing functional unit placement patterns.

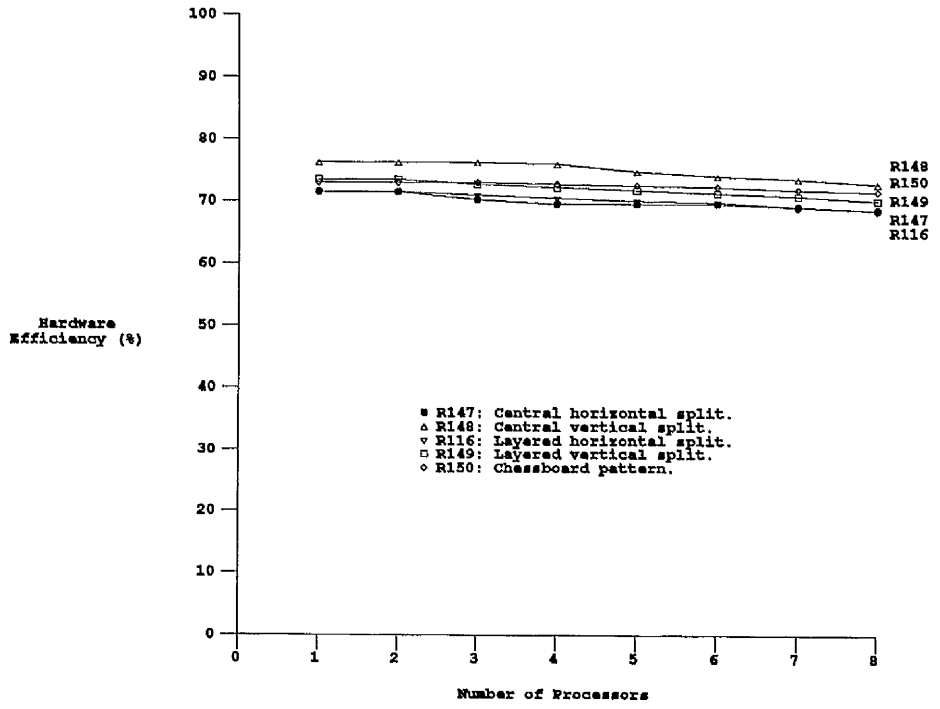


Fig. 14.19: Hardware efficiencies for differing functional unit placement patterns.

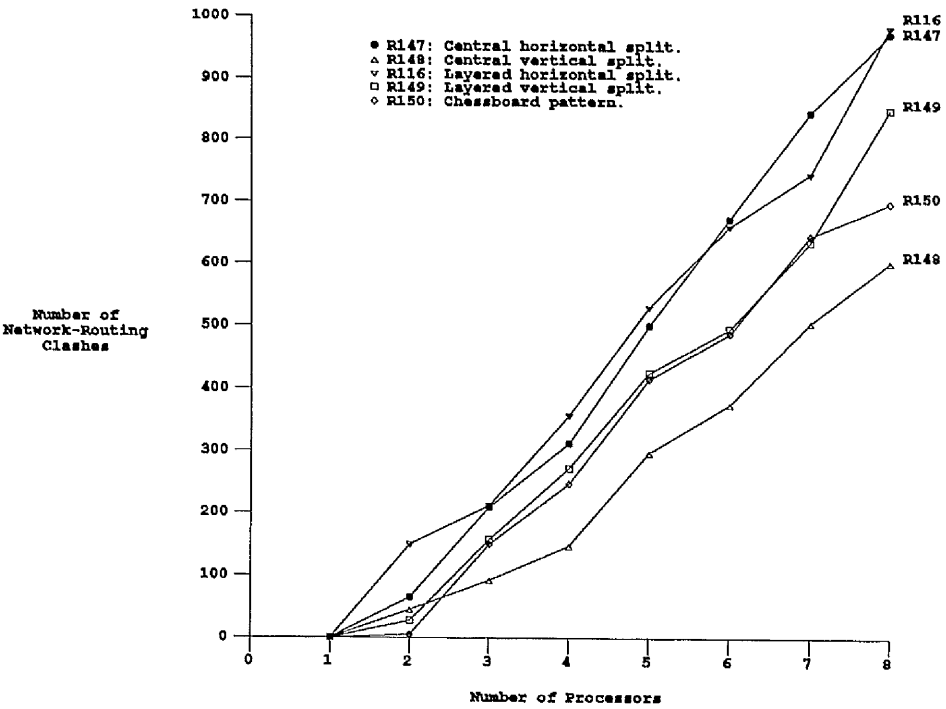


Fig. 14.20: Numbers of packet-routing clashes for differing functional unit placement patterns.

14.3.6 Effects of Local Instruction Stores

In the standard machine configuration, each processor is equipped with a local instruction store, which is loaded with a copy of the program to be executed. This obviates the need to perform a shared-memory access, via the interconnection network, for every instruction. A machine without a local program store was simulated, and the results are shown in Figs. 14.21 to 14.24. The total execution time (Fig. 14.21) is increased by a factor of greater than two, and the hardware efficiency (Fig. 14.22) falls to a very low level. This may be attributed to the greatly increased number of memory accesses, and the consequential increase in the numbers of packet-routing and memory-accessing clashes (Figs. 14.23 and 14.24).

14.3.7 Address Translation Mechanisms

The 2-repeated partitioned indirect binary 2-tube machine, defined by the standard machine configuration, uses an interleaved memory addressing scheme. Simulations were performed to compare this system with one using untranslated memory addresses. Fig. 14.25 shows the total completion time for these simulations. It may be seen that, surprisingly, when few processors are used, the system without any form of address translation performed better than the system with interleaved memory addresses. This effect is due to the placement of the processors in a favourable position in the network with respect to frequently-used memory units (as described earlier in this chapter) and may be explained with reference to the latency figures shown in Tables 14.2 and 14.3. For the case where a single processor operated on a non-interleaved memory (R146), the memory units which received frequent accesses were, by chance, those which could be accessed, and a reply packet returned, with a total latency of one circuit of the network. This gives a low mean latency figure compared with the interleaved case (R116) where the memory units are accessed in a more evenly distributed manner. This effect disappears when the machine is fully populated with eight processors, since the machine is then symmetric and favourable positions are balanced by less favourable ones. The graph of speedup factors (Fig. 14.26) shows that the rate of increase of speedup factor is slightly lower without address translation, and this trend is expected to be continued to larger machines. It may be seen from Fig. 14.27 that, as the number of processors in the system is increased, the numbers of memory-accessing clashes rises, as may be expected, if no address translation is used. A corresponding rise in the number of packet-routing clashes

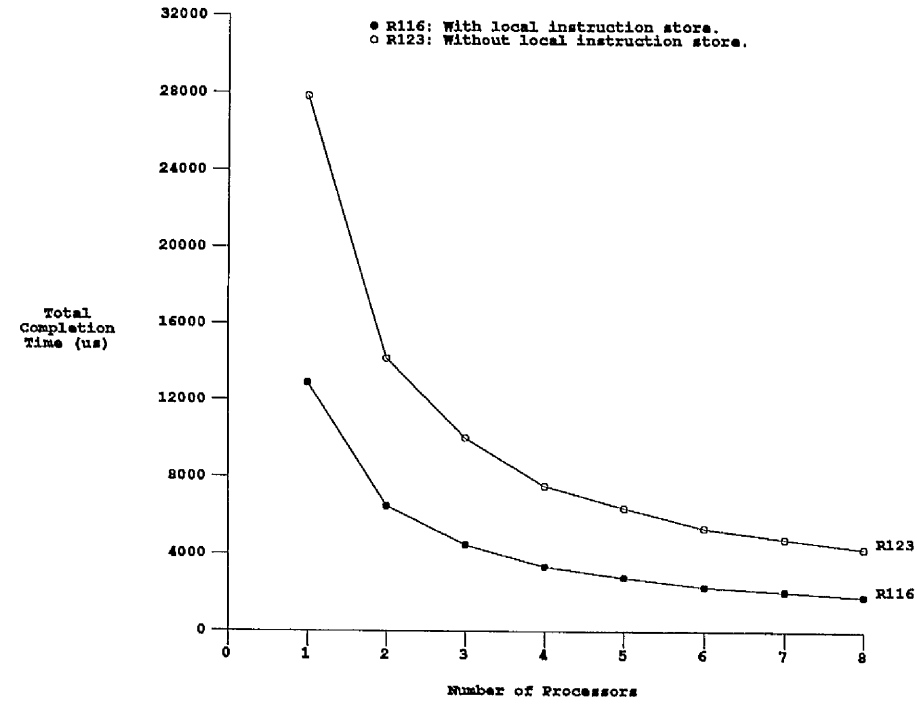


Fig. 14.21: Total completion times with, and without, local program stores.

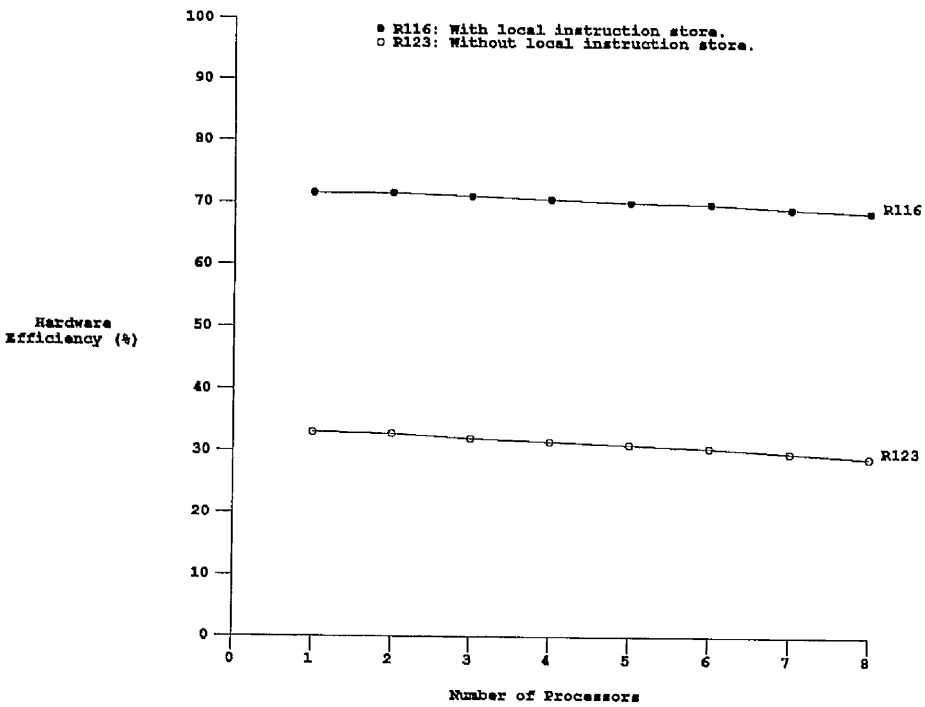


Fig. 14.22: Hardware efficiencies with, and without, local program stores.

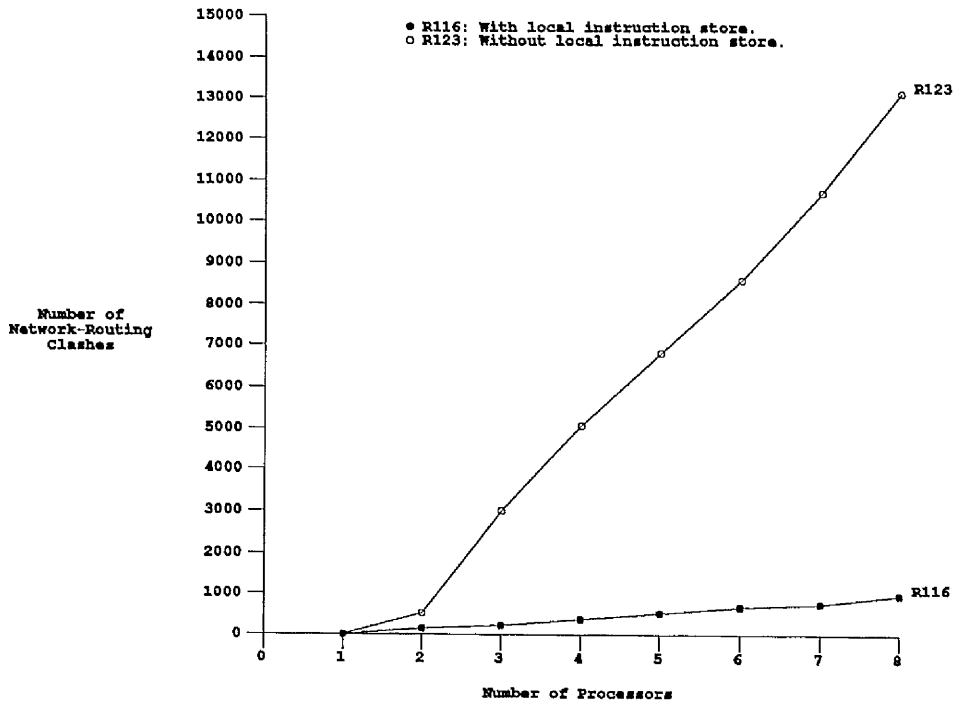


Fig. 14.23: Packet-routing clashes with, and without, local program stores.

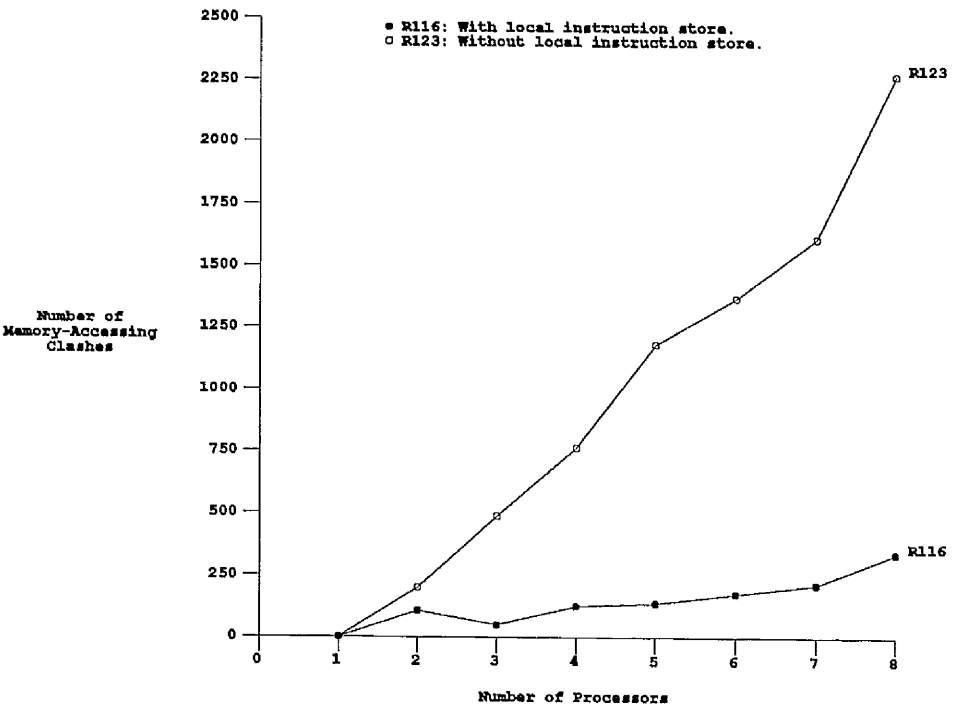


Fig. 14.24: Memory-accessing clashes with, and without, local program stores.

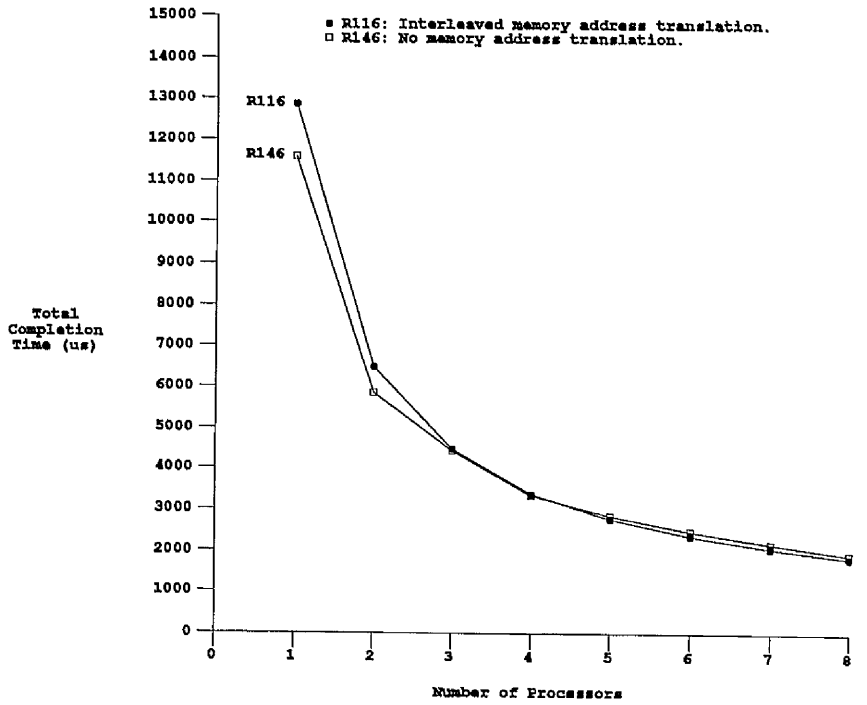


Fig. 14.25: Total completion times with, and without, memory address translation.

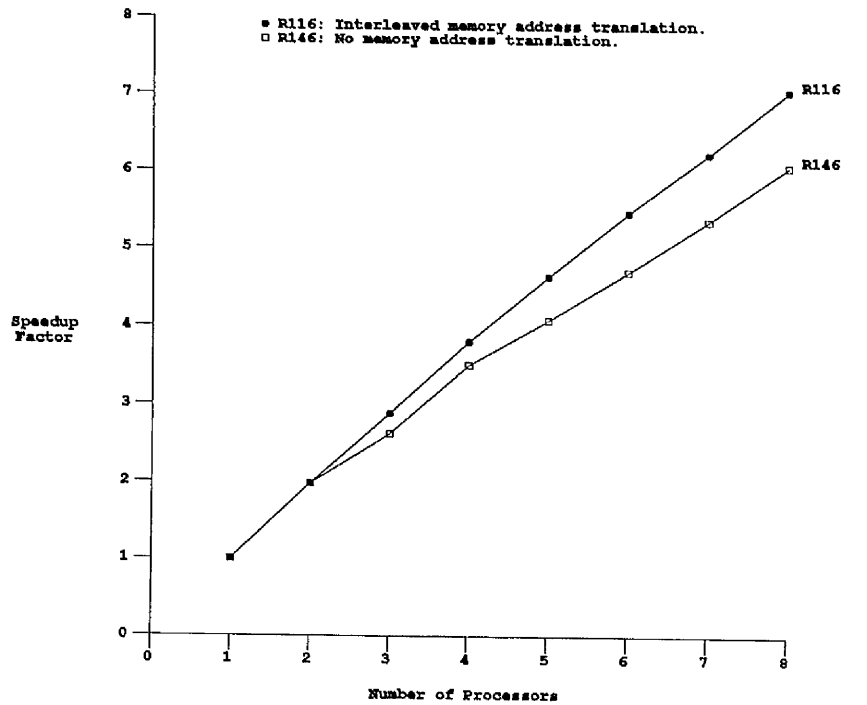


Fig. 14.26: Speedup factors with, and without, memory address translation.

Table 14.2
Network Activity for One Processor,
with Interleaved (R116), and
Non-Interleaved (R146), Store Configurations.

Unit Number	Unit Type	Interleaved		Non-Interleaved	
		Transfers	Latency	Transfers	Latency
0	M	832	4.0	0	0.0
1	P	4649	3.3	4649	2.0
2	M	836	4.0	0	0.0
3	–	0	0.0	0	0.0
4	M	321	3.0	0	0.0
5	–	0	0.0	0	0.0
6	M	834	3.0	0	0.0
7	–	0	0.0	0	0.0
8	M	325	2.0	3081	2.0
9	–	0	0.0	0	0.0
10	M	839	2.0	1559	2.0
11	–	0	0.0	0	0.0
12	M	334	5.0	0	0.0
13	–	0	0.0	0	0.0
14	M	328	5.0	9	5.0
15	–	0	0.0	0	0.0
Total		9298	–	9298	–
Mean		–	3.4	–	2.0

Table 14.3
Network Activity for Eight Processors,
with Interleaved (R116), and
Non-Interleaved (R146), Store Configurations.

Unit Number	Unit Type	Interleaved		Non-Interleaved	
		Transfers	Latency	Transfers	Latency
0	M	866	3.2	0	0.0
1	P	631	3.7	631	3.3
2	M	856	3.1	0	0.0
3	P	617	3.4	619	3.1
4	M	362	3.5	0	0.0
5	P	617	4.1	617	4.8
6	M	856	3.5	0	0.0
7	P	617	4.1	619	5.0
8	M	379	3.1	3184	3.0
9	P	619	3.5	619	3.8
10	M	859	3.1	428	2.2
11	P	619	3.5	619	5.5
12	M	395	3.6	852	2.5
13	P	619	4.0	619	4.3
14	M	385	3.7	498	4.1
15	P	619	4.0	619	4.3
Total		9916	–	9924	–
Mean		–	3.5	–	3.6

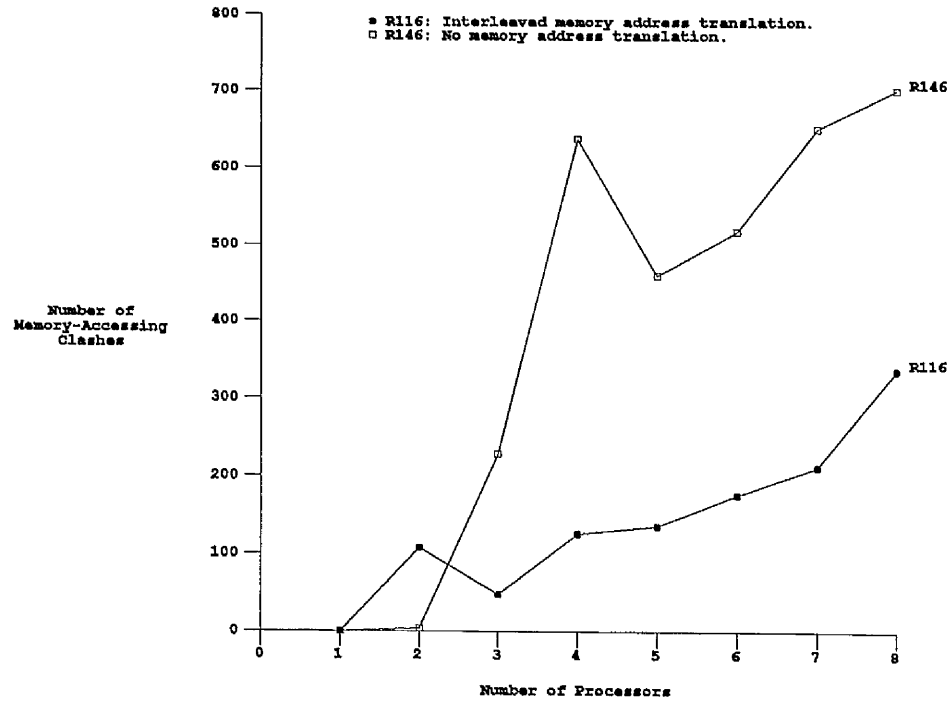


Fig. 14.27: Numbers of memory-accessing clashes with, and without, memory address translation.

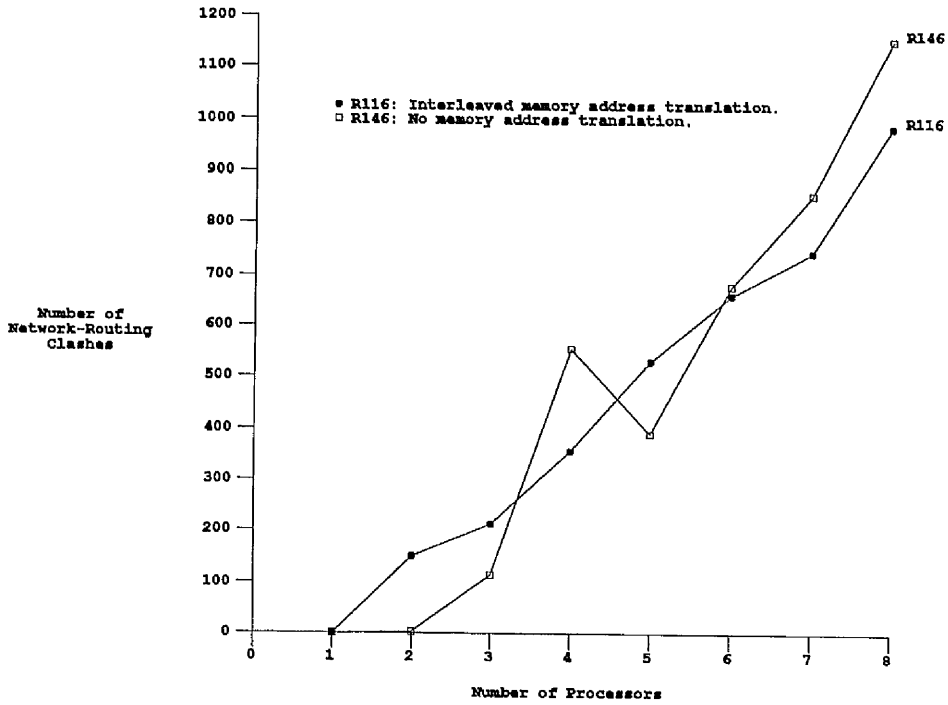


Fig. 14.28: Numbers of packet-routing clashes with, and without, memory address translation.

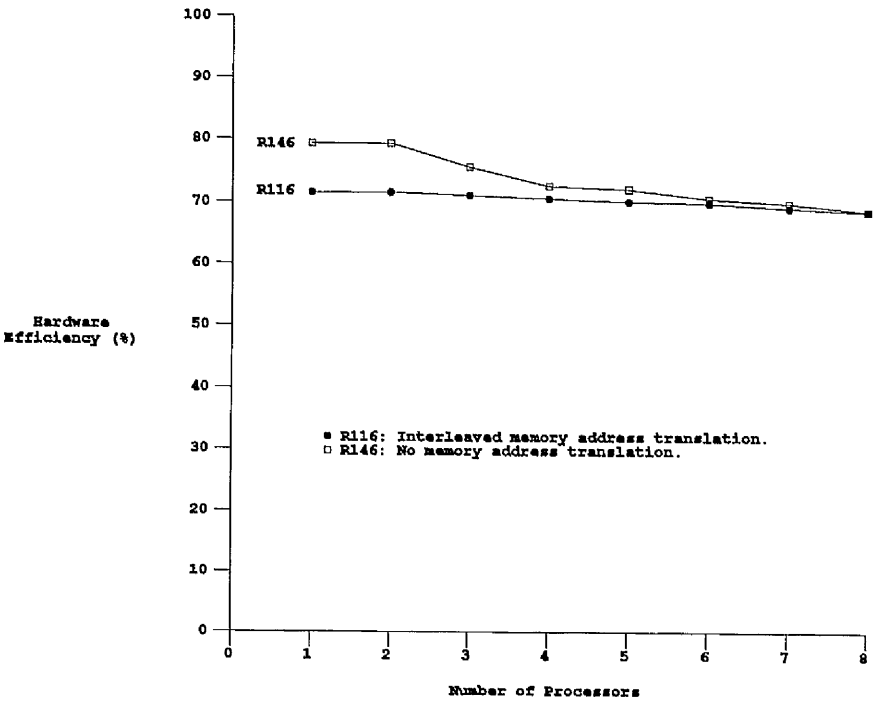


Fig. 14.29: Hardware efficiencies with, and without, memory address translation.

occurs (Fig. 14.28), and the resulting increase in mean latency causes the hardware efficiency to fall (Fig. 14.29).

14.4 Software Tests

A series of tests were performed on several aspects of the software configuration, using the standard hardware configuration described earlier. This machine configuration is based on the 2-repeated partitioned indirect binary 2-tube network, and contains eight processing units and eight memory units. The tests which were performed include comparisons of the machine performance obtained for the execution of differing tasks, run-time schedulers and inter-process synchronisation procedures.

14.4.1 Simulation of Differing Tasks

The three different tasks described in chapter 13 (linear filtering, region filling, and integer sorting) were simulated, and a graph of the speedup factor was plotted for each (Fig. 14.30). The region-filling

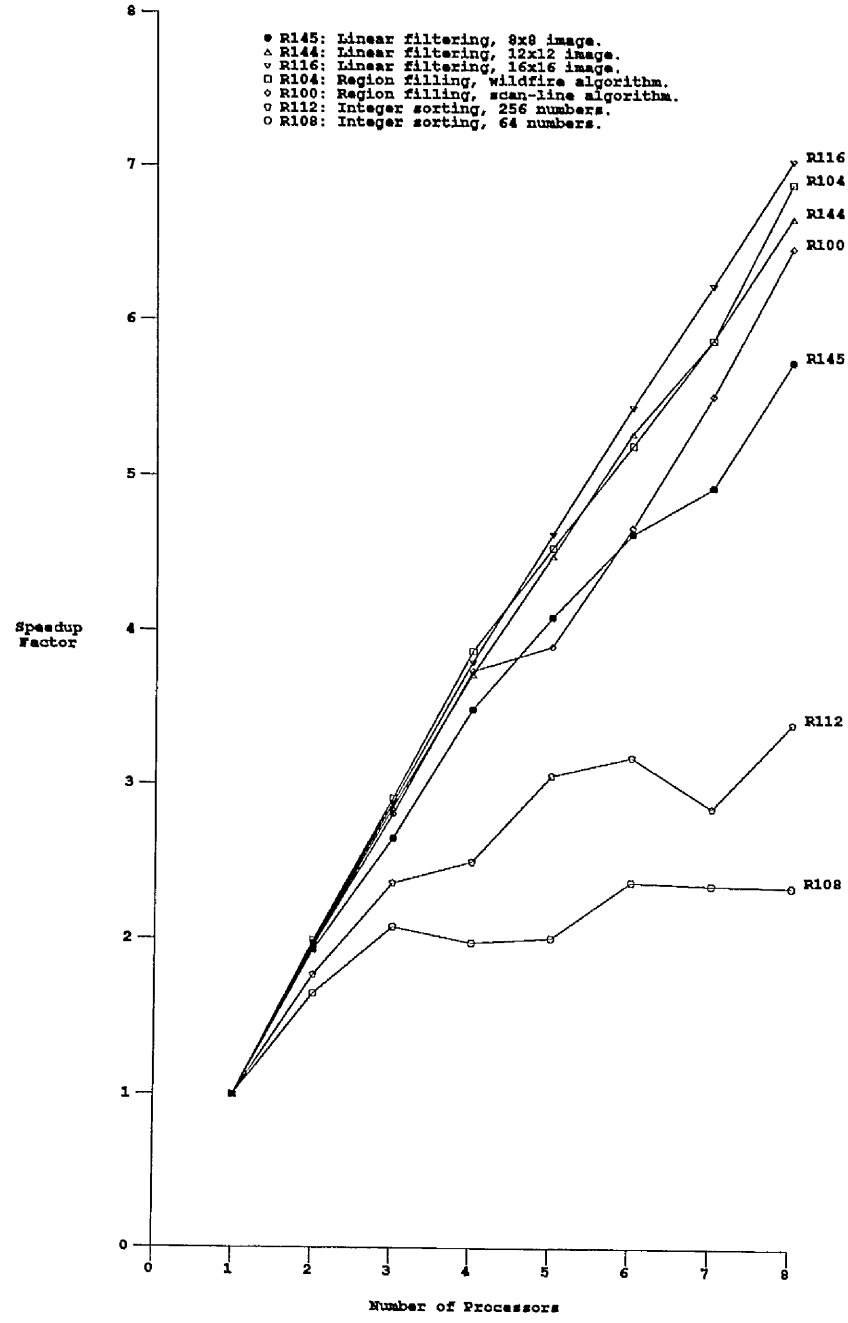


Fig. 14.30: Speedup factors for differing tasks.

and integer-sorting tasks were simulated using a pre-waiting run-time scheduler with private job lists and dynamic job re-allocation. This scheduler is compared with others, later in this chapter. The linear-filtering task may be seen to exhibit a near-linear speedup, for the cases using 16×16 (R116) and 12×12 images (R144). Linear filtering on the 8×8 image (R145) also shows good speedup, but since the image is much smaller than the other cases, the near-constant overheads involved in start-up and shut-down become more significant as the number of processors increases and the total completion time decreases. The two algorithms used for the region-filling tasks, the wildfire algorithm (R104) and the scan-line algorithm (R100), also exhibit good speedup factors. The speedup factor for the wildfire algorithm is higher, but this is an inherently slower algorithm.

The integer-sorting task does not show good speedup. The version which sorts 64 numbers (R108) exhibits a maximum speedup factor of less than three, and the 256-number version (R112) performs only slightly better. The reason for this poor performance is that the quicksort algorithm builds up parallelism slowly, from a sequential start. Initially, only one list of numbers exists, and so only one processor may be active. When this list has been partitioned, a second processor may be employed to sort one of the two new lists. After these are divided, four lists exist, and this process of division continues until all processor are occupied. The low speedup factor arises because the first partitioning of the list is the most time-consuming. If all lists are split evenly, the first division will take one half of the overall time taken to sort the list, assuming a sufficient number of processors are available to sort all available lists in parallel. Consequently, this algorithm suffers from a lack of parallelism for most of its execution, and the maximum number of processors which may be usefully employed depends on the length of the list to be sorted.

The hardware efficiency of the machine is shown in Fig. 14.31, for each of the different simulated tasks. All give similar values, but the efficiency for the sorting task is slightly higher than the others. This is because the lack of available parallelism causes most processors to be idle, and a large amount processor time is spent in the WAIT procedure, which exhibits a high hardware efficiency since it generates few shared-memory accesses.

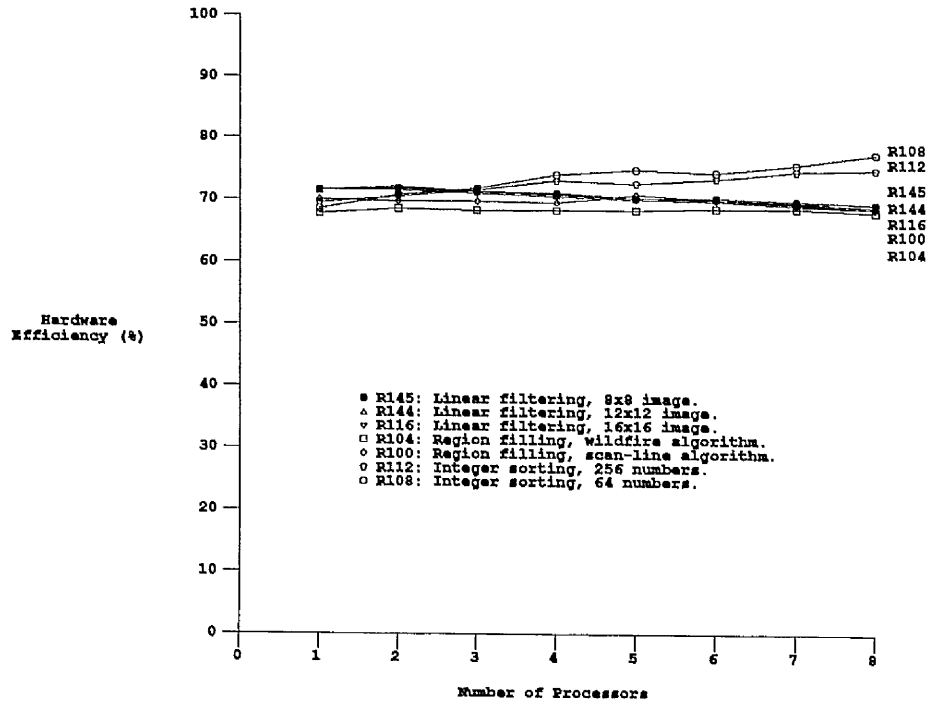


Fig. 14.31: Hardware efficiencies for differing tasks.

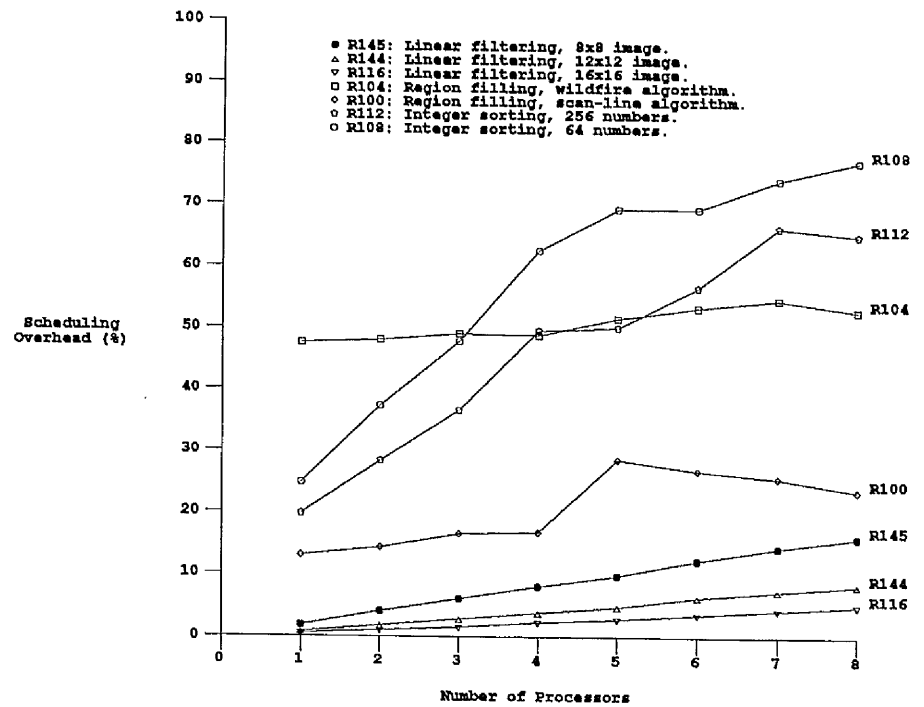


Fig. 14.32: Scheduling overheads, for differing tasks.

Fig. 14.32 shows the percentage of elapsed time which may be attributed to scheduling and synchronisation overheads, for each of the tasks under examination. The quoted overhead value includes time spent placing items on, and removing them from, private and global job lists. It may be seen that the quicksort algorithm (R108 and R112) has a low scheduling overhead for small numbers of processors, but this increases rapidly for larger numbers of processors, as more are left idle. For the linear-filtering task (R116, R144 and R145), all processors are actively engaged in the filtering process for most of the time, and the overheads involved are concerned with system start-up and shut-down overheads. These become more significant for the simulations which use very small images. The scan-line algorithm (R100) shows a quite low scheduling overhead, but since the wildfire algorithm (R104) breaks the task to be performed into very small pieces, its scheduling overheads are higher.

14.4.2 Run-Time Job Scheduling Algorithms

The wildfire algorithm and scan-line algorithm were simulated using a number of different run-time scheduling algorithms. These are listed in appendix 3, and include pre-waiting and non-pre-waiting schedulers with, and without, global job lists. Fig. 14.33 shows the total completion time for these programs, and Fig. 14.34 shows the speedup factors. Very little difference may be seen in the total completion times for the scan-line algorithm using private-list schedulers with, and without, pre-wait (R100 and R101). The execution of the scan-line algorithm using global-list schedulers (R102 and R103) may be seen to be marginally slower than with private-list schedulers (R100 and R101). The difference here is not large, since the job list is accessed relatively infrequently in the scan-line algorithm. The wildfire algorithms may be seen to be considerably slower than the corresponding scan-line algorithms, and the differences between the global-list (R106 and R107) and private-list (R104 and R105) versions of the wildfire program are pronounced. This may be attributed to the saturation of the global job list with access requests, and cases were observed where a processor was unable, for long periods of time, to place a new job on the global list, due to repeated access attempts by idle processors. In the global-list version, a slight improvement may be seen when the non-pre-waiting scheduler is used. This is because the pre-waiting system was designed to cope with the situation where idle processors prevented others from placing new jobs on the global list. Priority is effectively given to processes which are placing jobs onto the job list, since they do not enter the pre-wait state to do this. Using the wildfire algorithm,

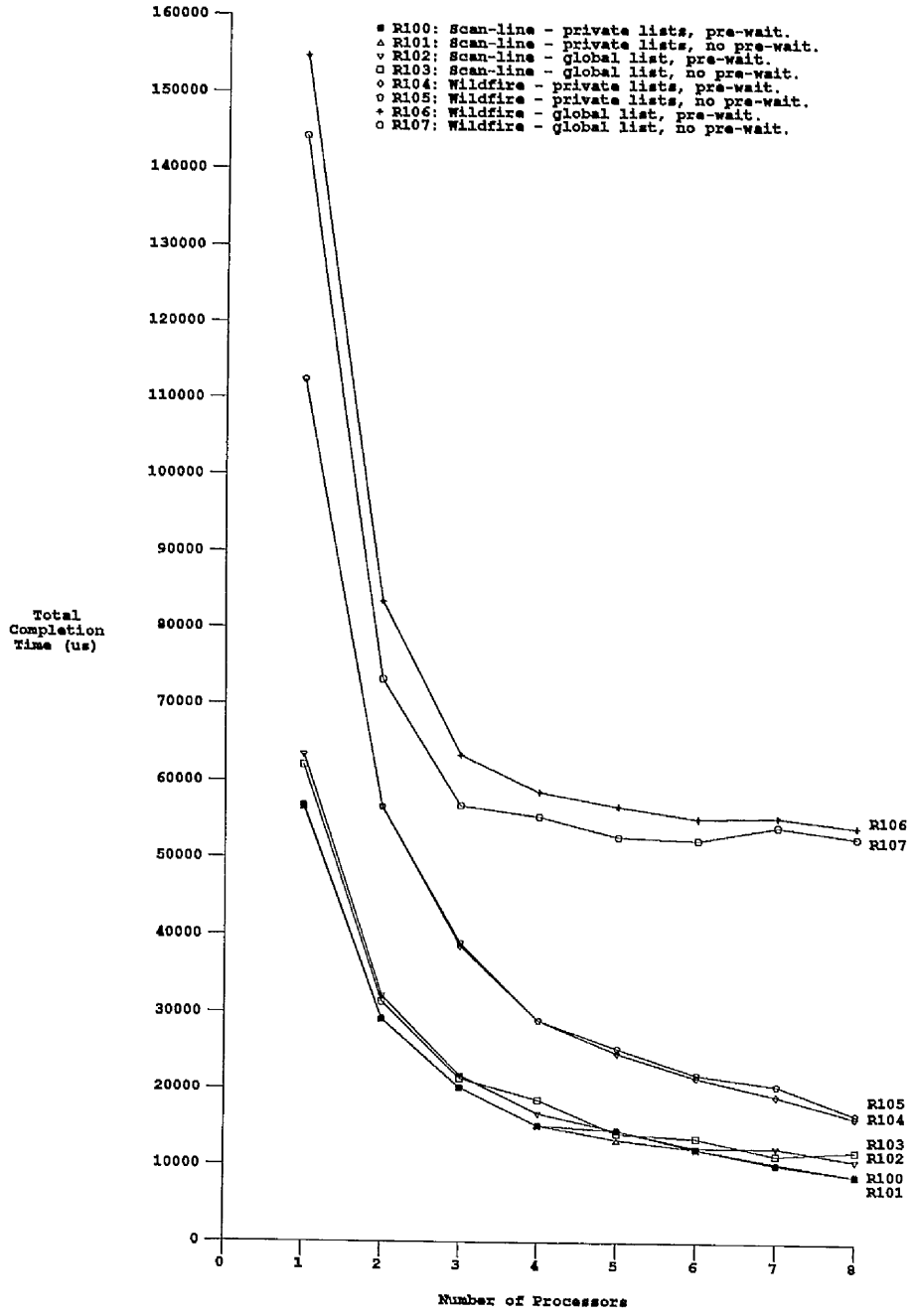


Fig. 14.33: Total completion times for the region-filling task, using various schedulers.

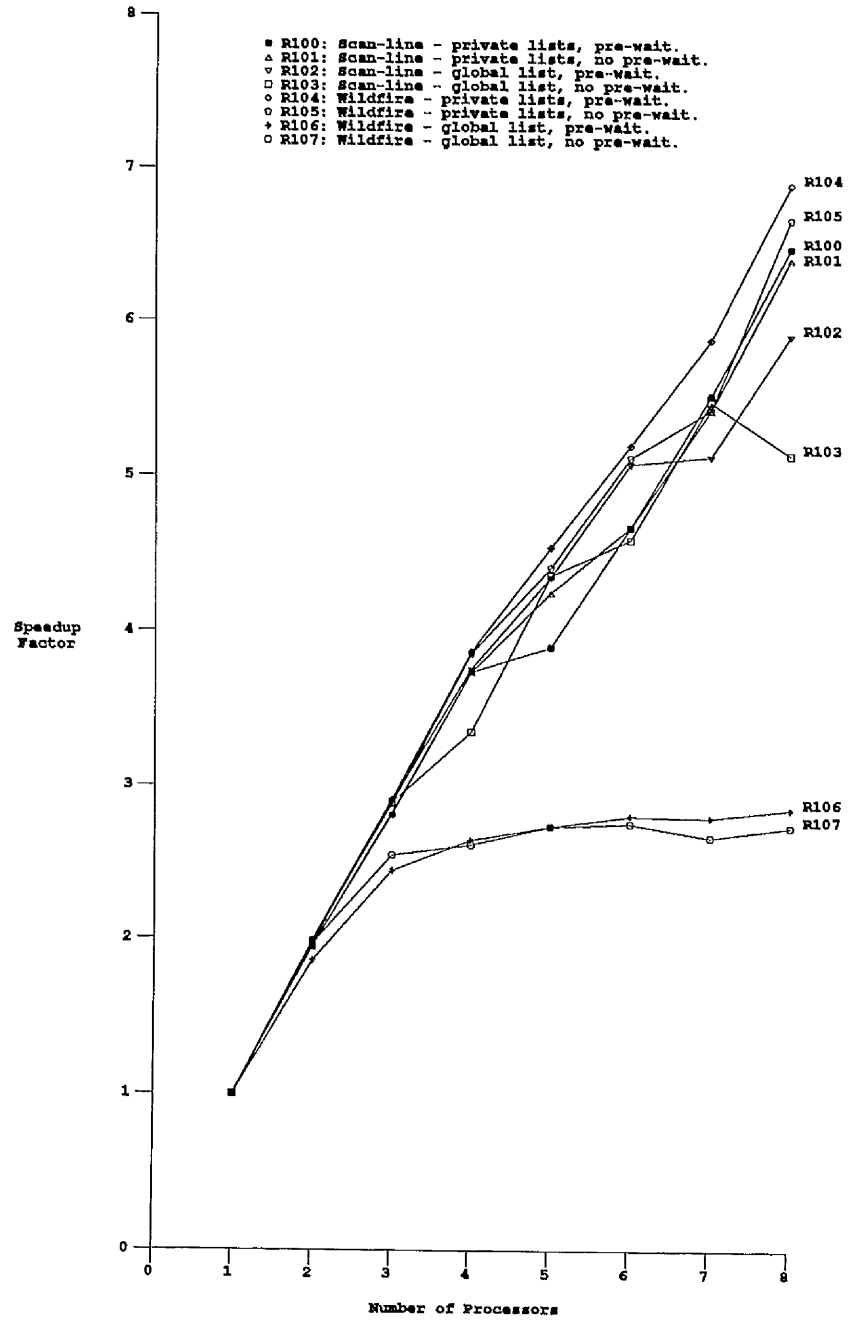


Fig. 14.34: Speedup factors for the region-filling task, using various schedulers.

however, the large amount of available parallelism ensures that few processors become idle, and the scheduler is unable to prevent the saturation of access to the job list caused, in this case, by processors attempting to place new seed points on it.

The speedup graph (Fig. 14.34) shows that all of the private-list algorithms exhibit good speedup, but the global-list ones do not perform quite as well. The best speedup is obtained by the wildfire algorithm using private-list schedulers (R104 and R105) but the wildfire algorithm is inherently slow, in absolute execution time. The wildfire algorithm, with global-list schedulers (R106 and R107) may be seen to encounter a speedup factor limit of less than three, due to global-list saturation, and it should be remembered that this is, in any case, an inherently slower algorithm.

Total completion times and speedup factors are shown in Figs. 14.35 and 14.36, for the quicksort task using the same selection of schedulers. In this case, the schedulers which use a pre-wait mechanism (R108, R110, R112, and R114) show slightly better performance than those which do not (unlike the results obtained for the region-filling task). This is because the pre-wait mechanism prevents the many idle processors from saturating the job-lists with requests when no jobs are available, and so the active processors have free access to the list, whenever new jobs are generated. Again, private-list schedulers (R108, R109, R112, and R113) may be seen to be more efficient than global-list ones. The graphs in Figs. 14.35 and 14.36 show a large amount of variation, and this may be attributed to the pseudo-random interactions of the many idle processors.

14.4.3 Wait Algorithms

In all of the previous tests, the improved WAIT algorithm, introduced in chapter 12, was used. For comparison, a number of tests were performed using the simple WAIT algorithm. Figs. 14.37 and 14.38 show completion times and speedup factors for the region-filling task, using a variety of schedulers and this simple WAIT algorithm. These may be compared with similar graphs presented previously, which use the improved WAIT algorithm (Figs. 14.33 and 14.34). Some points are missing from the graphs in Figs. 14.37 and 14.38, where the simulated machine entered an infinite loop, due to simulated hardware lock-up in the synchronisation routines, as described in chapter 12. The long completion time of the wildfire algorithm operating with a global-list scheduler (R141) is due the development of a partial lock-

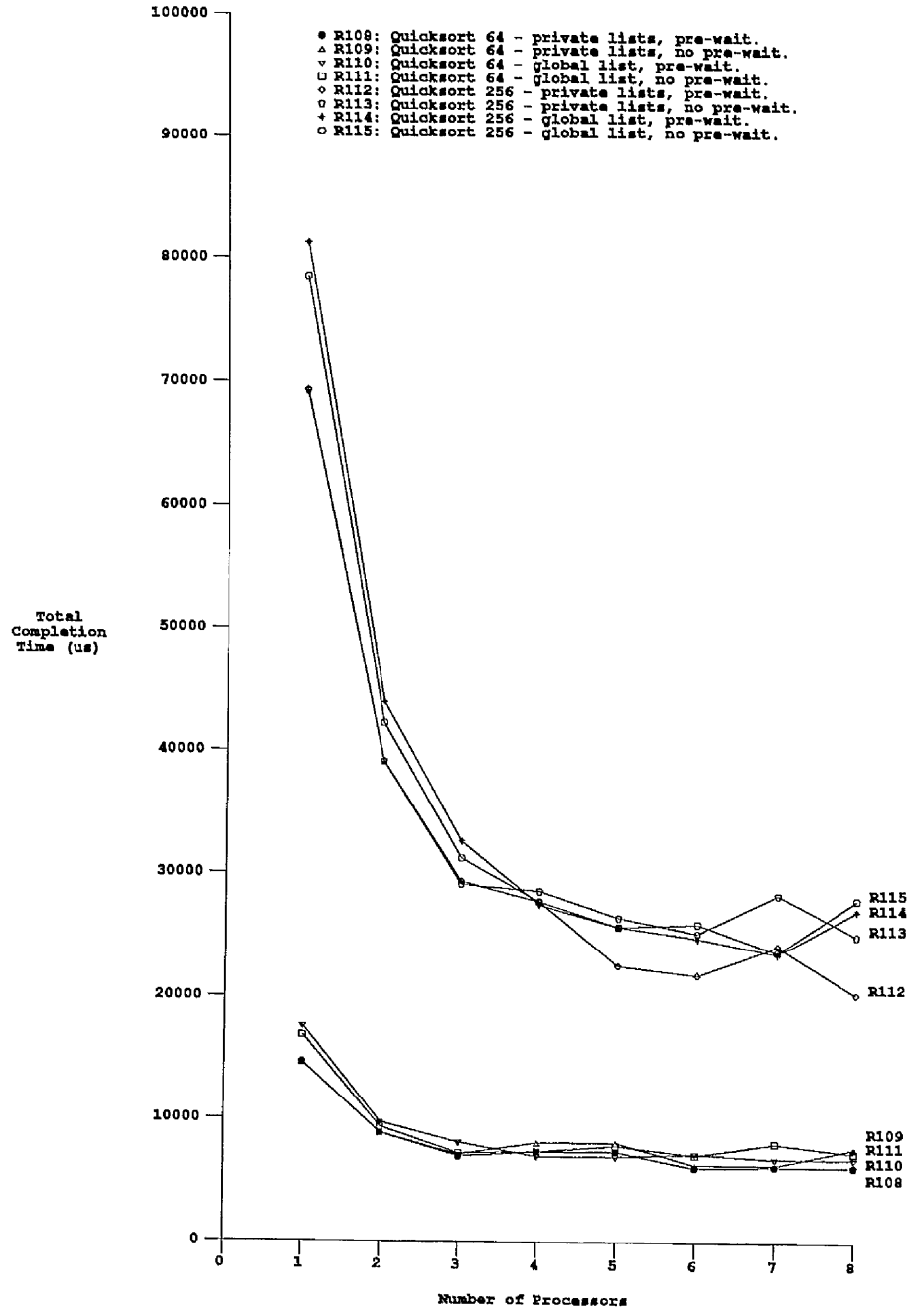


Fig. 14.35: Total completion times for the integer-sorting task.

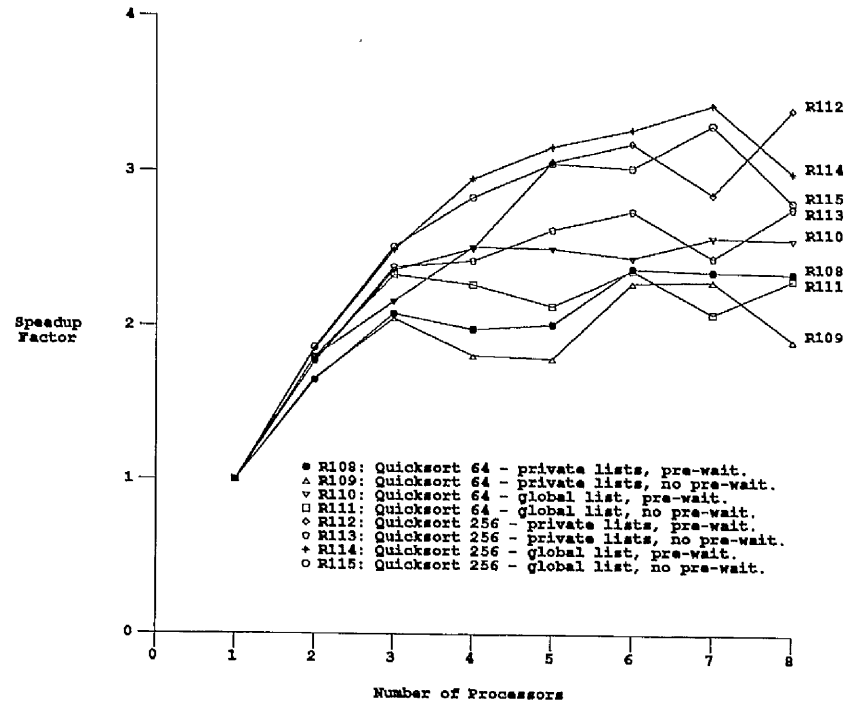


Fig. 14.36: Speedup factors for the integer-sorting task.

up state. The completion-time graphs for the simple, and the improved, WAIT algorithms (Figs. 14.33 and 14.37) appear similar, but comparison of the speedup graphs (Figs. 14.34 and 14.38) shows that, when the simple WAIT algorithm is used, the schedulers which do not use pre-wait mechanisms (R136, R138, R140, and R142) exhibit lower speedup factors than the corresponding pre-waiting schedulers (R135, R137, R139 and R141). This effect may be clearly seen in the case of the wildfire algorithm (R138, R139, R140 and R141). From these results, the use of the improved WAIT algorithm appears to be an advantage in some circumstances, and does not appear to cause an appreciable loss in performance in the remaining cases.

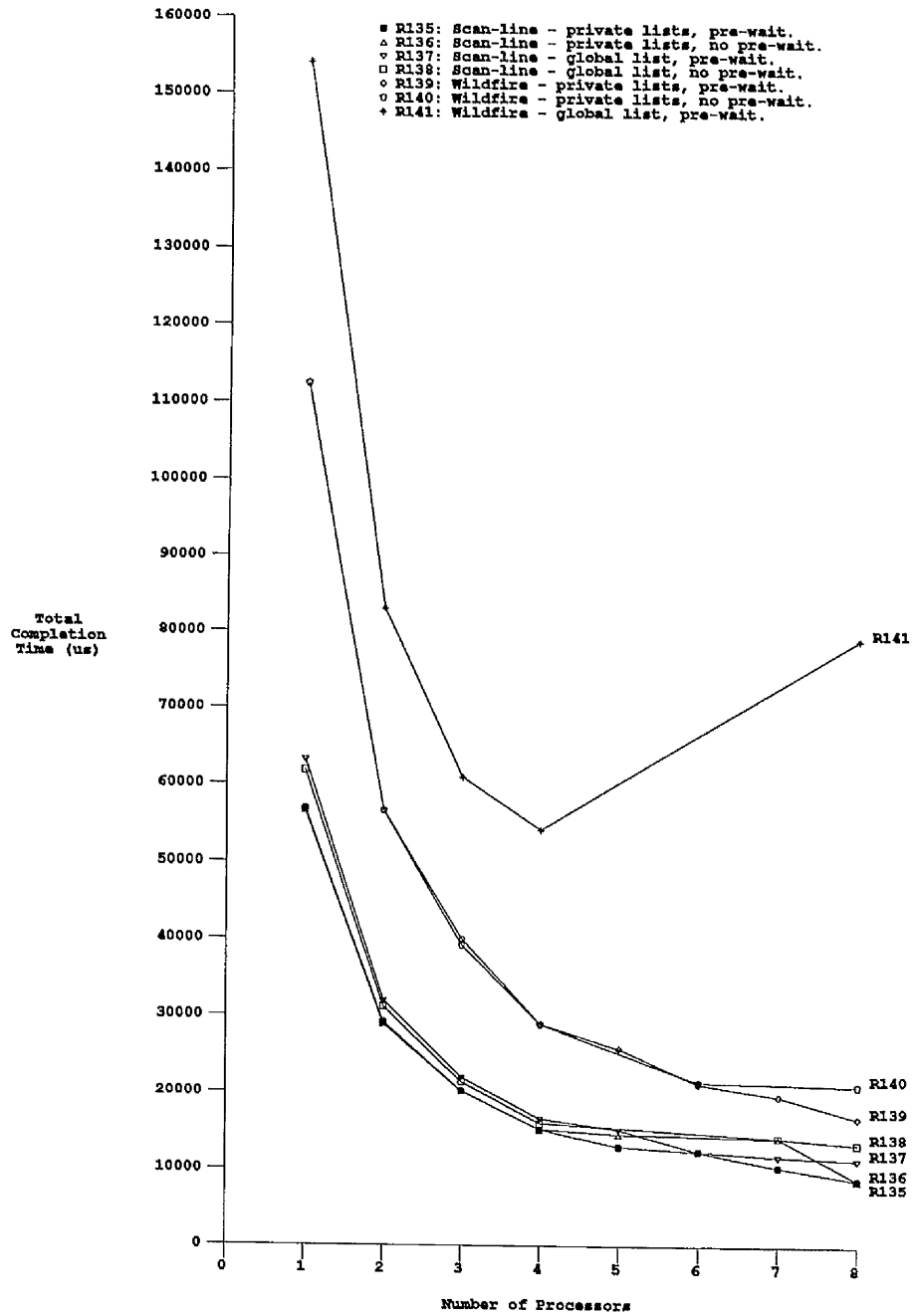


Fig. 14.37: Total completion times for the region-filling task using the simple WAIT algorithm.

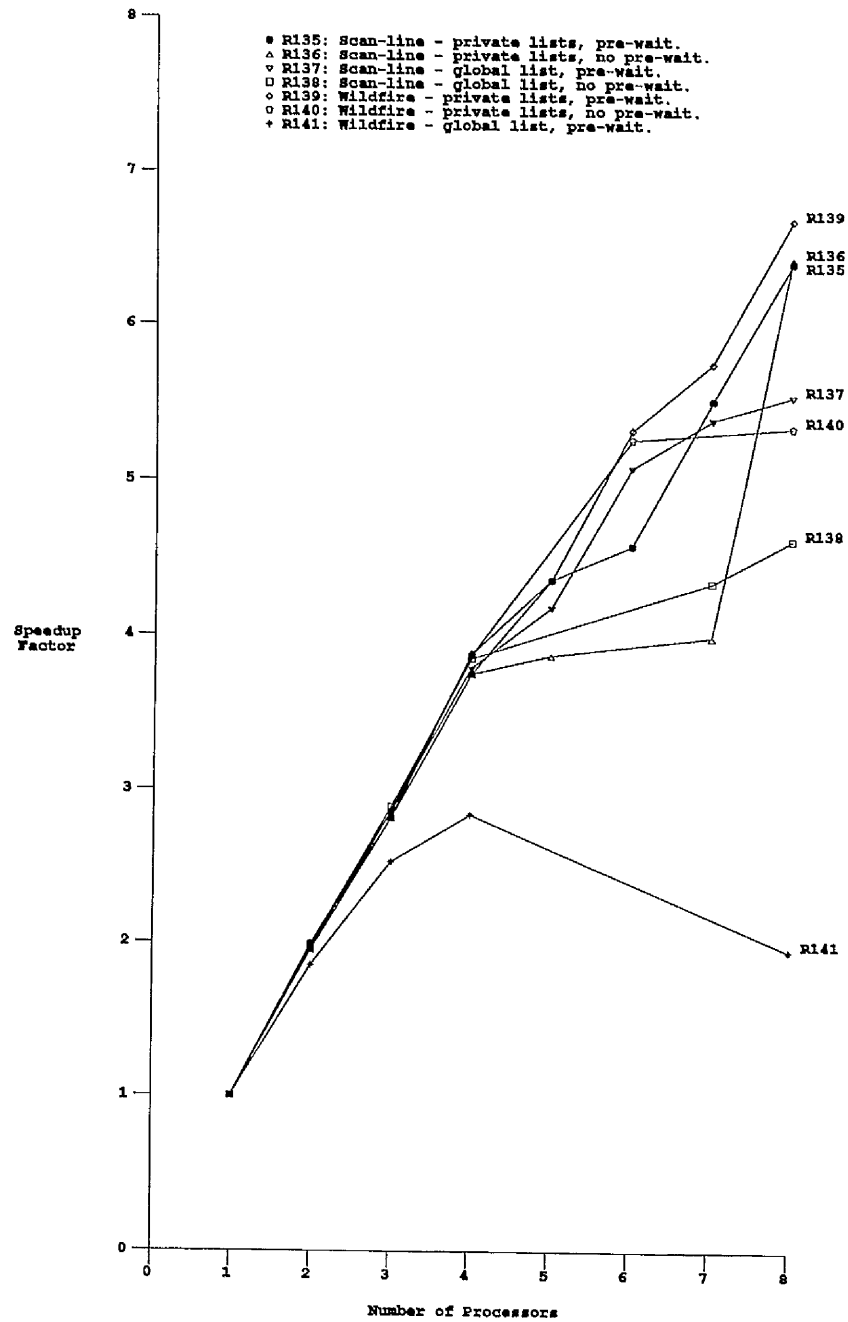


Fig. 14.38: Speedup factors for the region-filling task using the simple WAIT algorithm.

14.5 Summary

A number of shared-memory multiprocessors have been simulated, using the low-level simulator in conjunction with the MIMD-Pascal compiler, and the results of these experiments may be divided into the following categories:

i) Comparison of ring and binary n-tube based machines

Machines based on the partitioned indirect binary n-tube network have been compared with ones based on the simple unidirectional ring network, and it has been shown that the tube-based machine exhibits considerably better performance, attributable to its lower network latency.

ii) Investigation of the effects of variation of partitioned indirect binary n-tube length

A number of tests have been described which examine the performance of the partitioned indirect binary n-tube based machine, as the number of repeated sections is increased. It has been shown that the observed increase in network latency is not as significant as the theoretical predictions (made in chapter 8) might indicate. This is because packets which are deflected from their preferred route may, in the tube network, recover their course by making use of the multiple paths which exist between any pair of functional units.

iii) Comparison of arbitration strategies for packet-routing clashes

Two of the arbitration strategies for packet-routing clashes which were described in chapter 9 have been compared, and it has been shown that the 'nearest-to-destination has priority' strategy gives better machine performance than an approximation to a random priority allocation scheme.

iv) The effects of the speed of network control units

A number of machines were simulated with differing speeds of network control unit, and it has been shown that the operating speed of the NCU is an important factor in determining the overall machine performance. It would be essential, for a practical machine, that the network-cycle time be reduced to as low a figure as is economically possible.

v) Differences between functional unit placement patterns

It has been shown that performance of the simulated machines may be affected by the pattern in

which processing units and memory units are connected to the interconnection network. Although this is not a major factor in determining the machine performance, efficient placement patterns may be used in preference to others at no extra cost, and so this appears to be worthwhile.

vi) The use of local instruction stores

In most tests, the simulated machines made use of a local program store to allow instructions to be obtained without the need for access to the shared memory, across the interconnection network. An experiment in which this local store was removed demonstrated that machine performance is greatly reduced in such cases.

vii) The effects of memory address translation

The use of memory address translation hardware to reduce the number of memory-accessing clashes has been examined. Although some anomalous behaviour due to unit placement patterns was observed, the use of an address translation mechanism seems to reduce the number of memory-accessing clashes and, thus, improve the machine performance.

viii) Machine performance for differing tasks

A number of differing tasks were simulated, and it was found that the hardware performance of the machine varied slightly from one task to another. The dominant factor in the production of these differences appears to be the proportion of time spent by each task in the system synchronisation procedures. It has also been shown that, using shared-memory multiprocessors, it is possible to achieve good machine performance for a feature-oriented task (region filling), and that the speedup factor obtained for such tasks is only slightly less than that obtained for a task with pixel-oriented parallelism (linear filtering).

ix) Examination of run-time job scheduling algorithms

A number of run-time job scheduling algorithms have been investigated, and the use of private job lists has been shown to be advantageous in most circumstances. The use of pre-waiting in run-time schedulers has also been shown to be of use in some situations.

x) Implementation of system synchronisation procedures

Two different algorithms were tested for the system synchronisation procedure, WAIT. It has

been shown that, in most cases, the use of the improved WAIT algorithm described in chapter 12 results in better overall performance, particularly when large numbers of processors are involved.

These results appear to confirm that shared-memory multiprocessors are suitable for image analysis tasks, and that reasonable performance may be obtained using machines based on the partitioned indirect binary n-tube network.

"I am afraid that I rather give myself away when I explain," said he.
"Results without causes are much more impressive."

Sherlock Holmes, in 'The Adventure of the Stockbroker's Clerk'
— Sir Arthur Conan Doyle

Chapter 15:

Conclusions

15.1 Medical Applications of Computer Image Analysis

Computer image analysis has a number of medical applications, some of which have been examined in chapter 1. Two applications in clinical ophthalmology have also been investigated in detail:

i) Cup volume evaluation in glaucoma

A model-based system for the automatic evaluation of cupping volume in glaucoma was partially implemented. This scheme uses prior knowledge of the shape of the optic disc cup to construct a model of the disc surface, which is then iteratively modified. It was found, however, that this approach to cup volume evaluation was too computationally demanding for the available hardware.

ii) Automatic measurement of corneal endothelial cells

A program for the automatic detection and measurement of corneal endothelial cells was also developed. Although this program successfully detects a high proportion of the cells in an image, it requires some manual correction, and the hardware costs and appreciable run-time make the system somewhat uneconomic for clinical use.

From experience with these applications, and the study of others, it has become apparent that more processing power is required for many medical image analysis tasks than is currently available. It is possible that such increases in system throughput may be obtained by the exploitation of parallelism which exists in these tasks. A study has been made, therefore, of parallel hardware structures which may be suitable for image analysis.

15.2 A Taxonomy for Parallel Machine Description

A survey of existing parallel processors has been made, and a taxonomy developed to describe such machines. This taxonomy is based on the description of machines in terms of simple architectural units

and interconnection networks. A matrix notation has been introduced to describe these interconnection networks and four functions have been defined which may be used to derive useful quantitative characteristics of networks from their matrix representations. These functions are the cost, the bandwidth, the maximum latency and the mean latency of the network. The bandwidth function is computationally expensive to calculate for all but very small networks, but it is likely that analytic expressions may be derived for many regular networks by the application of rook polynomial techniques. The parallel machine taxonomy has been used to describe a number of classes of parallel machines, and examples of each class have been presented.

15.3 A Proposed Machine Structure for Image Analysis

A parallel machine structure, suitable for image analysis, has been proposed, and critical parts of this have been described in detail. In this machine, a number of processing units and memory units are connected, via network control units, to a partitioned indirect binary n-tube network. This network is used to pass data packets from one functional unit to another and, thus, a shared, distributed memory is provided. The partitioned indirect binary n-tube is a partially-connected network, composed of repeated sections which are based on partitioned indirect binary n-cubes. In general, a number of different routes exist between any two units, and this provides a certain amount of fault-tolerance, since the network control units may be designed so as to route packets around failed units. The partitioned indirect binary n-tube may be easily expanded, by adding extra sections to extend the tube length. Each section is identical, and this may be exploited in a VLSI implementation, which would reduce the hardware cost of the machine. A number of aspects of this machine structure have been examined:

i) The effects of network latency

From theoretical predictions and high-level simulations, it may appear that, in comparison with the partitioned indirect binary n-cube, the higher latency of the tube network may present serious problems. However, it was found that, in low-level simulation, the actual observed latency was not excessive. This is due, in part, to the multiple paths between units allowing course-corrections of deflected packets, which cannot occur in the partitioned indirect binary n-cube. The observed latency of an 8-repeated partitioned indirect binary 2-tube was only 1.7 times that for a partitioned indirect binary 4-cube, which supports the same number of functional units. This implies

that the performance of a tube-based machine would be slightly lower than that of a cube-based machine of equivalent size, but the tube machine has the advantages of fault-tolerance and expandability. The results of these simulations show that the partitioned indirect binary n-tube appears to be a practical network for the interconnection of processors and memory units, in a machine for image analysis applications.

ii) Run-time scheduling algorithms

Not all image analysis tasks may be efficiently divided between a number of processors using only the information available at compile-time. Many feature-oriented problems fall into this category and, for this form of task, it is necessary to use some form of run-time scheduler to distribute work evenly across the machine. A number of scheduling algorithms have been investigated, and it has been shown that efficient run-time scheduling methods are essential for good overall machine performance. Schedulers which use private job list structures have been shown to give better performance than those which use a single global list, but much scope exists for future research in this area.

iii) Process synchronisation

A simple method of inter-process communication was implemented, using semaphores. This mechanism was found to be inefficient when many processes required access to a single variable, and it is possible that better performance may be obtained from a system which maintains a queue of waiting processes for each semaphore, rather than repeatedly polling the semaphores as in the current system. This currently-implemented system uses increment-in-memory and decrement-in-memory instructions, and this was found to be the cause of hardware lock-ups, under certain circumstances. This could, however, be prevented by slight modification of the hardware provided in the memory unit to perform these operations (as described in chapter 12).

iv) Run-time system organisation

It would be a great advantage, in a system for practical use, if the compilation of programs could be performed without knowledge of the exact hardware configuration. In the event of a processor failure, this would allow the faulty unit to be removed, and the machine re-started without the need to re-compile all software for the new machine configuration. This could be achieved by

minor changes in the run-time scheduling algorithm, and one possible approach would be to only allocate parts of the task to those processors which have indicated their readiness by some fixed time after start-up.

v) The nature of simulated tasks

The tasks which were simulated were, necessarily, short and, because of this, some of the results may not accurately reflect the performance which might be achieved by a real machine. In most of the simulations described in the previous chapter, the time which was spent in processing each pixel was relatively short in comparison with the overheads involved. The overhead involved in processing each pixel is approximately constant, and is composed of two parts:

a) the time required by the run-time scheduler, and

b) the time taken to copy the required pixel values from the shared memory to a processor register.

In the case of more complex operators, such as the median filter, where more useful work is performed for this fixed overhead, it is possible that simulations might have produced more favourable estimates of machine performance.

vi) The use of parallelism at a procedure level

The language MIMD-Pascal has been presented, which allows parallelism to be expressed at a procedural level. The implementation of a compiler for MIMD-Pascal has also been described. The use of parallelism at a procedure level is seen to be appropriate for the expression of feature-oriented parallelism in a shared-memory multiprocessor, since generally, in such machines, the overheads involved in the exploitation of parallelism increase as the granularity of parallelism decreases. MIMD-Pascal has been successfully used to write application programs (for test purposes) and run-time scheduling routines for use with the low-level machine simulator.

vii) The use of local data memory units

MIMD-Pascal allows the user to specify which data items are to be globally accessible and which are private to some processor. In the current implementation, however, all data is stored in the distributed, shared memory. Although some areas of this shared memory are designated as private, and are protected by software, the use of these data areas requires a shared-memory access

to be made via the interconnection network. This is undesirable, as such accesses are relatively slow, and consume shared-memory resources which may be required by other processors. In a future implementation, it would be quite straightforward for private data items to be placed in a local data store within the processing unit, since all the information required to identify such items is readily available at compile-time.

viii) The use of interactive simulation

The use of an interactive simulator was found to be extremely useful in fault-finding, and in the interpretation of simulation results. The ability to observe the detailed action of the simulator was invaluable in understanding some of the complex interactions which occur both in hardware, and between software processes.

Considering all of these factors, the partitioned indirect binary n-tube architecture and the programming language, MIMD-Pascal, would appear to be useful for medical image analysis applications and, with the modifications outlined above, appear to be worthy of further research.

"All generalisations are dangerous, even this one."

Alexandre Dumas (fils)

"Well, the Tale is now told, from first to last.
Here we all are, and here is the Ring.
But we have not come any nearer to our purpose.
What shall we do with it?"

Gandalf, in 'The Lord of the Rings'

— J.R.R. Tolkien

References

[1] **International Association for Pattern Recognition (1983).** "Proceedings of the Third Scandinavian Conference on Image Analysis (Copenhagen, Denmark)". Chartwell-Bratt, Bromley.

[2] **Schuster, E. and Grösser, P. (1983).** "Digital Processing of Ultrasonic Images for Diagnostic Purposes". In [1], pp. 398–403.

[3] **Colonna, K. N. and Oliphant, G. (1986).** "Development and Evaluation of Digitally Processed Z-Contrast Imaging: A Technological Advance in Medical Imaging". *Computer Methods and Programs in Biomedicine* **22**, pp. 333–342.

[4] **Preston, K. Jr., Taylor, K. J. W., Johnson, S. A. and Ayers, W. R. (eds) (1979).** "Medical Imaging Techniques — A Comparison". Plenum Press, New York.

[5] **Preston, K. Jr. and Duff, M. J. B. (1984).** "Modern Cellular Automata". Plenum Press, New York.

[6] **Alexander, P. (1983).** "Array Processors in Medical Imaging". *IEEE Computer* **16(6)**, pp. 17–30.

[7] **Gonzalez, R. C. and Wintz, P. (1977).** "Digital Image Processing". Addison-Wesley, Reading, Massachusetts.

[8] **Castleman, K. R. (1979).** "Digital Image Processing". Prentice Hall, Englewood Cliffs, New Jersey.

[9] **Schellart, N. A. M., Zweijpfenning, R. C. J. V., van Marle, J. and Huijsmans, D. P. (1986).** "Computerized Pattern Recognition Used for Grain Counting in High Resolution Autoradiographs With Low Grain Densities". *Computer Methods and Programs in Biomedicine* **23**, pp. 103–109.

[10] **Gardner, W. E. (ed) (1979).** "Machine Aided Image Analysis 1978 — Proceedings of the International Conference on Applications of Machine Aided Image Analysis (Oxford, September 1978)". Institute of Physics, Bristol.

[11] **Jagoe, J. R. (1979).** "Measurement of Pneumoconiosis in Chest Films by Computer". In [10], pp. 203–209.

[12] **Onoe, M., Preston K. Jr. and Rosenfeld, A. (eds) (1980).** "Real-Time Medical Image Processing". Plenum Press, New York.

[13] **Sklansky, J., Sankar, P. V., Katz, M., Towtig, F., Hassner, D., Cohen, A. and Root, W. E. (1980).** "Towards Computed Detection of Nodules in Chest Radiographs". In [12], pp. 53–59.

[14] **Gonçalvez, J. G. M. (1982).** "Computer Processing of Chest Radiographs". Ph.D. Thesis, Department of Medical Biophysics, University of Manchester.

[15] **IEEE Computer Society (1985).** "Proceedings of the 1985 Computers in Cardiology Conference (Linköping, Sweden)". IEEE Computer Society Press, Washington, D. C..

[16] **Marcus, E., Lorente, P., Bartha, E., Beyar, R., Adam, D. and Sideman, S. (1985).** "A Comparative Study of Quantitative Methods for Characterisation of Left Ventricular Contraction". In [15], pp. 145–148.

[17] **Chappuis, F., Ratib, O., Chatelain, P., Meier, B., Righeti, A. and Rutishauser, W. (1985).** "Computer Analysis of Digitized Coronary Angiograms for the Assessment of Changes in Poststenotic Coronary Flow". In [15], pp. 21–26.

[18] **International Association for Pattern Recognition (1986).** "Proceedings of the Eighth International Conference on Pattern Recognition (Paris, France), Volume 1". IEEE Computer Society Press, Piscataway, New Jersey.

[19] **Kitamura, K., Tobis, J. M. and Sklansky, J. (1986).** "Estimating the X-Ray Intercepted Areas and Boundaries of Coronary Arteries.". In [18], pp. 478–480.

[20] **Boecker, F. R. P., Witte, G. and Hoehne, K. H. (1986).** "Contour Detection in DSA-Images as Basis for Three-Dimensional Reconstruction of Ventriculograms". In [18], pp. 484–486.

[21] **Wied, G. L. and Bahr, G. F. (eds) (1970).** "Automated Cell Identification and Cell Sorting". Academic Press, London.

[22] **Lester, J. M., Brenner, J. F. and Selles, W. D. (1980).** "Local Transforms for Biomedical Image Analysis". *Computer Graphics and Image Processing* **13**, pp. 17–30.

[23] **IEEE Computer Society (1981).** "IEEE Computer Society Workshop on Computer Architecture for Pattern Analysis and Image Database Mangement (Hot Springs, Virginia)". IEEE Computer Society Press, Piscataway, New Jersey.

[24] **Preston, K. Jr. (1981).** "Architectures for Image Processing in Biomedical Microscopy". In [23], pp. 318–322.

[25] **Pressman, N. J. and Wied, G. L. (eds) (1979).** "Proceedings of the Second International Conference on the Automation of Cancer Cytology and Cell Image Analysis (Tokyo, Japan, May 1977)". The Tutorials of Cytology, Chicago, Illinois.

[26] **Read, J. S., Borovec, R. T., Bartels, P. H., Bibbo, M., Puls, J. H., Reale, F. R., Taylor, J. and Wied, G. L. (1979).** "A Fast Image Processor for Locating Cell Nuclei in Uterine Specimens". In [25], pp. 143–155.

[27] **Mukawa, A., Tanaka, N., Ikeda, H. and Ueno, T. (1979).** "A Practical Evaluation of the CYBEST System in the Use of Mass-Screening on Epidermoid Carcinoma of the Uterine Cervix". In [25], pp. 193–199.

[28] **Tanaka, N., Ikeda, H., Ueno, T., Watanabe, S., Imasato, Y., Tsunekawa, S., Okamoto, Y., Kashida, R. and Mukawa, A. (1979).** "Experimental Practical Use of Automated Screening System (CYBEST) for the Mass-Screening of Gyneological Samples". In [25], pp. 123–134.

[29] **Watanabe, S., Tsunekawa, S., Okamoto, Y., Sasao, I., Tomaru, T. (1980).** "The Development of a New Model Cyto-Prescreener for Cervical Cancer". In [12], pp. 221–229.

[30] **Bengtsson, E., Dahlqvist, B., Eriksson, O., Jarkrans, T., Nordin, B. and Stenkvist, B. (1983).** "Cervical Pre-Screening Using Computerized Image Analysis". In [1], pp. 404–411.

[31] **American Association for Clinical Chemistry (1982).** "Proceedings of the International Conference on Clinical Applications and Developments in Two-Dimensional Electrophoresis (Rochester, Minnesota, November 1981)". Published as *Clinical Chemistry* **28(4)**.

[32] **Jellum, E. and Thorsrud, A. K. (1982).** "Clinical Applications of Two-Dimensional Electrophoresis". In [31], pp. 876–883.

[33] **O'Farrell P. H. (1975).** "High Resolution Two-Dimensional Electrophoresis of Proteins". *Journal of Biological Chemistry* **250**, pp. 4007–4021.

[34] **Andrews, A. T. (1986).** "Electrophoresis Theory, Techniques, and Biochemical and Clinical Applications (Second Edition)". Clarendon Press, Oxford.

[35] **Anderson, N. G. and Anderson, L. (1982).** "The Human Protein Index". In [31], pp. 739–748.

[36] **Brown, W. T. and Ezer, A. (1982).** "A Computer Program Using Gaussian Fitting for Evaluation of Two-Dimensional Gels". In [31], pp. 1041–1044.

[37] **Fox, S. H. (1982).** "Some Relatively Simple Steps Toward a Computer System for the Analysis of Two-Dimensional Gel-Electrophoresis Autoradiographs". In [31], pp. 932–934.

[38] **Lemkin, P. F. and Lipkin, L. E. (1981).** "GELLAB: A computer system for 2D Gel Electrophoresis Analysis I. Segmentation of Spots and System Preliminaries". *Computers and Biomedical Research* **14**, pp. 272–297.

[39] **Lemkin, P. F. and Lipkin, L. E. (1981).** "GELLAB: A computer system for 2D Gel Electrophoresis Analysis. II. Pairing Spots". *Computers and Biomedical Research* **14**, pp. 355–380.

[40] **Lemkin, P. F. and Lipkin, L. E. (1981).** "GELLAB: A computer system for Two-Dimensional Gel Electrophoresis Analysis. III. Multiple Two-Dimensional Gel Analysis". *Computers and Biomedical Research* **14**, pp. 407–446.

[41] **Miller, M. J., Vo, P. K., Nielsen, C., Geiduschek, E. P. and Xuong, Ng.-H. (1982).** "Computer Analysis of Two-Dimensional Gels: Semi-Automatic Matching". In [31], pp. 847–875.

[42] **Taylor, J., Anderson, N. L., Scandora, A. E. Jr., Willard, K. E. and Anderson, N. G. (1982).** "Design and Implementation of a Prototype Human Protein Index". In [31], pp. 861–866.

[43] **Skolnick, M. M., Sternberg, S. R. and Neel, J. V. (1982).** "Computer Programs for Adapting Two-Dimensional Gels to the Study of Mutation". In [31], pp. 969–978.

[44] **Skolnick, M. M. (1982).** "An Approach to Completely Automatic Comparison of Two-Dimensional Electrophoretic Gels". In [31], pp. 979–986.

[45] **Sternberg, S. R. (1983).** "Biomedical Image Processing". *IEEE Computer* **16(1)**, pp. 22–34.

[46] **Potter, D. J. (1986).** "Parallel Algorithms for the Analysis of Two-Dimensional Electrophoresis Gels". *Computers and Biomedical Research* **19**, pp. 565–574.

[47] **Granum, E. (ed) (1981).** "Proceedings of the IVth European Chromosome Analysis Workshop (Edinburgh)". MRC Clinical and Population Cytogenetics Unit, Edinburgh.

[48] **Philip, J. and Philip, K. (1981).** "The Need for Clinical Chromosome Analysis". In [47], pp. 6.3.1–6.3.8.

[49] **Graham, J. (1981).** "Magiscan as Metaphase Finder". In [47], pp. 8.1c.1–8.1c.3.

[50] **Graham, J. (1981).** "Metaphase Analysis Using Magiscan". In [47], pp. 10.1.1–10.1.3.

[51] **Lundsteen, C., Christiansen, U., Gerdes, T., Philip, J. and Phillip, K. (1981).** "A Semi-Automatic Karyotyping System for Prenatal Chromosome Analysis". In [47], pp. 10.2.1–10.2.6.

[52] **Lundsteen, C., Gerdes, T., Philip, J., Graham, J. and Pycock, D. (1983).** "An Interactive System for Chromosome Analysis: Tests of Clinical Performance". In [1], pp. 392–397.

[53] **Graham, J., Taylor, C. J., Cooper, D. H. and Dixon, R. N. (1986).** "A Compact Set of Image Processing Primitives and Their Role in a Successful Application Program". *Pattern Recognition Letters* 4, pp. 325–333.

[54] **Piper, J. and Nickolls, P. (1981).** "The Edinburgh Chromosome Analysis Project: Progress Report - Digital Filtering and Centromere Finding". In [47], pp. 4.5.1–4.5.3.

[55] **Khochtinat, N. and Le Gô, R. (1981).** "Semi-Automatic Karyotyping. Actual and Future Developments at Fontenay-Aux-Roses". In [47], pp. 10.3.1–10.2.12.

[56] **Baudoin, C. E., Lay, B. J. and Klein, J. C. (1984).** "Automatic Detection of Microaneurisms in Diabetic Fluorescein Angiography". *Revue d'Epidemiologie et de Sante Publique* **32**, pp. 254–261.

[57] **Scheie, H. G. and Albert, D. M. (1977).** "Adler's Textbook of Ophthalmology, Ninth Edition". W. B. Saunders Company, Philadelphia.

[58] **Naumann, G. O. H. and Apple, D. J. (1986).** "Pathology of the Eye". Springer-Verlag, New York.

[59] **Ruprecht, K. W. and Naumann, G. O. H. (1986).** "The Eye and Systemic Disease". In [58], pp. 873–956.

[60] **Greenfield, R. H. and Colenbrander, A. (eds) (1979).** "Proceedings of the First Meeting on Computers in Ophthalmology (St. Louis, Missouri, April 1978)". IEEE Press, New York.

[61] **Read, J. S., Peterson, A. C., McCormick, B. H. and Goldberg, M. F. (1979).** "The Television Ophthalmoscope Image Processor: Methods and Applications". In [60], pp. 123–132.

[62] **Fram, I., Read, J. S., McCormick, B. H. and Fishman, G. A. (1979).** "In Vivo Study of the Photolabile Visual Pigment Utilizing the Television Ophthalmoscope Image Processor". In [60], pp. 133–144.

[63] **Hitchings, R. A. (1978).** "The Optic Disc in Glaucoma (III) — Diffuse Optic Disc Pallor With Raised Intraocular Pressure". *British Journal of Ophthalmology* **62**, pp. 670–675.

[64] **Vaughan, D. and Asbury, T. (1980).** "General Ophthalmology, Ninth Edition". Lange Medical Publications, Los Altos, California.

[65] **Dorrel, E. D. (1978).** "Surgery of the Eye". Blackwell Scientific Publications, Oxford.

[66] **Lim, A. S. M. and Constable, I. J. (1979).** "A Colour Atlas of Ophthalmology". Henry Kimpton Publishers, London.

[67] **Morgan, R. W. and Drance, S. M. (1975).** "Chronic Open Angle Glaucoma and Ocular Hypertension". *British Journal of Ophthalmology* **59**, pp. 211–215.

[68] **Hitchings, R. A., Powell, D. J., Arden, G. B. and Carter, R. M. (1981).** "Contrast Sensitivity in Glaucoma Family Screening". *British Journal of Ophthalmology* **65**, pp. 515–517.

[69] **Talusan, E. D. and Schwartz, B. (1981).** "Episcleral Venous Pressure Differences Between Normal, Ocular Hypertensive and Primary Open Angle Glaucomas". *Archives of Ophthalmology* **99**, pp. 824–828.

[70] **New Orleans Academy of Ophthalmology (1967).** "Symposium on Glaucoma — Transactions of the Fifteenth Session of the New Orleans Academy of Ophthalmology". C. V. Mosby Company, St. Louis.

[71] **Kronfeld, P. C. (1967).** "The Optic Nerve". In [70], pp. 62–73.

[72] **Rock, W. J., Drance, S. M. and Morgan, R. W. (1973).** "Visual Field Screening in Glaucoma". *Archives of Ophthalmology* **89**, pp. 287–290.

[73] **Cant, J. S. (ed) (1972).** "The Optic Nerve — Proceedings of the Second William Mackenzie Memorial Symposium (Glasgow, 1971)". Henry Kimpton Publishers, London.

[74] **Haining, W. M. (1972).** "Stereomorphography of the Optic Nerve Head". In [73], pp. 317–322.

[75] **Snydacker, D. (1964).** "The Normal Optic Disc". *American Journal of Ophthalmology* **58**, pp. 958–963.

[76] **Fishman, R. S. (1970).** "Optic Disc Asymmetry: A Sign of Ocular Hypertension". *Archives of Ophthalmology* **84**, pp. 590–594.

[77] **Kirsch, R. F. and Anderson, D. R. (1973).** "Clinical Recognition of Glaucomatous Cupping". *American Journal of Ophthalmology* **75**, pp. 442–454.

[78] **Portney, G. L. (1973).** "Qualitative Parameters of the Normal Optic Nerve Head". *American Journal of Ophthalmology* **76**, pp. 655–659.

[79] **Schwartz, B., Reulin, F. H. and Garrison, R. J. (1975).** "Acquired Cupping of the Optic Nerve Head in Normotensive Eyes". *British Journal of Ophthalmology* **59**, pp. 216–222.

[80] **Schwartz, B. (1980).** "Optic Disc Changes in Ocular Hypertension". *U. S. A. Survey of Ophthalmology* **25**, pp. 148–154.

[81] **Read, R. M. and Spaeth, G. L. (1974).** "The Natural History of Cup Progression and Some Specific Disc-Field Correlations". *Transactions of the American Academy of Ophthalmology and Otolaryngology* **78**, pp. 255–274.

[82] **Tomlinson, A. and Phillips, C. I. (1974).** "Ovalness of the Optic Cup and Disc in the Normal Eye". *British Journal of Ophthalmology* **58**, pp. 543–547.

[83] **Kitazawa, Y., Horie, T., Aokie, S., Suzuki, M. and Nishioka, K. (1977).** "Untreated Ocular Hypertension: A Long Term Study". *Archives of Ophthalmology* **95**, pp. 1180–1184.

[84] **Krakau, C. E. T. (1972).** "Measurements of the Excavation Volume of the Optic Disc". In [73], pp. 304–310.

[85] **Schwartz, B. (1973).** "Cupping and Pallor of the Optic Disc". *Archives of Ophthalmology* **89**, pp. 272–277.

[86] **Spaeth, G. L. (1977).** "The Pathenogenesis of Nerve Damage in Glaucoma". Grune and Stratton, New York.

[87] **Schwartz, B., Reinstein, N. M. and Lieberman, D. M. (1973).** "Pallor of the Optic Disc". *Archives of Ophthalmology* **89**, pp. 278–286.

[88] **Hoskins, H. D. and Gelber, E. C. (1975).** "Optic Disc Topography and Visual Defects in Patients with increased Intraocular Pressure". *American Journal of Ophthalmology* **80**, pp. 284–292.

[89] **Hitchings, R. A. and Spaeth, G. L. (1976).** "The Optic Disc in Glaucoma (I) — Classification". *British Journal of Ophthalmology* **60**, pp. 778–785.

[90] **Hitchings, R. A. and Spaeth, G. L. (1977).** "The Optic Disc in Glaucoma (II) — Correlation of the Optic Disc with the Visual Field". *British Journal of Ophthalmology* **61**, pp. 107–113.

[91] **Hitchings, R. A. and Wheeler, C. A. (1978).** "The Optic Disc in Glaucoma (IV) — Optic Disc Evaluation in the Ocular Hypertensive Patient". *British Journal of Ophthalmology* **64**, pp. 232–239.

[92] **Crick, R. P. (ed) (1978).** "Glaucoma — The Way Ahead. Proceedings of the Bristol Symposium on Glaucoma 1978". MCS Consultants Publishing, Tunbridge Wells..

[93] **Heijl, A. and Krakau, C. E. T. (1975).** "An Automatic Static Perimeter, Design and Pilot Study". *Acta Ophthalmologica* **53**, pp. 293–310.

[94] **Krakau, C. E. T. (1978).** "Aspects on the Design of an Automatic Perimeter". *Acta Ophthalmologica* **56**, pp. 389–405.

[95] **McCray, J. A. and Feignon, J. (1979).** "Computerised Perimetry in Neuro-Ophthalmology". *Ophthalmology* **86**, pp. 1287–1301.

[96] **Johnson, C. A. and Keltner, J. L. (1981).** "Computer Analysis of Visual Field Loss and Optimisation of Automated Perimetric Test Strategies". *Ophthalmology* **88**, pp. 1058–1065.

[97] **Krakau, C. E. T. (1981).** "A Feasable Development of Computerised Perimetry". *Acta Ophthalmologica* **59**, pp. 485–495.

[98] **Ernst, W., Faulkner, D. J., Hogg, C. R., Powell, D. J., Arden, G. B. and Vaegan (1983).** "An Automated Static Perimeter/Adaptometer Using Light Emitting Diodes". *British Journal of Ophthalmology* **67**, pp. 431–442.

[99] **Biesel, H., Kulikowski, C., Weiss S. and Aviner, Z. (1979).** "Computer-Aided Acquisition and 3-Dimensional Display of Visual Field Data". In [60], pp. 182–185.

[100] **Hart, W. M. Jr. (1979).** "Computer Applications to Visual Fields". In [60], pp. 157–160.

[101] **Hart, W. M. and Hartz, R. K. (1982).** "Computer Generated Display for Three Dimensional Static Perimetry". *Archives of Ophthalmology* **100**, pp. 312–318.

[102] **Pe'er, J. and Zajicek, G. (1980).** "Computer Analysis of the Optic Disc". *Ophthalmologica* **181**, pp. 266–270.

[103] **Portney, G. L. (1974).** "Photogramatic Categorical Analysis of the Optic Nerve Head". *Transactions of the American Academy of Ophthalmology and Otolaryngology* **78**, pp. 275–289.

[104] **Portney, G. L. (1975).** "Photogramatic Analysis of Volume Asymmetry of the Optic Nerve Cup in Normal, Hypertensive and Glaucomatous Eyes". *American Journal of Ophthalmology* **80**, pp. 51–55.

[105] **Crone, D. R. (1963).** "Elementary Photogrammetry". Edward Arnold (Publishers) Limited, London.

[106] **Kottler, M. S., Rosental, A. R. and Falconer, D. G. (1974).** "Digital Photogrammetry of the Optic Nerve Head". *Investigative Ophthalmology* **13**, pp. 116–120.

[107] **Rosenthal, A. R., Kottler, M. S., Donaldson, D. D. and Falconer, D. G. (1976).** "Comparative Reproducibility of the Digital Photogrammetric Procedure Utilizing 3 Methods of Stereography". *Investigative Ophthalmology* **16**, pp. 54–60.

[108] **Falconer, D. G. and Rosenthal, A. R. (1979).** "Digital Photogrammetry on the Zeiss Model Eye". In [60], pp. 145–153.

[109] **Crock, G. and Pavel, J. M. (1969).** "Stereophotogrammetry of fluorescein angiographs in Ocular Biometrics". *Medical Journal of Australia* **2-1969**, pp. 586–590.

[110] **Crock, G. (1970).** "Stereotechnology in Medicine". *Transactions of the Ophthalmological Societies of the United Kingdom* **90**, pp. 577–636.

[111] **Gloster, J. (1972).** "Concerning Glautomatous Changes at the Disc". In [73], pp. 298–303.

[112] **Marr, D. (1982).** "Vision". W. H. Freeman and Company, San Francisco.

[113] **Ballard, D. H. and Brown, C. M. (1982).** "Computer Vision". Prentice Hall, Englewood Cliffs, New Jersey.

[114] **Newman, W. M. and Sproull, R. F. (1979).** "Principles of Interactive Computer Graphics (Second Edition)". McGraw-Hill Kogakusha, Tokyo.

[115] **Bui-Tong, P. (1975).** "Illumination for Computer Generated Images". *Communications of the ACM* **18**, pp. 311–317.

[116] **Whitted, T. (1980).** "An Improved Model for Shaded Display". *Communications of the ACM* **23**, pp. 343–349.

[117] **Duff, M. J. B. (ed) (1986).** "Intermediate Level Image Processing". Academic Press, London.

[118] **Taylor, C. J., Dixon, R. N., Gregory, P. J. and Graham, J. (1986).** "An Architecture for Integrating Symbolic and Numerical Image Processing". In [117], pp. 19–34.

[119] **Joyce-Loebl (Vickers) Limited (1981).** "Magiscan 2 Pascal Manual". Joyce-Loebl (Vickers) Limited, Newcastle-upon-Tyne.

[120] **Wied, G. L., Bahr, G. F. and Bartels, P. H. (1970).** "Automated Analysis of Cell Images by TICAS". In [21], pp. 195–360.

[121] **Preston, K. Jr. and Onoe, M. (eds) (1976).** "Digital Processing of Biomedical Images". Plenum Press, New York.

[122] **Preston, K. Jr. (1976).** "Clinical use of Automated Microscopes for Cell Analysis". In [121], pp. 47–58.

[123] **Tou, J. T. (1976).** "Automatic Analysis and Interpretation of Cell Micrographs". In [121], pp. 243–262.

[124] **Hamilton, W. J. (1976).** "A Textbook of Human Anatomy". Macmillan Press, London.

[125] **Svedbergh, B. and Bill, A. (1972).** "Scanning Electron Microscopic Studies of the Corneal Endothelium in Man and Monkeys". *Acta Ophthalmologica* **52**, pp. 321–336.

[126] **Bourne, W. M. and Kaufmann, H. E. (1976).** "Endothelial Damage Associated with Intraocular Lenses". *American Journal of Ophthalmology* **81**, pp. 482–485.

[127] **Bourne, W. M. and Kaufmann, H. E. (1976).** "Specular Microscopy of Human Corneal Endothelium in Vivo". *American Journal of Ophthalmology* **81**, pp. 319–323.

[128] **Blackwell, W. L., Gravenstein, N. S. and Kaufmann, H. E. (1977).** "Comparison of Central Corneal Endothelial Cells with Peripheral Areas". *American Journal of Ophthalmology* **84**, pp. 473–476.

[129] **Sturrock, G. D., Sherrard, E. S. and Rice, N. S. C. (1978).** "Specular Microscopy of the Corneal Endothelium". *British Journal of Ophthalmology* **62**, pp. 809–814.

[130] **Hoffer, K. J. (1979).** "Vertical Endothelium Cell Disparity". *American Journal of Ophthalmology* **87**, pp. 344–349.

[131] **Bron, A. J. and Brown, N. A. P. (1974).** "The Endothelium of the Corneal Graft". *Transactions of the Ophthalmological Societies of the United Kingdom* **94**, pp. 863–873.

[132] **Ridgeway, A. (1982).** Personal communication.

[133] **Maurice, D. M. (1968).** "Cellular Membrane Activity in the Corneal Endothelium of the Intact Eye". *Experimentia (Basel)* **24**, pp. 1094–1095.

[134] **Laing, R. A., Sandstrom, M. and Leibowitz, H. M. (1975).** "In Vivo Photomicrography of the Corneal Endothelium". *Archives of Ophthalmology* **93**, pp. 143–145.

[135] **Binder, P. S., Akers, P. and Zavala, E. Y. (1979).** "Endothelial Cell Density Determined by Specular Microscopy and Scanning Electron Microcopy". *Ophthalmology* **86**, pp. 1831–1847.

[136] **Langston, R. H. S. and Roisman, T. S. (1981).** "Comparison of Endothelial Evaluation Techniques". *American Intra-Ocular Implant Society Journal* **7**, pp. 239–241.

[137] **Drewe, R. H. (1981).** Personal communication.

[138] **Shaw, E. L., Rao, G. N., Arthur, E. J. and Aquavella, J. V. (1978).** "The Functional Reserve of Corneal Endothelium". *Ophthalmology* **85**, pp. 640–649.

[139] **Rao, G. N., Shaw, E. L., Stevens, R. E. and Aquavella, J. V. (1979).** "Automated Pattern Analysis of the Corneal Endothelium". *Ophthalmology* **86**, pp. 1367–1373.

[140] **Jacobi, K. W. and Strobel, J. (1981).** "Different Parameters of the Corneal Endothelium of the Human Eye". *American Intra-Ocular Implant Society Journal* **7**, pp. 140–142.

[141] **Lester, J. M., McFarland, J. L., Bursell, S., Laing, R. A. and Brenner, J. F. (1981).** "Automated Morphological Analysis of Corneal Endothelial Cells". *Investigative Ophthalmology and Visual Science* **20**, pp. 407–410.

[142] **Nilsson, N. J. (1971).** "Problem-Solving Methods in Artificial Intelligence". McGraw-Hill, New York.

[143] **Lester, J. M., Williams, H. A., Weintraub, B. A. and Brenner, J. F. (1978).** "Graph Searching Techniques for Boundary Finding in White Blood Cell Images". *Computers in Biology and Medicine* **8**, pp. 293–308.

[144] **Dixon, R. N. (1980).** "Aspects of Picture Processing". Ph.D Thesis, Department of Medical Biophysics, University of Manchester.

[145] **Marr, D. and Hildreth, E. (1980).** "Theory of Edge Detection". *Proceedings of the Royal Society, London* **B207**, pp. 187–217.

[146] **Torre, V. and Poggio, T. (1984).** "On Edge Detection". Massachusetts Institute of Technology A. I. Memorandum 768.

[147] **Moore, G. V. (1982).** "Applications of Computer Image Analysis in Clinical Ophthalmology". Internal Report, Department of Medical Biophysics, University of Manchester.

[148] **Edwards, W. (1966).** "Revision of Opinions by Men and Man-Machine Systems - Introduction". *IEEE Transactions on Human Factors in Electronics* **HFE-7**, pp. 1–7.

[149] **Lusted, L. B. (1968).** "Introduction to Medical Decision Making". Charles C. Thomas Publishing, Springfield, Illinois.

[150] **Wardle, A. and Wardle, L. (1978).** "Computer Aided Diagnosis — A Review of Research". *Methods of Information in Medicine* **17**, pp. 15–28.

[151] **Fritz, K. J., Polascik, M. A. and Potts, A. M. (1978).** "Computer Assisted Diagnosis for Ophthalmology". *Computers in Biology and Medicine* **8**, pp. 223–228.

[152] **Poppelbaum, W. J. (1968).** "What Next in Computer Technology?". *Advances in Computers* **9**, pp. 1–21.

[153] **Lorin, H. (1972).** "Parallelism in Hardware and Software: Real and Apparent Concurrency". Prentice Hall, Englewood Cliffs, New Jersey.

[154] **Enslow, P. H. Jr. (ed) (1974).** "Multiprocessors and Parallel Processing". John Wiley, New York.

[155] **Kuck, D. J. (1977).** "Parallel Processing of Ordinary Programs". *Advances in Computers* **15**, pp. 119–179.

[156] **Dertouzos, M. L. and Moses, J. (eds) (1979).** "The Computer Age: A Twenty Year Review (Revised Second Printing)". MIT Press, Cambridge, Massachusetts.

[157] **Noyce, R. N. (1979).** "Hardware Prospects and Limitations". In [156], pp. 321–337.

[158] **Cordella, L. P., Duff, M. J. B. and Levialdi, S. (1978).** "An Analysis of the Computational Cost in Image Processing". *IEEE Transactions on Computers* **C-27**, pp. 904–910.

[159] **Flynn, M. J. (1972).** "Some Computer Organisations and Their Effectiveness". *IEEE Transactions on Computers* **C-21**, pp. 948–960.

[160] **Shore, J. E. (1973).** "Second Thoughts on Parallel Processing". *Computers and Electrical Engineering* **1**, pp. 95–109.

[161] **Hockney, R. W. and Jesshope, C. R. (1981).** "Parallel Computers". Adam Hilger, Bristol.

[162] **Bell, C. G. and Newell, A. (1970).** "PMS and ISP Descriptive Systems". *AFIPS SJCC* **36**, pp. 351–374.

[163] **Haynes, L. S., Lau, R. L., Siewiorek, D. P. and Mizell, D. W. (1982).** "A Survey of Highly Parallel Computing". *IEEE Computer* **15(1)**, pp. 9–24.

[164] **IEEE Computer Society and IEEE Circuits and Systems Society (1984).** "IEEE International Conference on Computer Design: VLSI in Computers (Port Chester, New York)". IEEE Computer Society Press, Silver Spring, Maryland.

[165] **Basu, A. (1984).** "A Classification of Parallel Processing Systems". In [164], pp. 222–225.

[166] **Feng, T. (ed) (1975).** "Proceedings of the 1975 Sagamore Computer Conference on Parallel Processing (Sagamore, New York)". IEEE Computer Society Press, Long Beach, California.

[167] **Enslow, P. H. Jr. (1975).** "Multiprocessor Architecture — A Survey". In [166], pp. 63–70.

[168] **Pergamon Infotech (1976).** "Infotech State of the Art Report on Multiprocessor Systems". Pergamon Infotech, Maidenhead, U. K..

[169] **Enslow, P. H. (1976).** "Multiprocessors and Other Parallel Systems: An Introduction and Overview". In [168], pp. 219–261.

[170] **Enslow, P. H. Jr. (1977).** "Multiprocessor Organisation — A Survey". *Computing Surveys* **9**, pp. 103–129.

[171] **Genuys, F. (ed) (1968).** "Programming Languages". Academic Press, London and New York.

[172] **Dijkstra, E. W. (1968).** "Cooperating Sequential Processes". In [171], pp. 43–112.

[173] **IEEE and ACM (1979).** "Proceedings of the Sixth Annual Symposium on Computer Architecture". Published as *Computer Architecture News* **7(6)**.

[174] **Maekawa, M., Yamazaki, I., Maeda, A., Miyata, M., Kamiya, S. and Kasai, H. (1979).** "Experimental Polyprocessor System (EPOS) — Architecture". In [173], pp. 188–195.

[175] **Joel, A. E. Jr. (1979).** "Circuit Switching: Unique Architecture and Applications". *IEEE Computer* **12(6)**, pp. 10–22.

[176] **Masson, G. M., Gingher, G. C. (1979).** "A Sampler of Circuit Switching Networks". *IEEE Computer* **12(6)**, pp. 32–48.

[177] **Dennis, J. B. (1975).** "Packet Communication Architecture". In [166], pp. 224–229.

[178] **Lavington, S. H. (1976).** "Processor Architecture". National Computing Centre Press, Manchester.

[179] **Lorin, H. (1980).** "Aspects of Distributed Systems". John Wiley and Sons, New York.

[180] **Gosling, J. B. (1980).** "Design of Arithmetic Units for Digital Computers". Macmillan Press, London.

[181] **IEEE and ACM (1980).** "Proceedings of the Seventh Annual Symposium on Computer Architecture (La Baule, France)". Published as *Computer Architecture News* **8(3)**.

[182] **Edwards, D. G. B., Knowles, A. E. and Woods, J. V. (1980).** "MU6-G. A New Design to Achieve Mainframe Performance From a Mini-Sized Computer". In [181], pp. 161–167.

[183] **Agrawal, D. P. (1982).** "Testing and Fault Tolerance of Multistage Interconnection Networks". *IEEE Computer* **15(4)**, pp. 41–53.

[184] **Pease, M. C. (1977).** "The Indirect Binary N-Cube Microprocessor Array". *IEEE Transactions on Computers* **C-26**, pp. 458–473.

[185] **IEEE and ACM (1977).** "Proceedings of the Fourth Annual Symposium on Computer Architecture". Published as *Computer Architecture News* **5(7)**.

[186] **Siegel, H. J. (1977).** "The Universality of Various Types of SIMD Machine Interconnection Networks". In [185], pp. 70–79.

[187] **IEEE and ACM (1978).** "Proceedings of the Fifth Annual Symposium on Computer Architecture". Published as *Computer Architecture News* **6(5)**.

[188] **Siegel, H. J. and Smith, S. D. (1978).** "Study of Multistage SIMD Interconnection Networks". In [187], pp. 223–229.

[189] **Enslow, P. H. Jr. (ed) (1976).** "Proceedings of the 1976 International Conference on Parallel Processing (Waldenwood, Michigan)". IEEE Press, Piscataway, New Jersey.

[190] **Batcher, K. E. (1976).** "The Flip Network in STARAN". In [189], pp. 65–71.

[191] **Lawrie, D. H. (1975).** "Access and Alignment of Data in an Array Processor". *IEEE Transactions on Computers* **C-24**, pp. 1145–1155.

[192] **Patel, J. H. (1979).** "Processor-Memory Interactions for Multiprocessors". In [173], pp. 168–177.

[193] **Adams, G. B. and Siegel, H. J. (1982).** "On the Number of Permutations Performable on the Augmented Data Manipulator Network". *IEEE Transactions on Computers* **C-31**, pp. 270–277.

[194] **IEEE and ACM (1982).** "Proceedings of the Ninth Annual Symposium on Computer Architecture (Austin, Texas)". Published as *Computer Architecture News* **10(3)**.

[195] **McMillen, R. J. and Siegel, H. J. (1982).** "Performance and Fault Tolerance Improvements in the Inverse Augmented Data Manipulator". In [194], pp. 63–72.

[196] **Parker, D. S. and Raghavendra, C. S. (1982).** "The Gamma Network: A Multiprocessor Interconnection Network With Redundant Paths". In [194], pp. 73–80.

[197] **Lipovski, G. J. (1970).** "The Architecture of a Large Associative Processor". *AFIPS SJCC* **36**, pp. 385–396.

[198] **Lipovski, G. J. and Szygenda, S. A. (eds) (1973).** "Proceedings of the First Annual Symposium on Computer Architecture (Gainesville, Florida)". Published as *Computer Architecture News* **2(4)**.

[199] **Goke, L. R. and Lipovski, G. J. (1973).** "Banyan Networks for Partitioning Multiprocessor Systems". In [198], pp. 21–28.

[200] **Tripathi, A. R. and Lipovski, G. J. (1979).** "Packet Switching in Banyan Networks". In [173], pp. 160–167.

[201] **Premkumar, U. V., Kapur, R., Malek, M., Lipovski, G. J. and Horne, P. (1980).** "Design and Implementation of the Banyan Interconnection Network in TRAC". *AFIPS NCC* **49**, pp. 643–653.

[202] **Premkumar, U. V. and Browne, J. C. (1982).** "Resource Allocation in Rectangular SW Banyans". In [194], pp. 326–333.

[203] **IEEE and ACM (1975).** "Proceedings of the Second Annual Symposium on Computer Architecture". Published as *Computer Architecture News* **3(4)**.

[204] **Reames, C. C. and Ming, T. L. (1975).** "A Loop Network for Simoultaneous Transmission of Variable Length Messages". In [203], pp. 7–12.

[205] **Squire, J. S. and Palais, S. M. (1963).** "Programming and Design Considerations of a Highly Parallel Computer". *AFIPS SJCC* **23**, pp. 395–400.

[206] **Despain, A. M. and Patterson, D. A. (1978).** "X-tree: A Tree Structured Multi-Processor Computer Architecture". In [187], pp. 144–151.

[207] **Patterson, D. A., Fehr, E. S. and Sequin, C. H. (1979).** "Design Considerations for the VLSI Processor of X-TREE". In [173], pp. 90–101.

[208] **Siegel, H. J. (ed) (1980).** "Proceedings of the Workshop on Interconnection Networks for Parallel and Distributed Processing (Purdue University, Indiana)". IEEE Computer Society, Long Beach, California.

[209] **Horowitz, E. and Zorat, A. (1980).** "The Binary Tree as an Interconnection Network: Applications to Multiprocessor Systems and VLSI". In [208], pp. 1–10.

[210] **Siegel, H. J. (1979).** "Interconnection Networks for SIMD Machines". *IEEE Computer* **12(6)**, pp. 57–65.

[211] **Siegel, H. J., McMillen, R. J. and Mueller, P. T. Jr. (1979).** "A Survey of Interconnection Methods for Reconfigurable Parallel Processing Systems". *AFIPS NCC* **48**, pp. 529–542.

[212] **Siegel, H. J. (1985).** "Interconnection Networks for Large-Scale Parallel Processing". Lexington books (D. C. Heath), Lexington.

[213] **Stone, H. S. (1971).** "Parallel Processing With the Perfect Shuffle". *IEEE Transactions on Computers* **C-20**, pp. 153–161.

[214] **Lang, T. and Stone, H. S. (1976).** "A Shuffle-Exchange Network With Simplified Control". *IEEE Transactions on Computers* **C-25**, pp. 55–65.

[215] **Lawrie, D. H. and Padua, D. A. (1980).** "Analysis of Message Switching With Shuffle-Exchanges in Multiprocessors". In [208], pp. 116–123.

[216] **Riordan, J. (1958).** "An Introduction to Combinatorial Analysis". John Wiley and Sons, New York.

[217] **Anderson, I. (1974).** "A First Course in Combinatorial Mathematics". Clarendon Press, Oxford.

[218] **Digital Equipment Corporation (1981).** "PDP11 Processor Handbook". Digital Equipment Corporation, U.S.A..

[219] **Townsend, R. (1975).** "Digital Computer Structure and Design". Newnes-Butterworth, London.

[220] **Kilburn, T., Edwards, D. G. B., Lanigan, M. J. and Sumner, F. H. (1962).** "One-Level Storage System". *IRE Transactions on Electronic Computers* **EC-11**, pp. 223–235.

[221] **IEEE Computer Society (1972).** "Proceedings of the 1972 Annual Conference — Innovative Architecture — COMPCON 72 Digest". IEEE Press, New York.

[222] **Hintz, R. G. and Tate, D. P. (1972).** "Control Data STAR-100 Processor Design". In [221], pp. 1–4.

[223] **Ramamoorthy, C. V. and Li, H. F. (1977).** "Pipeline Architecture". *Computing Surveys* **9**, pp. 61–102.

[224] **Ibbett, R. (1982).** "The Architecture of High Performance Computers". Macmillan Press, London.

[225] **Lincoln, N. R. (1983).** "Supercomputers = Colossal Computations + Enormous Expectations + Renowned Risk". *IEEE Computer* **16(5)**, pp. 38–47.

[226] **Duff, M. J. B. and Levialdi, S. (eds) (1981).** "Languages and Architectures for Image Processing". Academic press, London.

[227] **Gerritsen, F. A. and Monhemius, R.D. (1981).** "Evaluation of the Delft Image processor DIP-1". In [226], pp. 189–203.

[228] **IEEE Computer Society (1980).** "Proceedings of the Workshop on Picture Data Description and Management (Asilomar, Pacific Grove, California)". IEEE Press, Los Alamitos, California.

[229] **Danielsson, P. E., Kruse, B. and Gudmundsson, B. (1980).** "Memory Hierarchies in PICAP II". In [228], pp. 275–280.

[230] **Antonsson, D., Danielsson, P. E., Gudmundsson, B., Hedblom, T., Kruse, B., Linge, A., Lord, P. and Ohlsson, T. (1981).** "PICAP — A System Approach to Image Processing". In [23], pp. 35–42.

[231] **Antonsson, D., Gudmundsson, B., Hedblom, T., Kruse, B., Linge, A., Lord, P. and Ohlsson, T. (1982).** "Picap II — A System Approach to Image Processing". *IEEE Transactions on Computers* **C-31**, pp. 997–1000.

[232] **Thornton, J. E. (1970).** "Design of a Computer: The Control Data 6600". Scott Foresman and Company, Glenview, Illinois.

[233] **Watson, W. J. (1972).** "The TI ASC — A Highly Modular and Flexible Super Computer Architecture". *AFIPS FJCC* **41**, pp. 221–228.

[234] **Watson, W. J. (1972).** "The Texas Instruments Advanced Scientific Computer". In [221], pp. 291–293.

[235] **Control Data Corporation (1981).** "CDC Cyber 200 Model 205 Computer System Hardware Reference Manual". CDC Publications and Graphics Division, St. Paul, Minnesota.

[236] **IEEE Computer Society (1976).** "Proceedings of the Third International Joint Conference on Pattern Recognition (Coronado, California)". IEEE Press, Long Beach, California.

[237] **Kruse, B. (1976).** "The PICAP Picture Processing Laboratory". In [236], pp. 875–881.

[238] **Kruse, B. (1978).** "Experience with a Picture Processor in Pattern Recognition Processing". *AFIPS NCC* **47**, pp. 1015–1024.

[239] **Association for Computing Machinery (1975).** "Proceedings of the Conference on Programming Languages Suitable for Parallel and Vector Machines". Published as *Sigplan Notices* **10(3)**.

[240] **Capon, P. C. and Ibbett, R. N. (1975).** "Array Operations in MU5". In [239], pp. 133–137.

[241] **Morris, D. and Ibbett, R. (1979).** "The MU5 Computer System". Macmillan Press, London.

[242] **Hobbs, L. C., Theis, D. J., Trimble, J., Titus, H. and Highberg, I. (eds) (1970).** "Parallel Processor Systems, Technologies, and Applications". Spartan books, New York.

[243] **Shooman, W. (1970).** "Orthogonal Processing". In [242], pp. 297–308.

[244] **Higbie, L. C. (1972).** "The Omen Computers: Associative Array Processors". In [221], pp. 287–290.

[245] **Reddaway, S. F. (1973).** "DAP — A Distributed Processor Array". In [198], pp. 61–65.

[246] **Pergamon Infotech (1979).** "Infotech State of the Art Report on Supercomputers, Volume 2". Pergamon Infotech, Maidenhead, U. K..

[247] **Reddaway, S. F. (1979).** "The DAP Approach". In [246], pp. 309–329.

[248] **Hunt, D. J. (1981).** "The ICL DAP and its Application to Image Processing". In [226], pp. 275–282.

[249] **Duff, M. J. B. (ed) (1983).** "Computing Structures for Image Processing". Academic Press, London.

[250] **Gerritsen, F. A. (1983).** "A Comparison of the CLIP4, DAP and MPP Processor Array Implementations". In [249], pp. 15–30.

[251] **Preston, K. Jr. (1983).** "Cellular Logic Computers for Pattern Recognition". *IEEE Computer* **16(1)**, pp. 36–47.

[252] **Githens, J. A. (1970).** "An Associative, Highly-Parallel Computer for Radar Data Processing". In [242], pp. 71–86.

[253] **Crane, B. A., Gilmartin, M. J., Huttenhoff, J. H., Rux, P.T. and Shively, R. R. (1972).** "PEPE Computer Architecture". In [221], pp. 57–60.

[254] **Wilson, D. E. (1972).** "The PEPE Support Software System". In [221], pp. 61–64.

[255] **Cornell, J. A. (1976).** "PEPE: Parallel Element Processing Ensemble". In [168], pp. 171–190.

[256] **Rudolph, J. A. (1972).** "A Production Implementation of an Associative Processor — STARAN". *AFIPS FJCC* **41**, pp. 229–241.

[257] **Meilander, W. C. (1976).** "STARAN, an Associative Approach to Multiprocessor Architecture". In [168], pp. 345–372.

[258] **Graham, M. D. and Norgren, P. E. (1980).** "The Diff3 Analyzer: A Parallel/Serial Golay Image Processor". In [12], pp. 163–182.

[259] **Golay, M. J. E. (1969).** "Hexagonal Pattern Transformations". *IEEE Transactions on Computers* **C-18**, pp. 733–740.

[260] **Barnes, G. H., Brown, R. M., Kato, M., Kuck, D. J., Slotnick, D. L. and Stokes, R. A. (1968).** "The ILLIAC IV Computer". *IEEE Transactions on Computers* **C-17**, pp. 746–757.

[261] **Davis, R. L. (1969).** "The ILLIAC IV Processing Element". *IEEE Transactions on Computers* **C-18**, pp. 800–816.

[262] **Bouknight, W. J., Denenberg, S. A., McIntyre, D. E., Randall, J. M., Sameh, A. H. and Slotnick, D. L. (1972).** "The Illiac IV System". *Proceedings of the IEEE* **60**, pp. 369–388.

[263] **Barnes, G. H. (1972).** "The Use of ILLIAC IV". In [221], pp. 295–296.

[264] **Duff, M. J. B. (1976).** "CLIP, an Array Processor for Image Processing". In [168], pp. 191–203.

[265] **Duff, M. J. B. (1976).** "CLIP4: A Large Scale Integrated Circuit Array Parallel Processor". In [236], pp. 728–733.

[266] **Fountain, T. J. and Goetcherian, V. (1980).** "CLIP4 Parallel Processing System". *IEE Proceedings* **127E**, pp. 219–224.

[267] **Fountain, T. J. (1981).** "CLIP4: A Progress Report". In [226], pp. 283–291.

[268] **Fountain, T. J. (1983).** "Bit-Serial Array Processor Circuits". In [249], pp. 1–14.

[269] **Batcher, K. E. (1980).** "Architecture of a Massively Parallel Processor". In [181], pp. 168–173.

[270] **Batcher, K. E. (1982).** "Bit-Serial Parallel Processing Systems". *IEEE Transactions on Computers* **C-31**, pp. 377–384.

[271] **Potter, J. L. (1983).** "Image Processing on the Massively Parallel Processor". *IEEE Computer* **16(1)**, pp. 62–67.

[272] **Preston, K. Jr. and Uhr, L. (eds) (1982).** "Multicomputers and Image Processing". Academic Press, New York.

[273] **Tanimoto, S. L. (1982).** "Programming Techniques for Hierarchical Parallel Image Processors". In [272], pp. 421–429.

[274] **Gelsema, E. S. and Kanal, L. N. (eds) (1980).** "Pattern Recognition in Practice". North-Holland Publishing Company, Amsterdam.

[275] **Sternberg, S. R. (1980).** "Language and Architecture for Parallel Image Processing". In [274], pp. 35–44.

[276] **Sternberg, S. R. (1980).** "Cellular Computers and Biomedical Image Processing". In [12], pp. 11–22.

[277] **Lougheed, R. M., McCubbrey, D. L. and Sternberg, S. R. (1980).** "Cytocomputers: Architectures for Image Processing". In [228], pp. 281–286.

[278] **Lougheed, R. M. and McCubbrey, D. L. (1980).** "The Cytocomputer: A Practical Pipelined Image Processor". In [181], pp. 271–278.

[279] **Snyder, L., Jamieson, L. J., Gannon, D. B. and Siegel, H. J. (eds) (1985).** "Algorithmically Specialised Parallel Computers: Proceedings of the Workshop on Algorithmically Specialised Computer Organisations (Purdue University, September 1982)". Academic Press, Orlando, Florida.

[280] **Sternberg, S. R. (1985).** "Computer Architectures Specialised for Mathematical Morphology". In [279], pp. 169–176.

[281] **Gerritsen, F. A. and Aardema, L. G. (1981).** "Design and Use of DIP-1: A Fast, Flexible and Dynamically Microprogrammable Pipelined Image Processor". *Pattern Recognition* **14**, pp. 319–330.

[282] **Kung, H. T. and Picard, R. L. (1981).** "Hardware Pipelines for Multi-Dimensional Convolution and Resampling". In [23], pp. 273–278.

[283] **Kung, H.T. (1982).** "Why Systolic Architectures?". *IEEE Computer* **15(1)**, pp. 37–46.

[284] **Kuhn, R. H. (1980).** "Transforming Algorithms for Single-Stage and VLSI Architectures". In [208], pp. 11–19.

[285] **Yen, D. W. L. and Kulkarni, A. V. (1981).** "The ESL Systolic Array Processor for Signal and Image Processing". In [23], pp. 265–272.

[286] **Kulkarni, A. V. and Yen, D. W. L. (1982).** "Systolic Processing and an Implementation for Signal and Image Processing". *IEEE Transactions on Computers* **C-31**, pp. 1000–1009.

[287] **Ahmed, H. M., Delosme, J. M. and Morf, M. (1982).** "Highly Concurrent Computing Structures for Matrix Arithmetic and Signal Processing". *IEEE Computer* **15(1)**, pp. 65–86.

[288] **Chuang, H. Y. H. and He, G. (1984).** "Design of Problem-Size Independent Systolic Array Systems". In [164], pp. 152–157.

[289] **Shapiro, E. Y. (1983).** "The Bagel: A Systolic Concurrent Prolog Machine". ICOT Research Centre Technical Memorandum TM-0031.

[290] **Rieger, C., Bane, J. and Trigg, R. (1980).** "ZMOB: A Highly Parallel Multiprocessor". In [228], pp. 298–304.

[291] **Rieger, C. (1980).** "ZMOB: Doing it in Parallel". In [23], pp. 119–124.

[292] **Bane, R., Stanfill, C. and Weiser, M. (1981).** "Operating System Strategy on ZMOB". In [23], pp. 125–132.

[293] **Manara, R. and Stringa, L. (1981).** "The EMMA System: An Industrial Experience on a Multiprocessor". In [226], pp. 215–227.

[294] **Siegel, L. J. (1981).** "Image Processing on a Partitionable SIMD Machine". In [226], pp. 294–300.

[295] **Kuehn, J. T. and Siegel, H. J. (1981).** "Simulation Studies of PASM in SIMD Mode". In [23], pp. 43–50.

[296] **Siegel, L. J., Siegel, H. J. and Feather, A. E. (1982).** "Parallel Approaches to Image Correlation". *IEEE Transactions on Computers* **C-31**, pp. 208–218.

[297] **Warpenburg, M. R. and Siegel, L. J. (1982).** "SIMD Image Resampling". *IEEE Transactions on Computers* **C-31**, pp. 934–942.

[298] **Siegel, H. J., Schwederski, T., Davis, N. J. IV and Kuehn, J. T. (1984).** "PASM: A Reconfigurable Parallel System for Image Processing". *Computer Architecture News* **12(4)**, pp. 7–19.

[299] **Treleaven, P. C. (1982).** "VLSI Processor Architectures". *IEEE Computer* **15(6)**, pp. 33–45.

[300] **Aspinall, D. (ed) (1978).** "The Microprocessor and Its Application". Cambridge University Press, Cambridge.

[301] **Barron, I. M. (1978).** "The Transputer". In [300], pp. 343–357.

[302] **Inmos Limited (1984).** "IMS T424 Transputer". Product information, Inmos Limited, Bristol.

[303] **IEEE and ACM (1985).** "Proceedings of the Twelfth Annual Symposium on Computer Architecture". Published as *Computer Architecture News* **13(3)**.

[304] **Whitby-Strevens, C. (1985).** "The Transputer". In [303], pp. 292–300.

[305] **Kung, S. Y., Arun, K. S., Gal-Ezer, R. J. and Rao, D. V. B. (1982).** "Wavefront Array Processor: Language, Architecture and Applications". *IEEE Transactions on Computers* **C-31**, pp. 1054–1066.

[306] **Metcalfe, R. M. and Boggs, D. R. (1976).** "Ethernet: Distributed Packet-Switching for Local Computer Networks". *Communications of the ACM* **19**, pp. 395–403.

[307] **Shepherd, W. D., Blair, G. S. and Hutchison, D. (1982).** "Comparison of an Ethernet-Like Communication System with the Cambridge Ring". *IEE Proceedings* **129E**, pp. 147–155.

[308] **Hillis, W. D. (1981).** "The Connection Machine (Computer Architecture for the New Wave)". Massachusetts Institute of Technology, Artificial Intelligence Laboratory A. I. Memorandum 646.

[309] **Hillis, W. D. (1985).** "The Connection Machine". MIT Press, Cambridge, Massachusetts.

[310] **Hillis, W. D. (1987).** "The Connection Machine". *Scientific American* **256(6)**, pp. 86–93.

[311] **Feldman, J. A. and Ballard, D. H. (1982).** "Connectionist Models and Their Properties". *Cognitive Science* **6**, pp. 205–254.

[312] **Ackley, D. H., Hinton, G. E. and Sejnowski, T. J. (1985).** "A Learning Algorithm for Boltzmann Machines". *Cognitive Science* **9**, pp. 147–169.

[313] **Joshi, A. (ed) (1976).** "Proceedings of the Ninth International Joint Conference on Artificial Intelligence (Los Angeles, California), Volume 1". Morgan Kaufmann Publishers, Los Altos, California.

[314] **Touretzky, D. S. and Hinton, G. E. (1985).** "Symbols Among the Neurons: Details of a Connectionist Inference Architecture". In [313], pp. 238–243.

[315] **Granlund, G. H. (1981).** "GOP: A Fast and Flexible Processor for Image Analysis". In [226], pp. 179–188.

[316] **Granlund, G. H., Antonsson, D., Arvidsson, J., Hedlund, M., Henden, P., Knutsson, H., Lundgren, K., Nilsson, B., von Post, B. and Wilson, R. (1981).** "The GOP Image Processor". In [23], pp. 195–200.

[317] **Dyer, C. R. (1982).** "Pyramid Algorithms and Machines". In [272], pp. 409–420.

[318] **Levialdi, S. (ed) (1985).** "Integrated Technology for Parallel Image Processing". Academic Press, London.

[319] **Cantoni, V., Ferretti, M., Levialdi, S. and Maloberti, F. (1985).** "A Pyramid Project Using Integrated Technology". In [318], pp. 121–132.

[320] **International Association for Pattern Recognition (1986).** "Proceedings of the Eighth International Conference on Pattern Recognition (Paris, France), Volume 2". IEEE Computer Society Press, Piscataway, New Jersey.

[321] **Merigot, A., Garda, P., Zavidovique, B. and Devos, F. (1986).** "Interlayer Communication in MIMD Pyramidal Computer". In [320], pp. 954–957.

[322] **Uhr, L. (1981).** "Converging Pyramids of Arrays". In [23], pp. 31–34.

[323] **Uhr, L. (1985).** "Pyramid Multi-Computers, and Extensions and Augmentations". In [279], pp. 177–186.

[324] **Schwartz, J. T. (1980).** "Ultracomputers". *ACM Transactions on Programming Languages and Systems* **2**, pp. 484–521.

[325] **Gottlieb, A., Grishman, R., Kruskal, C. P., McAuliffe, K. P., Rudolph, L. and Snir, M. (1982).** "The NYU Ultracomputer — Designing a MIMD, Shared-Memory Parallel Machine". In [194], pp. 27–42.

[326] **Gottlieb, A., Grishman, R., Kruskal, C. P., McAuliffe, K. P., Rudolph, L. and Snir, M. (1983).** "The NYU Ultracomputer — Designing an MIMD Shared Memory Parallel Computer". *IEEE Transactions on Computers* **C-32**, pp. 175–189.

[327] **Edler, J., Gottlieb, A., Kruskal, C. P., McAuliffe, K. P., Rudolph, L., Snir, M., Teller, P. J. and Wilson, J. (1985).** "Issues Related to MIMD Shared-Memory Computers: the NYU Ultracomputer Approach". In [303], pp. 126–135.

[328] **Wulf, W. A. and Bell, C. G. (1972).** "C.mmp: A Multi-Mini-Processor". *AFIPS FJCC* **41**, pp. 765–777.

[329] **Swan, R. J., Fuller, S. H. and Siewiorek, D. P. (1977).** "Cm* — A Modular Multi-Microprocessor". *AFIPS NCC* **46**, pp. 637–663.

[330] **Swan, R. J., Bechtolsheim, A., Lai, K. W. and Ousterhout, J. K. (1977).** "The Implementation of the Cm* Multi-Microprocessor". *AFIPS NCC* **46**, pp. 645–655.

[331] **Jones, A. K., Chansler, R. J. Jr., Durham, I., Feiler, P. and Schwans, K. (1977).** "Software Management of Cm* — A Distributed Multiprocessor". *AFIPS NCC* **46**, pp. 657–663.

[332] **Deminet, J. (1982).** "Experience with Multiprocessor Algorithms". *IEEE Transactions on Computers* **C-31**, pp. 278–288.

[333] **Guzmán, A. (1981).** "A Parallel Heterarchical Machine for High-Level Language Processing". In [226], pp. 229–244.

[334] **Guzmán, A. (1981).** "A Heterarchical, Multi-Microprocessor Lisp Machine". In [23], pp. 309–317.

[335] **Sullivan, H. and Bashkov, T. (1977).** "A Large Scale, Homogeneous, Fully Distributed Parallel Machine, I". In [185], pp. 105–117.

[336] **Sullivan, H., Bashkov, T. and Klappholz, D. (1977).** "A Large Scale, Homogeneous, Fully Distributed Parallel Machine, II". In [185], pp. 118–124.

[337] **Visual Machines Limited (1985).** "C-VAS 3000 Computer Vision System". Product information, Visual Machines Limited, Manchester.

[338] **Briggs, F. A., Fu, K. S., Hwang, K. and Wah, B. W. (1982).** "PUMPS Architecture for Pattern Analysis and Database Management". *IEEE Transactions on Computers* **C-31**, pp. 969–983.

[339] **Association for Computing Machinery (1981).** "Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture (Portsmouth, New Hampshire)". ACM, New York.

[340] **Darlington, J. and Reeve, M. (1981).** "ALICE: A Multi-Processor Reduction Machine for the Parallel Evaluation of Applicative Languages". In [339], pp. 65–75.

[341] **Reeve, M. J. (1982).** "The ALICE Compiler Target Language". Department of Computing Internal Report, Imperial College, London.

[342] **Watson, I. and Gurd, J. R. (1979).** "A Prototype Dataflow Computer With Token Labelling". *AFIPS NCC* **48**, pp. 623–628.

[343] **Gurd, J. R. and Watson, I. (1980).** "Data Driven System for High Speed Computing — Part 1: Structuring Software for Parallel Execution". *Computer Design* **19(6)**, pp. 91–100.

[344] **Gurd, J. R. and Watson, I. (1980).** "Data Driven System for High Speed Computing — Part 2: Hardware Design". *Computer Design* **19(7)**, pp. 97–106.

[345] **da Silva, J. G. D. and Woods, J. V. (1981).** "Design of a Processing Subsystem for the Manchester Data-Flow Computer". *IEE Proceedings* **127E**, pp. 218–224.

[346] **Watson, I. and Gurd, J. R. (1982).** "A Practical Data Flow Computer". *IEEE Computer* **15(2)**, pp. 51–57.

[347] **Gurd, J. R., Kirkham, C. C. and Watson, I. (1985).** "The Manchester Prototype Dataflow Computer". *Communications of the ACM* **28**, pp. 34–52.

[348] **Dennis, J. B. and Misunas, D. P. (1975).** "A Preliminary Architecture for a Basic Data-Flow Processor". In [203], pp. 126–132.

[349] **Misunas, D. P. (1975).** "Structure Processing in a Data-Flow Computer". In [166], pp. 230–235.

[350] **Farrel, E. P., Ghani, N. and Treleaven, P. C. (1979).** "A Concurrent Computer Architecture and a Ring Based Implementation". In [173], pp. 1–11.

[351] **Batchelor, B. G. (ed) (1978).** "Pattern Recognition. Ideas in Practice". Plenum Press, New York.

[352] **Aleksander, I. (1978).** "Pattern Recognition with Networks of Memory Elements". In [351], pp. 43–64.

[353] **Aleksander, I., Thomas, W. V. and Bowden, P. A. (1984).** "WISARD — A Radical Step Forward in Image Recognition". *Sensor Review,* July 1984.

[354] **Potter, J. L. (1978).** "The STARAN Architecture and Its Application to Image Processing and Pattern Recognition Algorithms". *AFIPS NCC* **47**, pp. 1041–1047.

[355] **Kushner, T., Wu, A. Y. and Rosenfeld, A. (1981).** "Image Processing on ZMOB". In [23], pp. 88–95.

[356] **Kushner, T., Wu, A. Y. and Rosenfeld, A. (1982).** "Image Processing on ZMOB". *IEEE Transactions on Computers* **C-31**, pp. 943–951.

[357] **Gajski, D. D., Padua, D. A., Kuck, D. J. and Kuhn, R. H. (1982).** "A Second Opinion on Data Flow Machines and Languages". *IEEE Computer* **15(2)**, pp. 58–69.

[358] **King, T. and Knight, B. (1983).** "Programming the M68000". Addison-Wesley Publishing Company, London.

[359] **Starnes, T. W. (1983).** "Design Philosophy Behind Motorola's MC68000 — Part 1: A 16-bit Processor With Multiple 32-bit Registers". *Byte* **8(4)**, pp. 70–92.

[360] **Starnes, T. W. (1983).** "Design Philosophy Behind Motorola's MC68000 — Part 2: Data-Movement, Arithmetic, and Logic Instructions". *Byte* **8(5)**, pp. 342–367.

[361] **Starnes, T. W. (1983).** "Design Philosophy Behind Motorola's MC68000 — Part 3: Advanced Instructions". *Byte* **8(6)**, pp. 339–349.

[362] **Jaulent, P. (1985).** "The 68000 Hardware and Software". Macmillan Publishers Limited, Basingstoke.

[363] **Witten, I. H. (1981).** "The New Microprocessors". *IEE Proceedings* **127E**, pp. 197–204.

[364] **Zingale, T. (1983).** "Intel's 80186: A 16-Bit Computer on a Chip". *Byte* **8(4)**, pp. 132–150.

[365] **Leedy, G. (1983).** "The National Semiconductor NS16000 Microprocessor Family". *Byte* **8(4)**, pp. 53–66.

[366] **Benes, V. E. (1965).** "Mathematical Theory of Connecting Networks and Telephone Traffic". Academic Press, New York.

[367] **Darlington, J., Henderson, P. and Turner, D. A. (eds) (1982).** "Functional Programming and its Applications". Cambridge University Press, Cambridge.

[368] **Treleaven, P. C. (1982).** "Computer Architecture for Functional Programming". In [367], pp. 281–306.

[369] **McCarthy, J., Abrahams, P. W., Edwards, D. J., Hart, T. P. and Levin, M. I. (1965).** "LISP 1.5 Programmers Manual (Second Edition)". M. I. T. Press, Cambridge, Massachusetts.

[370] **Backus, J. (1977).** "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs". *Communications of the ACM* **21**, pp. 613–641.

[371] **Turner, D. A. (1979).** "A New Implementation Technique for Applicative Languages". *Software Practice and Experience* **9**, pp. 31–49.

[372] **Henderson, P. (1980).** "Functional Languages". Prentice-Hall, Englewood Cliffs, New Jersey.

[373] **Berkling, K. J. (1975).** "Reduction Languages for Reduction Machines". In [203], pp. 133–140.

[374] **Church, A. (1941).** "The Calculi of Lambda Conversion". Princeton University Press, Princeton.

[375] **Berkling, K. J. (1978).** "Computer Architecture for Correct Programming". In [187], pp. 78–84.

[376] **Association for Computing Machinery (1976).** "Conference Record of the Third Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages (Atlanta, Georgia)". ACM, New York.

[377] **Henderson, P. and Morris, J. H. (1976).** "A Lazy Evaluator". In [376], pp. 95–103.

[378] **Ackerman, W. B. (1982).** "Data Flow Languages". *IEEE Computer* **15(2)**, pp. 15–25.

[379] **Davis, A. L. and Keller, R. M. (1982).** "Data Flow Program Graphs". *IEEE Computer* **15(2)**, pp. 26–41.

[380] **Kluge, W. E. and Schlüter, H. (1983).** "Petri Net Models for the Evaluation of Applicative Programs Based on λ-Expressions". *IEEE Transactions on Software Engineering* **SE-9**, pp. 415–427.

[381] **Skedzielewski, S. (1983).** "IF1: An Intermediate Form for Applicative Languages". Department of Computer Science Internal Report, University of Manchester.

[382] **Adams, D. A. (1970).** "A Model for Parallel Computation". In [242], pp. 311–333.

[383] **Garcia, O. N. (ed) (1979).** "Proceedings of the 1979 International Conference on Parallel Processing (Bellaire, Michigan)". IEEE Computer Society, Long Beach, California.

[384] **Kuck, D. J. and Padua, D. A. (1979).** "High Speed Multiprocessors and Their Compilers". In [383], pp. 5–16.

[385] **Padua, D. A., Kuck, D. J. and Lawrie, D. H. (1980).** "High Speed Multiprocessors and Compilation Techniques". *IEEE Transactions on Computers* **C-29**, pp. 763–776.

[386] **Doroshenko, A. E. (1983).** "Conversion of Cyclic Operators to a Parallel Form". *Cybernetics* **18(4)**, pp. 429-435. (Translation of *Kibernetika* **18(4)**, pp. 22-28).

[387] **Uhr, L. (1981).** "A Language for Parallel Processing of Arrays, Embedded in PASCAL". In [226], pp. 53–87.

[388] **Smith, K. A. (1983).** "DIMP: DAP Image Manipulation Package". DAP Support Unit Internal Report, Queen Mary College, London.

[389] **Paddon, D. J. (ed) (1984).** "Supercomputers and Parallel Computation". Oxford University Press, Oxford.

[390] **Davies, S. T. (1984).** "The Implementation of the FFT on the DAP". In [389], pp. 195–208.

[391] **Brinch Hansen, P. (1975).** "The Programming Language Concurrent Pascal". *IEEE Transactions on Software Engineering* **SE-1**, pp. 199–207.

[392] **Brinch Hansen, P. (1978).** "Multiprocessor Architectures for Concurrent Programs". *Computer Architecture News* **7(4)**, pp. 4–23.

[393] **Wirth, N. (1971).** "The Design of a PASCAL Compiler". *Software Practice and Experience* 1, pp. 309–333.

[394] **Wilson, I. R. and Addyman, A. M. (1982).** "A Practical Introduction to Pascal — with BS6192". Macmillan Publishers Limited, Basingstoke.

[395] **Jensen, K. and Wirth, N. (1985).** "PASCAL: User Manual and Report: Third Edition". Springer-Verlag, New York.

[396] **Gottlieb, A. and Kruskal, C. P. (1981).** "Coordinating Parallel Processors: A Partial Unification". *Computer Architecture News* **9(6)**, pp. 16–24.

[397] **Lister, A. M. (1979).** "Fundamentals of Operating Systems (Second Edition)". Macmillan Press, London.

[398] **Gottlieb, A. and Schwartz, J. T. (1982).** "Networks and Algorithms for Very-Large-Scale Parallel Computation". *IEEE Computer* **15(1)**, pp. 27–36.

[399] **IEEE Computer Society (1980).** "Proceedings of the 1980 International Conference on Parallel Processing (Harbor Springs, Michigan)". IEEE Press, Piscataway, New Jersey.

[400] **Deo, N., Pang, C. Y. and Lord, R. E. (1980).** "Two Parallel Algorithms for Shortest Path Problems". In [399], pp. 244–253.

[401] **Rohl, J. S. (1975).** "An Introduction to Compiler Writing". Macdonald and Jane's, London.

[402] **Aho, A. V. and Ullman, J. D. (1977).** "Principles of Compiler Design". Addison-Wesley, Reading, Massachusetts.

[403] **Rustin, R. (ed) (1971).** "Design and Optimisation of Compilers". Prentice-Hall, Englewood Cliffs, New Jersey.

[404] **Traub, J. F. (ed) (1976).** "Algorithms and Complexity — New Directions and Recent Results". Academic Press, New York..

[405] **Tarjan, R. E. (1976).** "Iterative Algorithms for Global Flow Analysis". In [404], pp. 71–101.

[406] **Hanson, D. R. (1983).** "Simple Code Optimisations". *Software Practice and Experience* **13**, pp. 745–763.

[407] **Pollack, B. W. (ed) (1979).** "Proceedings of the Sixth Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 79)". Published as *Computer Graphics* **13(2)**.

[408] **Smith, A. R. (1979).** "Tint Fill". In [407], pp. 276–283.

[409] **Rogers, D. F. (1985).** "Procedural Elements for Computer Graphics". McGraw-Hill Book Company, Singapore.

[410] **Knuth, D. E. (1983).** "The Art of Computer Programming, Volume 3 — Sorting and Searching". Addison Wesley, Reading, Mass..

[411] **Chen, J., Dagless, E. L. and Guo, Y. (1984).** "Performance Measurements of Scheduling Strategies and Parallel Algorithms for a Multiprocessor Quick Sort". *IEE Proceedings* **127E**, pp. 45–54.

[412] **Wettstein, H. (1981).** "Locking Operations for Maximum Concurrency". *The Computer Journal* **24**, pp. 243–248.

Appendix 1:

Calculation of Mean Latencies for Interconnection Networks

This appendix contains the detailed derivation of the expressions for mean latencies of networks quoted in chapters 5 and 8.

A1.1 The Unidirectional Ring

Consider a unidirectional ring network, connecting p source units to p destination units. Since this network is symmetric, the mean latency (between all possible pairs of units) is equal to the mean of the latencies from any single specified unit to all others. The latency from this unit to itself is not included in this calculation, for reasons stated in chapter 5. If the latency from unit zero to unit i is L_i, then the sum of latencies, S_l, is given by

$$S_l = \sum_{i=1}^{p-1} L_i.$$

Since, for this simple network, $L_i = i$,

$$\begin{aligned} S_l &= \sum_{i=1}^{p-1} i \\ &= \frac{p}{2}(p-1), \end{aligned}$$

and the mean latency, $\mathrm{L_{mean}}$, is given by

$$\begin{aligned} \mathrm{L_{mean}} &= \frac{S_l}{p-1} \\ &= \frac{p}{2}. \end{aligned}$$

A1.2 The Bidirectional Ring

The bidirectional ring is, again, symmetric, and so the overall mean latency may be calculated as the mean latency from unit zero to all other units. The latency for this network must be calculated separately for rings containing odd and even numbers of units. If p is odd, there are $\frac{p-1}{2}$ units which may be most efficiently reached by passing data in the clockwise direction, and $\frac{p-1}{2}$ in the anticlockwise direction. The sum of latencies, S_l, is given by

$$\begin{aligned} S_l &= 2\sum_{i=1}^{\frac{p-1}{2}} i \\ &= 2\left(\frac{1}{2}\left(\frac{p-1}{2}\right)\left(\frac{p+1}{2}\right)\right) \\ &= \frac{1}{4}(p-1)(p+1). \end{aligned}$$

The mean latency, L_{mean}, is given by

$$\begin{aligned} L_{mean} &= \frac{S_l}{p-1} \\ &= \frac{1}{4}(p+1). \end{aligned}$$

If p is even, $\frac{p}{2}-1$ units may be most efficiently reached by passing data in the clockwise direction, $\frac{p}{2}-1$ in the anticlockwise direction, and the single unit in the diametrically opposing position has equal latency in each direction. The sum of latencies, S_l, is given in this case by

$$\begin{aligned} S_l &= 2\sum_{i=1}^{\frac{p}{2}-1} i + \frac{p}{2} \\ &= 2\left(\frac{1}{2}\left(\frac{p}{2}-1\right)\left(\frac{p}{2}-1+1\right)\right) + \frac{p}{2} \\ &= \frac{p^2}{4} - \frac{p}{2} + \frac{p}{2} \\ &= \frac{p^2}{4}, \end{aligned}$$

and the mean latency, L_{mean}, is given by

$$\begin{aligned} L_{mean} &= \frac{S_l}{p-1} \\ &= \frac{p^2}{4(p-1)}. \end{aligned}$$

The mean latency for the bidirectional ring network may, therefore, be expressed as

$$L_{mean}(Y) = \begin{cases} \frac{1}{4}(p+1), & \text{if } n \text{ is odd,} \\ \frac{p^2}{4(p-1)}, & \text{if } n \text{ is even.} \end{cases}$$

A1.3 The Binary n-Cube

In chapter 5, the binary n-tube network was defined as a single-stage network which connects 2^n inputs to 2^n outputs in such a way that, if the n-bit number i is represented as a binary number $b_{n-1}\ldots b_2b_1b_0$, then each input, s_i, may be connected to its corresponding output, d_i, or to one of the outputs whose binary representation is $b_{n-1}\ldots\overline{b_k}\ldots b_2b_1b_0$, for any k $(0 \leq k < n)$. Since this network is, again, symmetric, the overall mean latency may be calculated as the mean latency from unit zero to each of the other destination units. The number of units which have a latency of one may be seen, from the above definition, to be equal to the number of different ways in which one digit may be inverted in the n-digit binary representation of the unit number.

$$\begin{aligned} N_1 &= \binom{n}{1} \\ &= \frac{n!}{(n-1)!\ 1!} \\ &= n. \end{aligned}$$

Similarly, the number of units which have a latency of two is equal to the number of different ways in which any two of the n digits may be inverted.

$$N_2 = \binom{n}{2}$$

$$= \frac{n!}{(n-2)!\ 2!}$$
$$= \frac{n}{2}(n-1).$$

This may be generalised, so that the number of units with latency d is

$$N_d = \binom{n}{d}$$
$$= \frac{n!}{(n-d)!\ d!}$$

The sum of latencies may now be formed:

$$S_l = \sum_{d=1}^{n} dN_d$$
$$= \sum_{d=1}^{n} \frac{dn!}{(n-d)!\ d!}$$
$$= \sum_{d=1}^{n} \frac{n!}{(n-d)!\ (d-1)!}.$$

If $i = d-1$,

$$S_l = \sum_{i=0}^{n-1} \frac{n!}{(n-(i+1))!\ i!}$$
$$= n\sum_{i=0}^{n-1} \frac{(n-1)!}{((n-1)-i)!\ i!}$$
$$= n\sum_{i=0}^{n-1} \binom{n-1}{i}.$$

Since the binomial theorem states that

$$(a+b)^k = \sum_{j=0}^{k} \binom{k}{j} a^{k-j}b^j,$$

setting both a and b to one, and reversing the equation, gives

$$\sum_{j=0}^{k} \binom{k}{j} = 2^k.$$

Thus, the sum of latencies may be expressed as

$$S_l = n2^{n-1}.$$

Therefore, the mean latency for the binary n-cube network,

$$\mathrm{L_{mean}} = \frac{S_l}{(2^n-1)}$$
$$= \frac{n2^{n-1}}{2^n-1}.$$

A1.4 The Partitioned Indirect Binary n-Cube

Consider a partitioned indirect binary n-cube network composed of 2^n rings, each of n units. The total number of units, $p = n2^n$. The network is symmetric, and so the overall mean latency may be

calculated as the mean latency from unit zero to all other units. This source unit connects directly to two destination units whose latencies are, therefore, one. These, in turn, connect to four units for which $d = 2$. This process of expansion continues until the 2^{n-1} units for which $d = n - 1$. The remaining units are those which are unreachable in a single circuit around the network. The first rank of these, with a latency $d = w$, contains $2^n - 1$ units. Subsequent ranks contain $2^n - 2$, $2^n - 4, \ldots,$ $2^n - 2^{n-1}$ units. Thus, the number of units, N_d, which have a latency, d, may be expressed as

$$N_d = \begin{cases} 2^d & \text{if } 1 \leq d \leq n-1; \\ 2^n - 2^{d-n} & \text{if } n \leq d \leq 2n-1. \end{cases}$$

The sum of latencies, S_l, may, therefore, be formed as the sum of two terms:

$$\begin{aligned} S_l &= \sum_{d=1}^{n-1} d2^d + \sum_{d=n}^{2n-1} d(2^n - 2^{d-n}) \\ &= \sum_{d=0}^{n-1} d2^d + \sum_{d=0}^{n-1} (d+n)(2^n - 2^d) \\ &= \sum_{d=0}^{n-1} d2^d + \sum_{d=0}^{n-1} (d2^n + n2^n - d2^d - n2^d) \\ &= \sum_{d=0}^{n-1} d2^d + \sum_{d=0}^{n-1} d2^n + \sum_{d=0}^{n-1} n2^n - \sum_{d=0}^{n-1} d2^d - \sum_{d=0}^{n-1} n2^d \\ &= \sum_{d=0}^{n-1} d2^n + \sum_{d=0}^{n-1} n2^n - \sum_{d=0}^{n-1} n2^d \\ &= 2^n \sum_{d=0}^{n-1} d + n2^n \sum_{d=0}^{n-1} 1 - n \sum_{d=0}^{n-1} 2^d. \end{aligned}$$

Expanding,

$$\begin{aligned} S_l &= 2^n \left(\frac{1}{2}(n-1)n \right) + 2^n n^2 - n(2^n - 1) \\ &= 2^{n-1} n(n-1) + 2^n n^2 - 2^n n + n \\ &= 2^{n-1} n(n-1) + 2^n n(n-1) + n \\ &= (2^{n-1} + 2^n) n(n-1) + n \\ &= 3 \times 2^{n-1} n(n-1) + n. \end{aligned}$$

Therefore, the mean latency for the partitioned indirect binary n-cube network,

$$\begin{aligned} \mathrm{L}_{\mathrm{mean}} &= \frac{S_l}{(p-1)} \\ &= \frac{3 \times 2^{n-1} n(n-1) + n}{n2^n - 1}. \end{aligned}$$

A1.5 The Partitioned Indirect Binary n-Tube

Consider an R-repeated partitioned indirect binary n-tube network composed of 2^n rings, each of Rn units. The total number of units, $p = Rn2^n$. As in the the case of the partitioned indirect binary n-cube, the source unit connects to two other units whose latencies are one. These, again, connect to four units for which $d = 2$, and the process of expansion continues to the 2^{n-1} units for which $d = n-1$. In this case, however, there follows a series of ranks in which the latencies are the same for all 2^n units

of the rank. The remaining units are, once more, those which are unreachable in a single circuit around the network. The first rank of these, with a latency $d = Rn$, contains $2^n - 1$ units. Subsequent ranks contain $2^n - 2,\ 2^n - 4, \ldots,\ 2^n - 2^{n-1}$ units. Thus, the number of units N_d with a latency d may be expressed as

$$N_d = \begin{cases} 2^d & \text{if } 1 \le d \le n-1; \\ 2^n & \text{if } n \le d \le Rn-1; \\ 2^n - 2^{d-Rn} & \text{if } Rn \le d \le (R+1)n-1. \end{cases}$$

The sum of latencies may now be formed as the sum of three terms:

$$\begin{aligned} S_l &= \sum_{d=1}^{n-1} d2^d + \sum_{d=n}^{Rn-1} d2^n + \sum_{d=Rn}^{(R+1)n-1} d(2^n - 2^{d-Rn}) \\ &= \sum_{d=0}^{n-1} d2^d + 2^n \sum_{d=n}^{Rn-1} d + \sum_{d=Rn}^{(R+1)n-1} d2^n - \sum_{d=Rn}^{(R+1)n-1} 2^{d-Rn} \\ &= \sum_{d=0}^{n-1} d2^d + 2^n \left(\sum_{d=0}^{Rn-1} d - \sum_{d=0}^{n-1} d \right) + 2^n \sum_{d=Rn}^{(R+1)n-1} d - \sum_{d=0}^{n-1} (d + Rn)2^d \\ &= \sum_{d=0}^{n-1} d2^d + 2^n \sum_{d=0}^{Rn-1} d - 2^n \sum_{d=0}^{n-1} d + \left(2^n \sum_{d=0}^{(R+1)n-1} d - 2^n \sum_{d=0}^{Rn-1} d \right) - \left(\sum_{d=0}^{n-1} d2^d + Rn \sum_{d=0}^{n-1} 2^d \right) \\ &= 2^n \sum_{d=0}^{(R+1)n-1} d - 2^n \sum_{d=0}^{n-1} d - Rn \sum_{d=0}^{n-1} 2^d. \end{aligned}$$

Expanding,

$$\begin{aligned} S_l &= 2^n \left(\frac{1}{2}\big((R+1)n-1\big)\big((R+1)n\big) \right) - 2^n \left(\frac{1}{2}(n-1)n \right) - Rn(2^n - 1) \\ &= n2^{n-1}\big((Rn+n-1)(R+1) - (n-1) - 2R\big) + Rn \\ &= n2^{n-1}(R^2n + Rn - R + Rn + n - 1 - n + 1 - 2R) + Rn \\ &= n2^{n-1}(R^2n + 2Rn - 3R) + Rn \\ &= Rn2^{n-1}(Rn + 2n - 3) + Rn \\ &= Rn\big(1 + 2^{n-1}(Rn + 2n - 3)\big). \end{aligned}$$

Therefore, the mean latency for the partitioned indirect binary n-tube network,

$$\begin{aligned} \mathrm{L_{mean}} &= \frac{S_l}{(p-1)} \\ &= \frac{Rn}{Rn2^n - 1}\big(1 + 2^{n-1}(Rn + 2n - 3)\big). \end{aligned}$$

Appendix 2:

A Survey of Parallel Machines

This appendix contains brief descriptions of a number of sequential and parallel machines. These are classified according to the scheme presented in chapter 6 and, within these groups, are ordered chronologically. Where more than one date is given, these are the dates of design commencement and completion of commissioning, respectively. The specified dates are, however, only approximate.

A2.1 Sequential (Von Neumann) Machines

A2.1.1 Cellscan

Perkin-Elmer Corporation / Navigation Computer Corporation (1961).

Cellscan was a machine specialised to perform 3×3 neighbourhood convolutions on images. Images were divided into 63-pixel wide slices before processing serially. Shift registers were used to store two lines of input image and a line of the output image, to allow the ALU to access a 3×3 window and, thus, produce a bit-serial output image. Images were stored on magnetic tape.

References: [A1]

A2.1.2 The Amdahl 470

Amdahl Corporation (1975–1980).

The Amdahl 470 series were pipelined von Neumann machines, with pre-fetched instructions and operands, and a data cache. The machine is divided into: an I-unit, which fetches and decodes instructions; an E-unit which performs data transactions; an S-unit, which manages store accesses; and a C-unit, which performs input/output operations. Several versions of the machine were produced, starting with the 470V/6 in 1975, each of which used the most up-to-date technology available.

References: [A2, A3]

A2.1.3 The IBM 3033

International Business Machines (1978).

The IBM 3033 was a conventional von Neumann machine, with pre-fetched instructions and operands, and a data cache. The machine was similar in structure to the Amdahl 470V/8, but was microcoded for flexibility, rather than hard-wired as in the 470/V8.

References: [A3]

A2.2 Heterogeneous Sequential Machines with Multiple DPUs

A2.2.1 The ILLIAC III

University of Illinois (1963).

The ILLIAC III was a heterogeneous sequential machine designed for pattern recognition tasks; in particular, the analysis of bubble-chamber photographs. The machine was composed of a number of sections: a lattice-based SIMD section, a special-purpose list-manipulating unit, and a general-purpose front-end unit. The lattice section of the machine contained 1024 processing units in an 8-connected square array. These PUs were simple boolean units, with some thresholding logic.

References: [A4]

A2.2.2 The IBM System/360 Model 91

International Business Machines (1968).

This was an early pipelined machine, developed from STRETCH [A7]. Separate floating-point and fixed-point execution units (DPUs) could operate in parallel and, within these, functional units could accept new operands before completion of the previous calculation. A total of eleven functional units were provided. A significant innovation was the common data bus between the functional units, which allowed results to pass from one DPU to another without having to pass through a register.

References: [A5, A2, A6]

A2.2.3 The CDC STAR-100

Control Data Corporation (1971).

The STAR-100 was designed to allow operand pre-fetching in a similar manner to instruction pre-fetching. This can be particularly effective on vector instructions. A high memory bandwidth was provided by a 32-way interleaved store, with 512-bit super-words, and a mechanism was provided which enabled selected bits of these to be modified. Two pipelined 64-bit arithmetic units (DPUs) were provided, each of which was capable of splitting into two independent 32-bit units. The two pipes were not identical, but differed in their functional units. The two pipes could also function as one 128-bit pipe for certain instructions. A stream unit handled operand and result transfers to and from the main store. A technique known as 'shortstopping' allowed a result from a functional unit to be fed back directly to become an operand in another instruction. The STAR-100 used a paging system similar to that of Atlas [A14], and special look-ahead action had to be taken to deal with result page faults. A feature known as a control vector was provided to allow processing of non-zero elements of sparse vectors. The STAR-100 architecture was designed for efficient execution of the language APL [A15], but suffered from engineering difficulties, and attracted much criticism. The long start-up time for vectors and relatively poor scalar performance (one tenth of the vector speed) was particularly criticised.

References: [A9, A2, A11, A6, A12, A13]

A2.2.4 The Digital Image Processor 1 (DIP-1)

Department of Electrical Engineering and Applied Physics, Delft University (1976–1979).

The DIP-1 processor is a reconfigurable multifunction pipelined processor, with nine 12-bit integer/18-bit floating point functional units which may be dynamically configured in any order, selected by microcode. Operand fetching is also controlled by microcode, and may be overlapped with processing. These functional units include two ALUs, a multiplier and a look-up table module. Two line buffers are provided to store previous lines and, thus, allow easy neighbourhood accesses. The DIP-1 acts as a DPU attached to a Hewlett-Packard HP1000 host, which provides a FORTRAN programming system.

References: [A17, A19, A20]

A2.2.5 The Magiscan 1 (M1)

The Wolfson Image Analysis Unit, University of Manchester (1977).

The Magiscan 1 was a heterogeneous sequential machine in which a specialised image processing unit was attached to a Data General Nova minicomputer. The attached DPU was a 12-bit microprogrammed unit with an instruction cycle of 200ns. The execution of each instruction was overlapped with the instruction fetch of the next. This DPU operated on image data stored in one of two DMUs. One of these DMUs was a $512 \times 512 \times 1$-bit frame store, which was displayed to the user, overlayed on the input video image. The other DMU was a high-speed (67ns cycle-time) 4096×6-bit image-window memory, into which image sections were loaded by the host processor. The host processor and the attached processor did not operate simultaneously, and so the machine is placed in the heterogeneous sequential class.

References: [A21]

A2.2.6 The TOShiba Pattern Information Cognitive System (TOSPICS)

Toshiba Corporation, Kawasaki, Japan (1978).

TOSPICS is a system which comprises a host machine (a TOSBAC-40C), four $512 \times 512 \times 8$-bit frame stores, four $512 \times 512 \times 1$-bit frame stores, and a unit known as the parallel pattern processor (PPP). Within the PPP are a number of functional units (DPUs). These include a two-dimensional convolution unit, a logical filtering unit, a linear co-ordinate transformation unit, a region labeller, and a histogramming unit. The two-dimensional convolution unit operates on masks which may be up to 8×8 pixels. This unit contains eight multipliers and a tree of eight adders which allow an 8×8 convolution operation to be performed in eight operations rather than 64. The logical filtering unit performs binary morphological transforms on 3×3 neighbourhoods, and the co-ordinate transformation unit performs affine transformations on pixel co-ordinates. This unit also interpolates pixel values, to allow fast translation, rotation and scaling of images. Only one of the PPP units may operate on an image at any time.

References: [A22, A23]

A2.2.7 The AT4/Leitz TAS

Centre de Morphologie Mathematique, Fontainebleau (1979).

The AT4, also known as the Leitz TAS, operated on hexagonally-tesselated binary images. The video input was thresholded, and the resulting binary images were stored in eight 256 × 256 image registers. A number of processing units were available to operate on these registers, and these units could operate in parallel. These included two logical transform units similar to the GLOPR, and a feature analyser unit. The system was hosted by a DEC LSI-11/2.

References: [A24, A25]

A2.2.8 Picture Array Processor II (PICAP II)

Department of Electrical Engineering, Linköping University, Sweden (1979).

In PICAP II, seven specialised DPUs are linked by a single 40 MHz time-shared bus to sixteen 256 K-byte memory units. All are controlled by a single 32-bit host minicomputer. The processing units may operate in parallel, and these include a template matching unit (the PICAP I processor), a 64 × 64 convolution processor, a general measurement unit and a cellular logic processor similar to the DIP-1. The host minicomputer is used to maintain an image database.

References: [A27, A29, A30, A25]

A2.3 Vector-Serial and Array-Serial Machines

A2.3.1 The Golay Logic Processor (GLOPR)

Perkin-Elmer Corporation (1967–1968).

The GLOPR was an image processing unit, attached to a Varian 620i. It operated sequentially on hexagonally-tesselated 1-bit images, in a similar manner to Cellscan. Three lines at a time were stored in variable-length shift-registers, and the first three elements used to provide a 3 × 3 window. From this, a hexagonal section was extracted and compared with the fourteen Golay primitives [A31], in all possible orientations. The 14-bit output was gated with a control word to produce a 1-bit output image. The system could be configured to operate on 32 × 32, 64 × 64 or 128 × 128 images.

References: [A1, A25]

A2.3.2 The Binary Image Processor (BIP)

Information International Incorporated (1968–1971).

The BIP was an array-serial machine which was specialised towards the processing of binary images. The array processing unit was capable of generating generalised functions of 3 × 3 neighbourhoods of two source images. One of these was used as a mask, to determine which areas of the other were to be processed. A selection register was used to indicate which of the eight neighbouring pixel values were of significance, and a bit-count of these could be incorporated in the result calculation. The Euler number of the neighbourhood (the number of groups of pixels surrounding the centre pixel) was calculated by the hardware, and this could also be used in the calculation of a result image.

References: [A32, A1, A25]

A2.3.3 Picture Array Processor I (PICAP I)

Department of Electrical Engineering, Linköping University, Sweden (1973).

The PICAP I processor operated on 4-bit images in a serial manner, using shift registers to provide access to a 3×3 neighbourhood. A template store was also accessible to the ALU. On every pass of the image, a histogram, maximum and minimum pixel values, and the sum of pixels were made available in registers. A host processor controlled the PICAP processor, which was connected to four video cameras, two monitors, a 512×512 frame store and other peripherals, through a complex video interface module.

References: [A34, A35, A37, A25]

A2.3.4 The High-Speed Pattern Processor (HSPP)

Osaka University, Japan (1975).

The HSPP is a machine specialised for image processing using local operators. Neighbourhoods are loaded automatically from a double-buffered 64×64 image, to a double-buffered 5×5 window memory. This is addressable by the main ALU, under microprogram control. Microcode may be dynamically loaded from the host processor, which also organises the loading of the image memories.

References: [A23]

A2.3.5 The Image Analyser (IA)

Centre de Morphologie Mathematique, Fontainebleau (1975).

IA is an image analysis machine which operates on $256 \times 256 \times 1$-bit images, with hexagonal tesselation. Four $256 \times 256 \times 1$-bit memory units have serial outputs connected to a combinatory logic unit. The output of this is sent to the transformation processor, which allows access to immediate neighbours and performs hexagonal template matching operations. Result images pass to a measurement unit, which is connected to a host machine. The IA is programmed using MORPHAL, a FORTRAN-like language.

References: [A38]

A2.3.6 The Floating Point Systems AP-120B

Floating Point Systems, Beaverton, Oregon (1976).

The AP-120B was designed by Floating Point Systems as a floating-point vector processing unit, to be connected to minicomputers or mainframe machines manufactured by other companies. The AP-120B was a microprogrammed DPU, which contained two 38-bit pipelined floating-point arithmetic units (an adder and a multiplier), a number of scratchpad memory units, and a table look-up unit. The floating-point units were capable of simultaneous operation, and each was pipelined to increase performance. Vector operations were carried out by the use of microcode routines.

References: [A7]

A2.3.7 The Cray-1

Cray Research Incorporated (1977).

The Cray-1 performs vector instructions on a set of eight 64-word vector registers. These are streamed through one of a set of six functional units, back to a destination register. A total of 12 functional units are provided, but not all may operate on vectors. The purpose of these registers is to increase relative performance of vector instructions on short vectors. A technique known as chaining is used if the destination of one instruction is an operand of a subsequent operation. In such case, it is not necessary to completely fill the intermediate register before the second operation commences. A separate scalar unit is provided, but scalar registers may be used in some vector instructions. Four 64-parcel (16-bit) caches are used to buffer instructions.

References: [A2, A7, A6]

A2.3.8 The CDC Cyber 205

Control Data Corporation (1979–1981).

In the Cyber 205, instructions are decoded by an IPU and are executed by either a scalar processor or are passed to a vector processing unit, which comprises one, two or four reconfigurable vector pipelines. The scalar unit is based on that of the CDC 7600. Shortstopping is provided, in a similar manner to the CDC STAR-100 and, again, a virtual addressing scheme based on Atlas is used, quite like the MU5 store access control unit [A40]. The vector processor resembles that of the CDC STAR-100 and comprises up to four reconfigurable multifunction 64-bit floating-point pipelines which operate on alternate data elements. Each pipe may be split to operate on two 32-bit words. A vector control unit and input and output stream units control vector operations. Control vectors are used for handling sparse vectors, processing only non-zero elements.

References: [A39, A7, A6, A12]

A2.3.9 The CDC NASF

Control Data Corporation (1979).

The CDC NASF was designed in response to the NASA requirement for a large simulation facility, and is based on the design of the CDC Cyber 200 series. The NASF is equipped with a scalar processor and five reconfigurable 64-bit floating-point vector pipelines. At any time, only four of these are used, and one remains as an on-line spare. Many simple arithmetic units in the pipelines are duplicated and, during normal processing, these perform self-checking. If a fault is detected, the fifth pipeline unit is switched into use automatically, and the failed unit may be repaired with no break in processing. The NASF memory is divided into three levels: 8 M-words of ECL store, 16 M-words of MOS memory, and 64 Mwords of sequentially-accessed CCD memory. A complex interconnection network is provided between the memory unit and the DPUs, which allows easy access to vectors which are stored in a non-contiguous manner.

References: [A7]

A2.3.10 The Magiscan 2 (M2)

The Wolfson Image Analysis Unit, University of Manchester / Joyce-Loebl Ltd., Newcastle-Upon-Tyne (1980).

The Magiscan 2 is a specialised image analysis machine, with two data processing units; a general purpose processor and an image address processor. The main processor has a 16-bit word length and a cycle time of 150ns, and the image address processor is a specialised unit used for manipulating image memory addresses. Both the main processor and the memory address processor are microcoded bit-slice devices. Four K-bytes of microcode store is available, and this is partly taken up by the p-code interpreter, which supports the UCSD p-system operating system. The operating system and user programs are compiled into p-code, which is a pseudo code interpreted at run-time by the p-code interpreter. All p-code programs and data reside in the 64 K-words of 16-bit macro memory. The UCSD p-system provides editing, compiling and floppy disc file handling facilities. Microcode may be dynamically loaded from the main processor to allow critical sections of programs to be efficiently encoded. Images, which are of 512×512 pixels with 6-bit grey level resolution may be viewed on a built-in monochrome monitor, or on a separate colour monitor, which allows pseudo-coloured images to be displayed. An image store of up to 2 M-bytes is provided, composed of 8 or 16 planes of $1024 \times 1024 \times 1$-bit semiconductor storage. These are normally used to store a mixture of $512 \times 512 \times 1$-bit and $512 \times 512 \times 6$-bit images. Peripherals include a motorised microscope, a 35mm film transport, dot-matrix printers, and a Ramtek 4500 C photographic hard-copy output device.

References: [A41, A43]

A2.4 Orthogonal Machines

A2.4.1 The OMEN Series

Sanders Associates, Nashua, New Jersey (1970).

The OMEN series of computers were orthogonally organised machines, based on the DEC PDP-11. The PDP-11 accessed the OMEN memory in a conventional manner, using 16-bit words. An additional 64-bit vertical arithmetic unit accessed the memory as 64-bit bit slices. This unit was capable of performing operations such as bit reversal, and perfect shuffle, in addition to more conventional functions.

References: [A45, A46]

A2.4.2 The STARAN Series

Goodyear Aerospace (1971–1972).

The STARAN series of machines were designed to provide a content-addressable memory system for a host machine. In the STARAN machines, each word of store had a bit-serial DPU, intended to make serial comparisons with a broadcast word and, thus, provide an associative store. The processing elements were, however, able to perform other operations and so STARAN could be used as an orthogonal machine. The STARAN store was made up of between one and 32 'arrays', each of which was a 256×256 memory unit. A set of 256 data processing units was associated with each array, and these could read data in a 256-bit word mode, or a 256-bit bit-slice mode. No direct inter-processor communication was provided, but a 'flip unit' was placed between the processors and the memory unit, and this allowed the data from the memory units to be permuted on either read or write operations. All processors were controlled by a single instruction processing unit, which was linked to the host processor.

References: [A47, A48, A49, A51, A53, A54, A55]

A2.4.3 The Distributed Array Processor (ICL DAP)

International Computers Limited, Manchester (1976).

The ICL DAP is an orthogonal machine, in which a portion of the main store of an ICL 2980 is mapped onto the address space of a 64×64 array of 1-bit processing units. Each of these comprises a full adder, with three 1-bit registers, and is directly connected (multiplexed) to its four nearest neighbours, and also to X and Y buses. The DAP is constructed using VLSI components, with sixteen PUs per chip. No store is provided on-chip, but each PU has 4096 bits of external memory. The DAP is programmed in DAP-FORTRAN.

References: [A57, A59, A61, A62, A63, A65, A66, A67, A1, A69]

A2.5 Homogeneous Sequential Machines with Multiple DPUs

A2.5.1 Simultaneous Operation Linked Ordinal MOdular Network (SOLOMON)

Westinghouse Electric Corporation, Baltimore, Maryland (1960).

SOLOMON was a proposed machine design based on a 4-connected, 32×32 array of DPUs, each attached to a DMU containing 128 words of 32-bit memory. The ALU in each processing unit was bit-serial, but multi-bit registers were provided to store partial results. In addition to the hardware provided to inhibit the actions of individual processing units (to allow conditional code to be executed), SOLOMON had row-enable and column-enable options. The machine was never built, but its design was very influential, especially on ILLIAC IV.

References: [A70, A71, A7, A72]

A2.5.2 The ILLIAC IV

University of Illinois / Burroughs Corporation (1967–1975).

ILLIAC IV was based on the design of SOLOMON, and the original design comprised 256 data processing units in four 8×8 quadrants, with each quadrant under the control of a single IPU. Each DPU was a full 64-bit floating-point processor, with four 64-bit registers, a 16-bit index register, and was directly connected to a DMU containing 2048 words of private memory. Each DPU could be configured as as a 64-bit processor, two 32-bit processors, or eight 8-bit processors. Only a single quadrant was ever built, and this suffered from severe engineering difficulties, due to its size and its use of 'state of the art' technology. A Burroughs B6700 acted as a host to ILLIAC IV.

References: [A73, A74, A75, A76, A77, A72]

A2.5.3 The Parallel Element Processing Ensemble (PEPE)

Honeywell / Burroughs Corporation (1970).

PEPE was a multiprocessor specialised for radar tracking applications. A number of data processing units were controlled by a single instruction processing unit, which was in turn under the control of a host processor. Each data processing unit comprised a specialised correlation unit, an integer/floating point arithmetic unit, and was connected to a 1024 × 8-bit DMU. No inter-processor connections were provided. Several versions of PEPE were designed, of sizes ranging from 16 to 288 processing elements, and using different host machines (IBM 360/65 or CDC 7600). PEPE was programmed using a special version of FORTRAN, known as P-FOR.

References: [A78, A79, A80, A49, A81, A82, A83, A77]

A2.5.4 The Diff3

Coulter Biomedical Research Incorporated / Perkin-Elmer Corporation (1972–1977).

The Diff3 system was specialised towards microscope slide handling, and was equipped with a motorised microscope. The Diff3 comprised a Data General Nova-4, operating as a control unit, and the Golay processor as an image processing unit. Hexagonally tesselated 6-bit images were used, and these could be histogrammed by a special unit, or magnitude thresholded for processing by the Golay unit. The Golay unit implemented the Golay image transforms [A31]. Four 64 × 64 × 1-bit image memories acted as inputs to the eight transform generation circuits, which operated in parallel. These were similar to the single transform units in GLOPR, comprising a circular shift register and look-up ROMS to detect the Golay primitives.

References: [A85, A1]

A2.5.5 CLIP3

Department of Physics and Astronomy, University College, London (1973).

The CLIP3 system was a 12 × 16 lattice-connected processor array, constructed using TTL MSI chips. Each processing unit comprised an eight-function boolean processor, with two 1-bit registers, and had a 16-bit local store. Each processor was directly gated to its eight neighbours, and a threshold unit allowed a neighbour count to be made, and instructions performed conditionally on this. A hexagonal tesselation mode was also provided. Each CLIP3 board contained four processing elements, and a separate board contained their memory. One register from each processor formed part of a 192-bit shift register, which was used for input and output of images. Conditional jumps were permitted on the state of panel switches, and on the output of a gate which ANDed all the previous D-register outputs. A scanning system allowed the array to interface to a video camera, and perform operations on a 96 × 96 image. CLIP3 was hosted by a DEC PDP 11/10.

References: [A86, A87, A89]

A2.5.6 The Burroughs Scientific Processor (BSP)

University of Illinois / Burroughs Corporation (1974–1980).

The Burroughs BSP is a homogeneous sequential processor with sixteen data processing units and seventeen data memory units. The processing units are connected to the memory units by two full crossbar networks, one of which operates in each direction. The crossbar which is connected to the outputs of the processors may also be used for inter-processor connections. Each DPU is a microcoded 48-bit processor with floating-point capability. The distributed memory stores up to a total of 8 M-words with a cycle time of 160ns but, as sixteen words are fetched in a single cycle, a rate of one operand per 10ns may be reached in long vector operations. A single control processor (IPU) controls the actions of the DPUs, and also those of a separate scalar processor. In addition to the distributed memory, a central file memory unit is also provided, which has a maximum size of 64 M-words (48-bit words). This memory is used as a fast substitute for disc memory. The BSP is hosted by a conventional Burroughs B7800 series machine, and is programmed using a parallel FORTRAN system.

References: [A77, A7, A90]

A2.5.7 CLIP4

Department of Physics and Astronomy, University College, London (1976–1980).

The CLIP4 system is a 96×96 lattice-connected processor array, with two 6-bit frame stores and level selectors. CLIP4 was the first bit-serial processor array to be based on VLSI technology (40-pin DIL chips, each containing eight processing units). Each data processing unit comprises two identical boolean processors, one of which is equipped with a carry bit, plus two other 1-bit registers. All eight nearest neighbours are directly connected (AND-OR gated) to each processor, and a hexagonal tesselation mode is also provided. A 32-bit DMU is provided (on-chip) for each DPU, but no external memory may be added. Input and output is performed by treating all the 'A' registers of every processing unit as a large shift register. Typical operation time is 10μs. CLIP4 is hosted by a DEC PDP-11.

References: [A86, A87, A91, A93, A94, A95, A1, A65, A66]

A2.5.8 The BASE Systems

School of Electrical Engineering, Purdue University, Lafayette, Indiana (1979).

The BASE-4 and BASE-8 are two similar machines based on 8-connected 4×4 and 8×8 lattice networks. Each data processing unit of the BASE systems comprises a 3-input boolean ALU and is connected to a bit-organised DMU. The eight nearest neighbours are connected to the ALU by a masked AND-OR gating arrangement, which implements a number of commonly-encountered neighbourhood functions.

References: [A96, A97, A98]

A2.5.9 The Massively Parallel Processor (MPP)

Goodyear Aerospace Corporation / NASA (1979–1982).

The MPP system is a 128×128 lattice-connected array of processors, with four-neighbour (multiplexed) connectivity. Four redundant columns are provided as on-line spares, to allow fault tolerance. Each data processing unit comprises a one-bit full adder and logical unit, six one-bit registers, and a variable-length (maximum 30-bit) shift register. A 32-bit DMU is provided on-chip, and provision is made for additional external memory. Each VLSI chip contains eight processing units and their memory. Input and output to the array takes place through switching networks connected to two opposite edges of the array. Data is shifted across the array to communicate to the inner processors. A DEC VAX-11/780 acts as a host to the MPP system, but a fast scalar processor is provided within the MPP, for non-array operations.

References: [A99, A101, A102, A7, A103, A104, A65, A66, A105]

A2.5.10 The Preston-Herron Processor (PHP)

Carnegie-Mellon University / University of Pittsburgh / Perkin-Elmer Corporation (1979).

The PHP is a specialised image processing unit designed to implement local operators on a 3×3 neighbourhood. A 3×18 pixel image window is made available from three RAMs, each containing the same image segment, with each RAM providing one line of the window. This data is used by 16 ALUs, operating in parallel, to produce 16 results. A processing rate of 200ms is quoted for 512×512 images.

References: [A106, A1, A25]

A2.5.11 The Microprogrammable Vector Processor (MVP)

School of Electrical Engineering, Purdue University, Lafayette, Indiana (1980).

In the MVP, a single instruction stream controls four TTL/LSI floating point arithmetic units and a microprogrammed vector control unit (AMD 2901A). A single 96 K-word memory unit is used for image storage, and this is accessed through a vector control (streaming) unit. Each DPU has 256 data registers, accessible to the other DPUs via a 4×4 crossbar network. Transcendental functions are provided as ROM look-up tables. The MVP system is hosted by a DEC PDP-11/45.

References: [A107]

A2.5.12 CLIP5

Department of Physics and Astronomy, University College, London (1981).

CLIP5 is an updated version of CLIP4, with very similar architecture. Each data processing unit has a dual boolean processor, a full adder, and gated connections to eight neighbours. A hexagonal tesselation mode is provided. In CLIP5, each chip contains sixteen processors, but all memory is located on a separate chip, to allow the use of large commercially available memory devices. Three extra registers have been provided, one for memory buffering, one for neighbour connection buffering, and one for input/output. The neighbour gating system is modified to provide a thresholding system. CLIP5 is approximately ten times as fast as CLIP4, with an operation time of $1\mu s$.

References: [A108, A65]

A2.5.13 CLIP6

Department of Physics and Astronomy, University College, London (1981).

The design of CLIP6 is based on that of CLIP4, but operates with bit-parallel arithmetic, instead of bit-serial. Grey-level values are 6-bit quantities, but the single ALU, and some internal registers operate on 12-bit data. Inter-processor communication takes place through a multiplexer in CLIP6, rather than the gating arrangement of CLIP4. A condition code register is provided to allow the use of complex conditional operations. The DPUs are to be fabricated from TTL, and commercial MOS memory is used.

References: [A108]

A2.5.14 The Hitachi IP

Hitachi Corporation, Japan (1981).

The Hitachi IP comprises sixteen data processing units, in a 4×4 lattice configuration. Each DPU has five registers which may be loaded automatically from one of three 256×256 image stores. The system is hosted by a conventional von Neumann machine.

References: [A23]

A2.5.15 The Pipeline-Array Processor

Department of Computer Science, University of Washington (1981).

The pipeline-array processor system is a proposed homogeneous sequential machine based on lattice-connected VLSI processors. This VLSI chip contains eight data processing units, and associated with each of these are eight 'register sets' (64 register sets in total), which allow each processor to deal with data for eight pixels, The processing units comprise a 1-bit ALU and a gated connection to eight neighbours. Each register set contains four 1-bit registers, and an 8-bit store. Provision for external memory units is also made. The register sets are implemented as shift registers, so that the processors may switch rapidly from one pixel to another. The chip used in this design is similar, but not identical, to that used in PCLIP.

References: [A109, A65]

A2.5.16 Reeves' VLSI machine

School of Electrical Engineering, Purdue University, Lafayette, Indiana / Cornell University (1981).

This is a proposed lattice-based machine in which each data processing unit comprises one 32-bit parallel adder, a separate serial ALU, and two 32-bit, variable-length shift registers. Neighbours are connected bit-serially, via special interconnection chips which also contain a neighbourhood gating logic (masked AND-OR) and an input/output mechanism. A 1×16K-bit indexed store is provided on-chip, and provision is made for external memory units to be attached. Each chip is expected to contain 8-32 processors.

References: [A97]

A2.5.17 The Adaptive Array Processor (AAP)

Nippon Telegraph and Telephone Company, Japan (1982).

The AAP is a proposed machine to be constructed from LSI chips. Each chip contains a 8 × 8 lattice array of data processing units, each with a 96-bit on-chip DMU. No provision is made for the addition of external memory. The on-chip store is organised as two shift registers of 32 and 64 bits, which are fed serially to the sixteen-function ALU. Eight neighbours are connected via a multiplexer, and a separate multiplexer provides up and down connections, to permit three-dimensional arrays to be constructed.

References: [A65, A23]

A2.5.18 GRID

General Electric Company (UK) (1982).

Each data processing unit of the GRID system comprises a four-input, two-output ALU, with a number of single-bit registers, and has a 64-bit, on-chip DMU. An external memory port is also provided, and four neighbours are directly connected to each processor. The GRID system topology is reconfigurable, with a choice of two-dimensional lattice connections, X and Y bus connections, single-processor addressing or a chained mode, in which an entire binary image may be shifted into a counter. Each GRID chip contains 32 processors, with a 100ns cycle time.

References: [A65]

A2.5.19 LIPP

Department of Electrical Engineering, Linköping University, Sweden (1982).

Each data processing unit of LIPP comprises a 1-bit ALU, two variable length shift registers, a shiftable up/down counter and an index register which may be used to access external DMUs. Inter-processor communication takes place by means of shared multi-port DMUs, which are fabricated on custom VLSI chips. Each processor is connected to four DMUs, through a rectangular lattice network. Buses are provided to give rapid access to processors along rows or columns.

References: [A110, A65]

A2.5.20 CLIP7

Department of Physics and Astronomy, University College, London (1983).

The CLIP7 system is a 512 × 4 lattice-connected array, and each data processing unit deals with 128 pixels of image data. The DPU is similar to that of CLIP6, dealing with images of up to 8-bits in a bit-parallel manner. A single 16-bit ALU is used, with four addressable registers. Eight neighbours are serially connected to each processor, and are multiplexed internally. Each CLIP7 chip contains one DPU, and 256 bits of off-chip storage are provided for each pixel. A number of small Winchester disc units are distributed through the machine to provide an image database. A UNIX† machine is used as a host to the system. The CLIP7 processing unit is estimated to be 130 times as powerful as CLIP4, so the CLIP7 system will process a 512 × 512 image in the same time that CLIP4 processed a 96 × 96 image.

References: [A111, A112]

† UNIX is a registered trademark of AT&T Bell Laboratories in the USA and other countries.

A2.5.21 The Diff4

Coulter Biomedical Research Incorporated / Perkin-Elmer Corporation (1983).

The Diff4 is a development of the Diff3 automatic microscope slide analysis system. The main differences are the addition of ten Intel 8085/8086-based units for peripheral control, and that the GLOPR processing units are controlled by an Intel 8086 single-board computer system.

References: [A113]

A2.5.22 The Quadruple Alu-1 (QA-1)

Kyoto University, Japan (1983).

The QA-1 is a microprogrammed processor with four separate ALUs controlled by a single microinstruction stream, which contains separate instruction fields for each ALU. The four 16-bit ALUs are connected by two full crossbars, at their inputs and outputs, to 15 registers and two stacks. The other input of the ALU is connected to set of four memory units. The microinstructions are 160 bits with a cycle time of 350ns.

References: [A23]

A2.5.23 The Reconfigurable Processor Array (RPA)

University of Southampton (1984).

This proposed machine consists of a two-dimensional, 4-connected, lattice of single bit ALU cells, whose outputs may be cascaded along one axis to allow multi-bit operation. Each cell comprises a full adder, and four one-bit registers, (designated C, R, A and Y). The array is designed to be fault tolerant, and implements three actual VLSI cells for each cell required. Each NMOS chip will contain 16-32 DPUs, and a wafer-scale implementation is planned.

References: [A114]

A2.5.24 Picture Array Processor 3 (PICAP 3)

Department of Electrical Engineering, Linköping University, Sweden (1986).

PICAP 3 is designed to deal with three-dimensional image data, such as those obtained from some medical scanners. The machine is composed of a linear array of data processing units, under the control of a single instruction processing unit. This is, in turn, controlled by a host minicomputer. Each processing unit is a 16-bit AMD 29016 bipolar microprocessor, with 32 registers, and has a private 2 M-byte DMU. The processors are interlinked by a packet-switched ring network. A four-processor prototype is under construction, and a 64-processor version is planned.

References: [A116]

A2.6 Systolic Machines

A2.6.1 Kung's Systolic Machines

Carnegie-Mellon University, Pittsburgh (1978).

Kung and his associates at Carnegie-Mellon have developed a number of unnamed systolic machines based on fixed-program processing elements. These are fixed-function machines, in which each processing element repeatedly performs the same part of a calculation on different data items, which are then passed to neighbouring processors. The fixed program and interconnection pattern depend on the calculation to be performed. Systolic structures have been described to implement one-dimensional and two-dimensional convolution, matrix multiplication, least-squares fit, triangular linear systems solution, discrete Fourier transform and a number of other algorithms.

References: [A117, A118, A119, A120]

A2.6.2 The Cytocomputer

Environmental Research Institute of Michigan/University of Michigan (1980).

The Cytocomputer consists of 113 modified Cellscan machines in a linear array, with a controlling host machine. The processing units consist of 88 silhouette units and 25 umbra units. Silhouette units take a 3×3, 8-bit input and perform a preset magnitude thresholding operation, followed by a table look-up, using one bit from each input pixel as an index. Umbra units provide different threshold operations for each neighbour, followed by table look-up. This machine structure allows 113 separate operations to take place on a 60-pixel wide 8-bit image slice, in real time. A processing rate of 1.6 Megapixels/sec is quoted for 1024×1024 images.

References: [A122, A123, A124, A125, A126, A1, A128]

A2.6.3 The ESL Systolic Processor

ESL Incorporated, San Jose, California (1981).

The ESL systolic processor is a linear array of multiply-and-add systolic elements, very similar to those described by Kung [A120]. The array is hosted by a VAX 11/780.

References: [A129, A130]

A2.6.4 The Flexible Image Processor System (FLIP)

Research Institute for Information Processing and Pattern Recognition, Karlsruhe, W. Germany (1981).

FLIP is an array-serial, byte-parallel reconfigurable systolic pipeline, with sixteen identical data processing units. Each of these contains an ALU and three register files for temporary storage of partial results. A flexible inter-processor network allows the results from any processor to pass to any other, and complex arithmetic operations may be implemented in this way. A device known as the PEP controls data reads from, and writes to, the image store. FLIP is programmed at an assembly code level.

References: [A131, A132]

A2.6.5 The Configurable Highly Parallel Computer (CHiP)

School of Electrical Engineering, Purdue University, Lafayette, Indiana (1982).

CHiP is a reconfigurable systolic machine, to be implemented in VLSI. The VLSI chip contains a rectangular array of processors and switching elements, in a ratio of one to three. By suitably setting the switching elements, CHiP may be configured as a rectangular array, hexagonal array, torus, binary tree or double tree.

References: [A133, A134]

A2.6.6 The Bagel

The Weizmann Institute of Science, Rehovot, Israel (1983).

This is a systolic machine with a skewed torus interconnection. The skew allows the machine to function as a linear array or as a rectangular array. The processing elements used are Inmos transputers [A136, A138].

References: [A135]

A2.7 Unshared-Memory Multiprocessors

A2.7.1 The Processor for Information Storage and Retrieval (PISR)

University of Florida (1970).

This is a proposed machine, designed specifically to perform fast sub-string searches in stored text. PISR comprises a large number of 'cells' connected in a binary tree structure. Each cell consists of a comparator (DPU), some memory (DMU), and network connection logic. The tree-structured network is used for instruction and data broadcasting, and each cell receives instructions from its parent, and propagates this to both offspring. The tree may be partitioned, by isolating sub-trees, to form 'instruction domains' which operate individually as SIMD machines. Each cell may be used for data or program storage, and instruction cells broadcast instructions to all data cells within their instruction domain.

References: [A139]

A2.7.2 Elaboratore Multi Mini Associativo (EMMA)

ELSAG, Genova (1975).

EMMA was a multiple processor machine, with a hierarchical bus interconnecting the data processing units. Processors had their own private local data memory units. Each cluster was organised by a data exchange co-ordinator, which carried out DMA transfers across the buses when this was requested by a processor. Software was based on Petri nets [A141].

References: [A140]

A2.7.3 The CERVISCAN

MRC Clinical and Population Cytogenetics Unit, Western General Hospital, Edinburgh (1976).

The CERVISCAN system comprised a DEC PDP-9 connected to a Modular 1 machine. The PDP-9 was equipped with a motorised microscope and other peripherals. The system was primarily used for processing cervical smear tests, hence the name CERVISCAN.

References: [A142]

A2.7.4 The SUNY Hierarchical Machine

Department of Computer Science, State University of New York, Stony Brook (1977).

This machine has a hierarchical structure, in which each PU has control of a time-shared bus to which are attached DMUs and interfaces to subordinate processors. The processing units are AMD 2900 series bit-slice processors, and buses are controlled using commercially-available programmable interface chips. Each PU has a 'side door' to a shared mass store device for mass data communications. The prototype design uses ten PUs.

References: [A144, A146]

A2.7.5 ARES

Kyoto University / Kokusai Denshin Denwa Company, Japan (1978).

ARES is an unshared-memory multiprocessor composed of eight units which are themselves homogeneous sequential machines with eight DPUs each. Within these 'clusters', a single IPU controls the actions of all eight DPUs. These DPUs are constructed from SN74S481 bit-slice components, and each has a private memory unit. Associated with each cluster is a unit known as a multiple response resolver, which may be used to co-ordinate the actions of the cluster.

References: [A147]

A2.7.6 The Atmospheric and Oceanographic Information Processing System (AOIPS)

Goddard Space Centre, Maryland (1978).

The AOIPS comprises a PDP-11/70 and a PDP-11/45 together with mass store, including an optical disc, and image I/O devices. The main load of the system is the processing of LANDSAT images. The AOIPS system is the planned host for the MPP array.

References: [A148]

A2.7.7 MU6-G

Department of Computer Science, University of Manchester (1978–1982).

MU6-G was designed to be the general-purpose model of a range of machines varying in size from personal machines to 'supercomputers'. MU6-G was a pipelined machine, with instruction assembly, operand assembly, and instruction execution overlapped for consecutive instructions. Two separate arithmetic units operated on 32-bit integers, and 64-bit real numbers, but these did not operate in parallel. Virtual store was implemented by a system which used a combination of fast current page registers and slower page address registers. An RML 380Z microcomputer was used as a 'hardware front panel' to run diagnostic software, and a DEC PDP-11/20 acted as a front-end co-processor. Mass store devices and peripherals were attached to both machines. Since either the PDP-11/20 or MU6-G could operate independently, the system is classified as an unshared-memory multiprocessor.

References: [A149]

A2.7.8 The UCLA CHI

University of California, Los Angeles (1978).

The UCLA CHI is a multi-computer system used for large simulation tasks, and comprises a custom mainframe processor, a Floating Point Systems AP-120B array processing unit, and four input/output processors (IOPs). The mainframe processor is a 16-bit fixed-point machine with 64 K-words of memory. The IOPs handle disc transfers to and from the host and the AP-120B.

References: [A150]

A2.7.9 The X-Tree

University of California, Berkeley (1978).

X-Tree is made up of a number of 'X-nodes', connected by a binary tree network, augmented by ring connections at each level. These additional paths reduce the inter-processor latency considerably. Each X-node comprises a data processing unit with a private data memory unit, and a four-way message passing switch unit, similar in many ways to the transputer [A136, A138]. No memory is shared, and so all communication takes place directly between processing units.

References: [A152, A153, A154]

A2.7.10 The Cal-Tech Tree Machine

California Institute of Technology (1980).

This machine is based on the binary tree network, with a DPU, a DMU, and communications hardware at each node. The reduced instruction set processors operate on 12-bit words, and memory is organised in 4-bit sub-words. Instructions are of variable length (2-6 sub-words). All inter-processor communication is via the tree network, and no shared memory is provided.

References: [A156, A154]

A2.7.11 The Recursive Machine

Xerox Corporation, Palo Alto, California (1980).

This is a quaternary tree structured machine, in which each node comprises a DPU, a DMU, control and four subsidiary connections, which may be to the processing elements of other recursive nodes. Programs are based on the Smalltalk language and are tree/list oriented.

References: [A154]

A2.7.12 ZMOB

University of Maryland (1980).

ZMOB is a machine composed of 256 processing units connected by a single packet-switched ring network, known as the 'conveyor belt', with an additional connection provided for the host VAX 11/780. Each processing unit comprises a Z80A microprocessor with hardware multiplier, 1 K-byte of EPROM, 63 K-bytes of RAM, and a bus interface. The packet-switched ring is 48 bits wide, with a slot time of 100ns. The ring is used as an inter-processor message passing mechanism, where messages may be sent to a particular processor, or to a subset of processors. The MOBIX operating system allows a ZMOB processor to execute UNIX† processes which are sent to ZMOB by the host VAX-11/780, which runs UNIX.

References: [A157, A158, A159, A160, A161, A104]

A2.7.13 The General Operator Processor (GOP)

Department of Electrical Engineering, Linköping University, Sweden (1981).

The GOP comprises a conventional 16-bit 2900 bit-slice processor fed by four image-serial, 16-48 bit-parallel pipelines specialised for space-domain convolution calculations. These convolution pipes include a large mask store, which allows irregularly-shaped masks to be used. The main processor takes partial results from the convolution pipes, and sends results to the host. It cannot access the image slice directly, but feedback to the four convolvers may occur via a feedback memory. The GOP is designed to implement Granlund's 'General Operator' [A164]. A FORTRAN-based language, INTRAC, is used for high-level programming.

References: [A162, A163]

† UNIX is a registered trademark of AT&T Bell Laboratories in the USA and other countries.

A2.7.14 The MIT Connection Machine (CM-1)

Massachusetts Institute of Technology / Thinking Machines Corporation (1981).

Each of the 64 536 processing 'cells' of the connection machine is a finite state machine, which comprises a state vector, several registers and a local store containing 4096 4-bit words. No action is taken until the cell receives a message from some other cell. When this happens, the cell state and the message type are combined to determine the appropriate action, by table look-up. This action may involve a state change, register manipulation, or message sending. The 'action rules' form the machine's equivalent of a program, and these are the same for all cells. The cells communicate via a packet-switched network with a binary 12-tube topology (sixteen cells at each switching node). The machine is so named because it is designed to implement connectionist cognitive models [A169].

References: [A165, A166, A167]

A2.7.15 The Parallel Pyramidal Array/Net

University of Wisconsin (1981).

This proposed machine comprises 256 8-bit DPUs, and 16 IPUs, all constructed from AMD 2900 bit-slice chips. The 8-bit processors are arranged as a 16×16 array, with four-neighbour connections. This array is divided into 16 groups of 4×4 processors, and each group is independently controlled in an SIMD fashion by control processor. Each 8-bit processor has 16 K-bytes of memory, and this may be read by adjacent processors. The processors on the outer edge of the array may connect to the opposite edge, or to a common border value. The processors in a group may be reconfigured to form one 64-bit 'master processor', which then has access to the entire 256 K-bytes of memory in the group. These master processors are connected to the host, a DEC VAX machine.

References: [A170]

A2.7.16 The Parallel SIMD/MIMD System (PASM)

School of Electrical Engineering, Purdue University, Lafayette, Indiana (1981).

PASM is a machine intended for image analysis tasks, in which a set of control units, operating independently, each control a number of processing units in an SIMD mode. Each processing unit has a double-buffered memory unit, which is a copy of a section of a large global memory, organised by a host machine. All inter-processor communication is direct, via an interconnection network, for which two designs are under consideration, namely the indirect binary n-cube and the augmented data manipulator [A176]. PASM is considered to be an unshared memory multiprocessor, despite its global store, because the machine is not capable of interactively sharing access to the global store; copies of large sections must be made, and later replaced. In the prototype under construction, 16 Motorola MC68010 processors are to be used, and a 1024-processor machine based on custom VLSI processors is planned.

References: [A171, A172, A173, A174, A175]

A2.7.17 PCLIP

Department of Computer Science, University of Washington (1981).

The PCLIP system is an unshared-memory MIMD machine, intended for image processing applications. PCLIP is constructed from a number of lattice-connected homogeneous sequential machines, which are interconnected to form 'layers' of a pyramid structure. Each of the layers of the pyramid contains a single IPU and a number of DPUs and DMUs. The PCLIP chip contains a number of processing units, each of which comprises a 1-bit ALU and a gated connection to thirteen neighbours, including those on the adjacent layers of the pyramid. Each processing unit contains three 1-bit registers, and an unspecified amount of memory.

References: [A178]

A2.7.18 The Wavefront Array Processor (WAP)

University of Southern California, Los Angeles (1982).

The WAP is an unshared-memory multiprocessor machine, with a rectangular lattice interconnection, designed to be implemented in VLSI. Although the WAP is described by its designers as a systolic machine, the processors are not fixed-program devices, and the machine is not globally controlled. Instead, it relies on local synchronisation between processors when passing data across the array, and is therefore classified as an unshared-memory multiprocessor machine.

References: [A179]

A2.7.19 The Pyramidal Architecture for Parallel Image Analysis (PAPIA)

Pavia University, Italy (1985).

PAPIA is a multiprocessor machine with a pyramid structure, composed of several layers of processors. Each layer is a lattice-based homogeneous sequential machine with its own instruction processing unit, so that layers may operate independently of each other. Each processing unit is connected to four neighbours on its own level of the machine, four 'child' units on the layer below, and one parent unit on the layer above. All processing units are identical, and are composed of a 1-bit arithmetic and boolean unit, two 1-bit registers, and two variable-length shift registers (maximum length 32 bits). Each processor has an attached memory unit which provides 256 bits of storage.

References: [A181]

A2.7.20 Systeme Pyramidal Hierarchise pour le Traitement d'Images Numeriques (SPHINX)

Institut d'Electronique Fondamentale, Universite de Paris Sud, France (1986).

SPHINX is an unshared-memory multiprocessor machine with a pyramid structure. Each layer is a lattice-connected homogeneous sequential machine with its own instruction processing unit. In this machine, each processor is connected to four others on the same layer of the pyramid, two others on the layer below, and one on the layer above. To allow this form of interconnection, alternate layers of the pyramid have aspect ratios of 1:1 and 2:1. All of the processing units are identical, and are composed of a 1-bit ALU, a number of 1-bit registers, and additional logic to send output to the child units. A 128-bit on-chip memory unit is provided.

References: [A182]

A2.8 Shared-Memory Multiprocessors

A2.8.1 The CDC 6600

Control Data Corporation (Early 1960s).

The CDC 6600 was a machine which exploited parallelism at two levels. The central processor contained ten functional units and 24 registers. At an instruction level, instructions in a sequential program were executed simultaneously, provided that different functional units were specified, destinations were different, and that the result of the first was not an operand of the second. This was facilitated by the use of an instruction format incorporating three register-address fields. Co-ordination was performed by a unit known as the 'stunt box'. At a higher level of parallelism, a peripheral processing unit (PPU) was used to control peripheral units and to perform data transfers between peripherals and the main store. This PPU was constructed in such a way that ten independent programs could be executed concurrently, by rapid process-switching of the PPU processor. Ten sets of PPU registers were provided to do this, and these, together with ten 4096-word memory units, were used in a cyclic sequence. This allowed the relatively fast processor (100ns addition time) to be matched to the slower memory (1μs access time).

References: [A183, A11, A6]

A2.8.2 The Texas Instruments Advanced Scientific Computer (TI ASC)

Texas Instruments, Dallas, Texas (1966–1972).

The TI ASC contained up to four 64-bit parallel static (fixed order) multifunctional reconfigurable arithmetic pipelines (DPUs). These were controlled by one instruction stream, and were connected to an 8-bank interleaved store. A memory buffer unit (MBU) was associated with each arithmetic unit, and this provided the operand streams and stored the result streams in the execution of vector instructions. A 'vector parameter file' was passed to the MBU to control its actions, and contained start addresses, vector length and other information. A peripheral processing unit contained an independently programmed 32-bit processor which was used to handle the transfer of data between peripherals and main memory. This unit had multiple register sets, similar to the PPU of the CDC 6600, to allow concurrent execution of up to eight processes.

References: [A184, A185, A2, A7, A6]

A2.8.3 The CDC 7600

Control Data Corporation (Late 1960s).

The CDC 7600 was a development of the CDC 6600, based on the same principles, but approximately four times as fast. This was achieved by modifications to the instruction pre-fetching mechanism, pipelining some of the functional units, and increasing the clock speed.

References: [A6]

A2.8.4 MU5

Department of Computer Science, University of Manchester (1968–1971).

The MU5 system comprised a number of units connected by a network known as the MU5 exchange. These units were the MU5 processor, the MU5 local core store, an ICL1905E, the little mu5 processor, a PDP-11/10, a block transfer unit and a mass store unit. The MU5 exchange was a 113-bit wide, time-shared bus with a transfer time of 100ns, and allowed any unit to send data packets to any other unit. The MU5 processor used high-speed buffers within the CPU for both instructions and data. Two arithmetic units were provided, one operating on 64-bit floating point data and the other on 32-bit integers. A virtual memory addressing scheme was implemented using current page registers. MU5 was the prototype for the ICL 2900 series of machines.

References: [A40, A6]

A2.8.5 C.ai

Carnegie-Mellon University, Pittsburgh (1972).

C.ai was a proposed machine comprising sixteen memory units connected to sixteen assorted other units, via a circuit-switched crossbar network. These other units included data processors units, a mass store, and the host machine. A DEC unibus connected all the processing units. The C.ai design was developed in conjunction with that for C.mmp. There are no plans to implement C.ai.

References: [A186]

A2.8.6 C.mmp

Carnegie-Mellon University, Pittsburgh (1972).

C.mmp consisted of sixteen PDP-11s (DPUs) connected by a circuit-switched full crossbar network to sixteen 64K × 16-bit memory units. The processors were connected by a separate cross-bar to peripherals and mass store devices. Each store module was eight-way interleaved, to allow multiple simultaneous memory accesses. Software synchronisation and deadlock avoidance was implemented by the processor concerned excluding all other processors from the relevant memory unit. The HYDRA operating system treated the machine as a set of sixteen concurrent processes.

References: [A187]

A2.8.7 Arquitecturas Heterarquicas Reconfigurables (AHR)

National University of Mexico (1976).

The AHR is designed to execute LISP programs by parallel evaluation of function arguments. A set of up to 64 Z80-based processing units share a special structured memory unit known as 'the grill', which stores LISP programs. Access to the grill is controlled by a device known as 'the distributor', which also allocates nodes to be evaluated to idle processors. Communication between the processing units and the distributor is by two time-shared buses. The AHR is hosted by a commercial minicomputer.

References: [A188, A189]

A2.8.8 Cm*

Carnegie-Mellon University, Pittsburgh (1977).

This is a hierarchical machine, with three levels of buses. Each processing unit is a DEC LSI-11 and this has a direct link to a local memory unit, to which it attempts to confine its operations. The 256 M-byte virtual store is distributed amongst the processors in 64 K-byte sections. The store is segmented, with a 4 K-byte segment size, and extensive permission bits (known as capabilities). Up to 14 processors make up each cluster, which are in turn connected by an intra-cluster bus, arbited by a microprogrammed bus controller. Requests to non-local memory are packet-switched to the appropriate processor, via an inter-cluster bus if necessary. An inter-processor message passing system is implemented for software synchronisation.

References: [A190, A191, A192, A193]

A2.8.9 The Columbia Homogeneous Parallel Processor (CHOPP)

Columbia University (1977).

CHOPP is a shared-memory machine designed to have a large number of processing units (of the order of 10^5–10^6), interconnected by a packet-switched binary n-cube network. A shared memory is distributed across the machine, and is also located at the nodes of the binary n-cube network. The processors are to be implemented using custom VLSI chips.

References: [A194, A195]

A2.8.10 The Heterogeneous Element Processor (HEP)

Denelcor Corporation (1978).

The HEP is a shared-memory multiprocessor in which each data processing unit is highly pipelined. In order to avoid a problem which frequently occurs in pipelined machines, that of maintaining the pipeline in a full state, the HEP performs a process change on every machine cycle. In this way, several completely independent instructions (from different instruction streams) are in a state of partial execution on a single processor, at any time. A complete set of registers are kept in the processor, for each process, and up to 128 processes may simultaneously be active on one processor. The processors are connected, through an interconnection network, to a set of memory units in which the memory locations are 64-bits wide and, in addition, contain a single full-or-empty bit. This allows the hardware to trap any read-from-empty or write-to-full operations, as neither of these are allowed in the programming environment supported by this machine. A section of this shared memory appears as a local store to each processors. Programs are stored in a local instruction memory unit, and a device known as the scheduler function unit arbitrates accesses to the shared memory.

References: [A196, A197]

A2.8.11 The Applicative Multi-Processor System (AMPS)

University of Utah (1979).

AMPS is a tree-structured machine, in which each leaf contains a data processing unit and a data memory unit, and interior nodes contain communications and load balancing units. The data processors are custom-built LISP-oriented microprocessors. The memory is distributed in 64 K-word blocks, but these are globally accessed, and appear as a one-level store. The primary programming language is flowgraph LISP.

References: [A198, A188]

A2.8.12 The Burroughs Flow Model Processor (FMP)

Burroughs Corporation / NASA (1979).

The Burroughs FMP was designed in response to the NASA requirement for a large machine for aerodynamic simulation (to be known as the NASF), and is based on the design of the Burroughs BSP. The FMP has 512 48-bit floating-point DPUs which are connected to 521 memory units by two packet-switched omega networks. Unlike the BSP, each data processing unit has its own IPU and operates independently. It is, therefore, classified as a shared-memory multiprocessor. In addition to the 34 M-word distributed memory, each processor also has a 16 K-word local DMU, and a large semiconductor file memory is provided as a fast disc buffer.

References: [A7, A199]

A2.8.13 The Paracomputer

The Courant Institute, New York University (1980).

This machine was defined to provide a theoretical 'yardstick' against which the performance of other multiprocessors may be measured. It consists of a number of identical processors connected to a large global memory which may be totally transparently accessed. The designer describes it as unrealizably powerful.

References: [A200]

A2.8.14 The Texas Reconfigurable Array Computer (TRAC)

University of Texas, Austin, Texas (1980).

TRAC is a shared-memory multiprocessor in which 16 data processing units are connected to a group of 81 memory units by a 4-level banyan network (fan-out 3, spread 2). The memory units are divided into private data memory units, shared data memory units, and instruction memory units. Index registers in the memory units allow a short-form (8-bit) address to be transmitted through the (8-bit wide) network, instead of full-length addresses, which require a number of transfers to be made. A shared mass-store device is also provided. The processors may be dynamically partitioned into groups which are controlled by a single instruction stream, broadcast through the network by an instruction memory unit, and the name of the machine is derived from this feature.

References: [A201, A202]

A2.8.15 The Applicative Language Idealized Computing Engine (ALICE)

Imperial College, London (1981).

ALICE is a shared-memory multiprocessor based on a graph-reduction model of computing. In this machine, a program exists in the form of a large number of data packets which are placed in a shared store known as the 'pool'. Each data packet comprises an operator, some operands and a pointer to the destination of the result. A number of processing 'agents' (DPUs) repeatedly remove data packets from the pool and re-write the packet, if this is possible. By repeated application of this process, all packets are evaluated, and the program is, thus, executed.

References: [A204, A205]

A2.8.16 SYstème MultiProcesseur Adapté au Traitement d'Images (SY.MP.A.T.I.)

CERFIA-UPS, Toulouse (1981).

The SY.MP.A.T.I. system comprises sixteen 8-bit processing units, each connected to a local image memory. Images are distributed across these local memories in an interleaved fashion, so that vertically adjacent pixels reside in the same local memory, and horizontally adjacent pixels reside in adjacent memory units. The processing units are linked by a packet-switched ring. A time-shared bus connects the processors to a collection of other modules. These include two memory modules containing entire images, and a number of input and output units. Although SY.MP.A.T.I. is an shared-memory multiprocessor, it may operate in an SIMD mode.

References: [A206, A208, A209]

A2.8.17 PUMPS

School of Electrical Engineering, Purdue University, Lafayette, Indiana (1982).

PUMPS is a shared-memory multiprocessor machine, specially designed for image analysis, in which a number of task processing units (DPUs), interconnected by a time-shared bus, are connected via a delta network to a shared memory. In addition to a shared-memory cache, each processor has a private memory, with its own private cache. Block transfers are carried out, through the delta network, to and from the local stores and the shared memory. Both shared and private memory areas use paged virtual memory systems. All task processing units are linked, through an indirect binary n-cube network, to a group of specialised processors. These include peripheral processors, which handle input and output, and 'VLSI units', which are special-purpose processing units, such as matrix multipliers and language recognisers. A back-end sequential processor provides an image database, linked to the shared memory, and a front-end processor is also provided.

References: [A210]

A2.8.18 The Ultracomputer

The Courant Institute, New York University (1982).

The Ultracomputer is designed to implement, as far as possible, the ideal Paracomputer [A200]. The design uses a large number of processing units (up to 4096), each with some private memory, connected to a distributed global store. The interconnection network used is a packet-switched omega network. The processing units used are based on the CDC 6600 CPU. The memory units have additional hardware to queue requests, and to perform fetch-and-add instructions. The machine has only been simulated, and construction has not yet commenced.

References: [A212, A213, A214]

A2.8.19 Macsym

Kyoto University, Japan (1983).

In Macsym, sixteen identical processing units (one of which is designated 'master') have access to an asynchronous time-shared bus. The processing units are based on the Zilog Z8001 microprocessor, and have these are provided with local store units (quoted to be 12 Gigabytes), and a shared memory of 14 M-bits is provided via the bus. A large number of image-oriented peripherals are attached directly to the processing units.

References: [A23]

A2.8.20 The C-VAS 3000

Visual Machines Ltd., Manchester (1984).

The C-VAS 3000 system is a multiprocessor machine, specialised towards image analysis. A Motorola MC68000 series processor running UNIX† is used for system management and execution of Pascal programs. An AMD2900 series bit-slice processor is used for execution of image analysis operations, and a memory address processor is provided for image address calculations. A 2 M-byte store is shared by the MC68000 and the AMD2900, in addition to the image store of between 1 and 8 M-bytes.

References: [A215]

A2.8.21 MU6-V

Department of Computer Science, University of Manchester (1985).

MU6-V is a shared-memory multiprocessor machine, in which each processing unit comprises a scalar processor and a vector-serial processor. These processing units are interconnected by a time-shared bus network. The scalar processors are based on Motorola MC68000 microprocessors. A 16-processor machine is implemented but, in this prototype, the vector-serial units are not constructed in hardware, and vector instructions are trapped and executed as 'extracodes' by the MC68000.

References: [A216]

† UNIX is a registered trademark of AT&T Bell Laboratories in the USA and other countries.

A2.9 Data Flow Machines

A2.9.1 The MIT Data Flow Machine

Massachusetts Institute of Technology (1975).

In this machine, data flow programs are stored in a special VLSI structure composed of 'instruction cells'. Each cell contains three registers, one for the instruction, and two for operands. When all the operands required by an instruction have arrived, a signal is sent to an arbitration network, which then routes the executable package to a processing unit. The instruction cell address(es) to which the results are to be sent is contained within the instruction. A special structure store is provided, to store large history-sensitive data structures, which are not easily dealt with in the functional manner of data flow programs.

References: [A218, A220]

A2.9.2 The Irvine Data Flow Machine

University of California at Irvine (later, at Massachusetts Institute of Technology) (1977).

The Irvine data flow project was started at the University of California at Irvine, and was transferred when Arvind moved to MIT. A number of major changes to the system architecture have been made since the commencement of the project, and the machine described here is the latest described version [A225]. The machine is based on VLSI units, each of which contains a packet-switched ring around which tagged data flow tokens may circulate. On entering the unit, a token is compared with all elements in a matching store, and if no match is found, the token is placed in that store. When two matching tokens are found, these proceed to the instruction fetch section, and then to an ALU, where the instruction is executed. The tokens generated by this operation are tagged, and by means of this tag are routed to the appropriate unit to meet up with further matching tokens. This may involve routing to another chip, or re-circulation around to the input section of the same chip. This machine is now very similar to the Manchester data flow machine, other than in its VLSI implementation.

References: [A222, A154, A223, A225]

A2.9.3 The Generalised Control Flow Machine (GCF Machine)

University of Newcastle upon Tyne (1979).

The concept of control flow is similar to that of data flow, but the readiness of data is indicated by sending control tokens, instead of sending the entire data item itself. Data items reside in a conventional store, and control tokens are passed along a packet-switched ring network containing a number of different units. Code Memory Processors (IPUs) change Next Instruction Address (NIA) tokens into instructions. Data Computation Units (combined DPU and DMU) perform computations, and produce more NIA packets.

References: [A226]

A2.9.4 The Manchester Data Flow Machine

Department of Computer Science, University of Manchester (1979).

The Manchester data flow machine is designed to execute programs in the form of data flow graphs. Data tokens are stored in a matching store, until all operands required for an operation are present. When the last operand required for an instruction arrives at the matching store, an executable packet is passed to the node store, where the instruction is obtained, and the packet proceeds to one of sixteen AMD2900-based execution units. Each token is 'coloured' to allow recursion without mis-matching of tokens from different calls of the same procedure. Results are sent to the matching store, to meet up with more operands, and so the machine forms a ring, around which tokens circulate. A switch in the ring allows tokens to be injected or extracted for input/output. The switch is also capable of connecting to other data flow rings, to construct large multi-ring machines.

References: [A227, A228, A229, A230, A231, A232]

A2.10 Stochastic Machines

A2.10.1 WISARD

Brunel University (1981).

The WISARD is a stochastic machine, specialised for visual pattern recognition. A group of fifteen memory units are loaded with a training set, such that when their addressing inputs are presented with a pseudo-random function of a particular input image, the output will be affirmative. These groups are known as discriminators. A conventional sequential processor reads the response from several discriminators, and thus classifies the image.

References: [A233, A234]

"The Milliard Gargantuabrain?
A mere abacus, mention it not."

Deep Thought, in
'The Hitch-Hiker's Guide to the Galaxy'
— Douglas Adams

References

[A1] **Preston, K. Jr. (1983).** "Cellular Logic Computers for Pattern Recognition". *IEEE Computer* **16(1)**, pp. 36–47.

[A2] **Ramamoorthy, C. V. and Li, H. F. (1977).** "Pipeline Architecture". *Computing Surveys* **9**, pp. 61–102.

[A3] **Doran, R. W. (1982).** "The Amdahl 470V/8 and the IBM 3033: A Comparison of Processor Designs". *IEEE Computer* **15(4)**, pp. 27–40.

[A4] **McCormick, B. H. (1963).** "The Illinois Pattern Recognition Computer — ILLIAC III". *IEEE Transactions on Electronic Computers* **EC-12**, pp. 791–813.

[A5] **Anderson, D. W., Sparacio, F. J. and Tomasulo, R. M. (1967).** "The IBM System/360 model 91 : Machine Philosophy and Instruction Handling". *IBM Journal of Research and Development* **11**, pp. 8–24.

[A6] **Ibbett, R. (1982).** "The Architecture of High Performance Computers". Macmillan Press, London.

[A7] **Hockney, R. W. and Jesshope, C. R. (1981).** "Parallel Computers". Adam Hilger, Bristol.

[A8] **IEEE Computer Society (1972).** "Proceedings of the 1972 Annual Conference — Innovative Architecture — COMPCON 72 Digest". IEEE Press, New York.

[A9] **Hintz, R. G. and Tate, D. P. (1972).** "Control Data STAR-100 Processor Design". In [A8], pp. 1–4.

[A10] **Liu, M. T. and Rothstein, J. (eds) (1981).** "Proceedings of the 1981 International Conference on Parallel Processing (Bellaire, Michigan)". IEEE Press, Piscataway, New Jersey.

[A11] **Thornton, J. E. (1981).** "Control Data 6600 and STAR-100". In [A10], pp. 33–37.

[A12] **Lincoln, N. R. (1982).** "Technology and Design Tradeoffs in the Creation of A Modern Supercomputer". *IEEE Transactions on Computers* **C-31**, pp. 349–362.

[A13] **Lincoln, N. R. (1983).** "Supercomputers = Colossal Computations + Enormous Expectations + Renowned Risk". *IEEE Computer* **16(5)**, pp. 38–47.

[A14] **Kilburn, T., Edwards, D. G. B., Lanigan, M. J. and Sumner, F. H. (1962).** "One-Level Storage System". *IRE Transactions on Electronic Computers* **EC-11**, pp. 223–235.

[A15] **Iverson, K. E. (1962).** "A Programming Language". John Wiley and Sons, New York.

[A16] **IEEE Computer Society (1980).** "Proceedings of the Fifth International Conference on Pattern Recognition (Miami Beach, Florida), Volume 1". IEEE Press, Piscataway, New Jersey.

[A17] **Duin, R. P. W., Gerritsen, F. A., Groen, F. C. A., Verbeek, P. W. and Verhagen, C. J. D. M. (1980).** "The Delft Image Processing System, Design and Use". In [A16], pp. 768–774.

[A18] **Duff, M. J. B. and Levialdi, S. (eds) (1981).** "Languages and Architectures for Image Processing". Academic press, London.

[A19] **Gerritsen, F. A. and Monhemius, R.D. (1981).** "Evaluation of the Delft Image processor DIP-1". In [A18], pp. 189–203.

[A20] **Gerritsen, F. A. and Aardema, L. G. (1981).** "Design and Use of DIP-1: A Fast, Flexible and Dynamically Microprogrammable Pipelined Image Processor". *Pattern Recognition* **14**, pp. 319–330.

[A21] **Dixon, R. N. (1980).** "Aspects of Picture Processing". Ph.D Thesis, Department of Medical Biophysics, University of Manchester.

[A22] **Mori, K., Kidode, M., Shinoda, H. and Asada, H. (1978).** "Design of a Local Parallel Pattern Processor for Image Processing". *AFIPS NCC* **47**, pp. 1025–1031.

[A23] **Kidode, M. (1983).** "Image Processing Machines in Japan". *IEEE Computer* **16(1)**, pp. 68–80.

[A24] **Preston, K. Jr. (1981).** "Comparison of Parallel Processing Machines: A Proposal". In [A18], pp. 305–324.

[A25] **Preston, K. Jr. and Duff, M. J. B. (1984).** "Modern Cellular Automata". Plenum Press, New York.

[A26] **IEEE Computer Society (1980).** "Proceedings of the Workshop on Picture Data Description and Management (Asilomar, Pacific Grove, California)". IEEE Press, Los Alamitos, California.

[A27] **Danielsson, P. E., Kruse, B. and Gudmundsson, B. (1980).** "Memory Hierarchies in PICAP II". In [A26], pp. 275–280.

[A28] **IEEE Computer Society (1981).** "IEEE Computer Society Workshop on Computer Architecture for Pattern Analysis and Image Database Mangement (Hot Springs, Virginia)". IEEE Computer Society Press, Piscataway, New Jersey.

[A29] **Antonsson, D., Danielsson, P. E., Gudmundsson, B., Hedblom, T., Kruse, B., Linge, A., Lord, P. and Ohlsson, T. (1981).** "PICAP — A System Approach to Image Processing". In [A28], pp. 35–42.

[A30] **Antonsson, D., Gudmundsson, B., Hedblom, T., Kruse, B., Linge, A., Lord, P. and Ohlsson, T. (1982).** "Picap II — A System Approach to Image Processing". *IEEE Transactions on Computers* **C-31**, pp. 997–1000.

[A31] **Golay, M. J. E. (1969).** "Hexagonal Pattern Transformations". *IEEE Transactions on Computers* **C-18**, pp. 733–740.

[A32] **Gray, S. B. (1971).** "Local Properties of Binary Images in Two Dimensions". *IEEE Transactions on Computers* **C-20**, p. 551.

[A33] **IEEE Computer Society (1976).** "Proceedings of the Third International Joint Conference on Pattern Recognition (Coronado, California)". IEEE Press, Long Beach, California.

[A34] **Kruse, B. (1976).** "The PICAP Picture Processing Laboratory". In [A33], pp. 875–881.

[A35] **Kruse, B. (1978).** "Experience with a Picture Processor in Pattern Recognition Processing". *AFIPS NCC* **47**, pp. 1015–1024.

[A36] **Tanimoto, S. and Klinger, A. (eds) (1980).** "Structured Computer Vision — Machine Perception Through Hierarchical Computation Structures". Academic Press, London.

[A37] **Kruse, B. (1980).** "System Architecture for Image Analysis". In [A36], pp. 169–212.

[A38] **Lantuejoul, C. (1981).** "An Image Analyser". In [A18], pp. 165–177.

[A39] **Control Data Corporation (1981).** "CDC Cyber 200 Model 205 Computer System Hardware Reference Manual". CDC Publications and Graphics Division, St. Paul, Minnesota.

[A40] **Morris, D. and Ibbett, R. (1979).** "The MU5 Computer System". Macmillan Press, London.

[A41] **Joyce-Loebl (Vickers) Limited (1981).** "Magiscan 2 Pascal Manual". Joyce-Loebl (Vickers) Limited, Newcastle-upon-Tyne.

[A42] **Duff, M. J. B. (ed) (1986).** "Intermediate Level Image Processing". Academic Press, London.

[A43] **Taylor, C. J., Dixon, R. N., Gregory, P. J. and Graham, J. (1986).** "An Architecture for Integrating Symbolic and Numerical Image Processing". In [A42], pp. 19–34.

[A44] **Hobbs, L. C., Theis, D. J., Trimble, J., Titus, H. and Highberg, I. (eds) (1970).** "Parallel Processor Systems, Technologies, and Applications". Spartan books, New York.

[A45] **Shooman, W. (1970).** "Orthogonal Processing". In [A44], pp. 297–308.

[A46] **Higbie, L. C. (1972).** "The Omen Computers: Associative Array Processors". In [A8], pp. 287–290.

[A47] **Rudolph, J. A. (1972).** "A Production Implementation of an Associative Processor — STARAN". *AFIPS FJCC* **41**, pp. 229–241.

[A48] **Batcher, K. E. (1974).** "STARAN Parallel Processor System Hardware". *AFIPS NCC* **43**, pp. 405–410.

[A49] **Enslow, P. H. Jr. (ed) (1974).** "Multiprocessors and Parallel Processing". John Wiley, New York.

[A50] **Enslow, P. H. Jr. (ed) (1976).** "Proceedings of the 1976 International Conference on Parallel Processing (Waldenwood, Michigan)". IEEE Press, Piscataway, New Jersey.

[A51] **Batcher, K. E. (1976).** "The Flip Network in STARAN". In [A50], pp. 65–71.

[A52] **Pergamon Infotech (1976).** "Infotech State of the Art Report on Multiprocessor Systems". Pergamon Infotech, Maidenhead, U. K..

[A53] **Meilander, W. C. (1976).** "STARAN, an Associative Approach to Multiprocessor Architecture". In [A52], pp. 345–372.

[A54] **Potter, J. L. (1978).** "The STARAN Architecture and Its Application to Image Processing and Pattern Recognition Algorithms". *AFIPS NCC* **47**, pp. 1041–1047.

[A55] **Meilander, W. C. (1981).** "History of Parallel Processing at Goodyear Aerospace". In [A10], pp. 5–15.

[A56] **Lipovski, G. J. and Szygenda, S. A. (eds) (1973).** "Proceedings of the First Annual Symposium on Computer Architecture (Gainesville, Florida)". Published as *Computer Architecture News* **2(4)**.

[A57] **Reddaway, S. F. (1973).** "DAP — A Distributed Processor Array". In [A56], pp. 61–65.

[A58] **Kuck, D. J., Lawrie, D. H. and Sameh, A. H. (eds) (1977).** "High Speed Computer and Algorithm Organisation". Academic Press, New York.

[A59] **Flanders, P. M., Hunt, D. J., Reddaway, S. F. and Parkinson, D. (1977).** "Efficient High Speed Computing With the Distributed Array Processor". In [A58], pp. 113–127.

[A60] **Pergamon Infotech (1979).** "Infotech State of the Art Report on Supercomputers, Volume 2". Pergamon Infotech, Maidenhead, U. K..

[A61] **Flanders, P. M. (1979).** "Fortran Extensions for a Highly Parallel Processor". In [A60], pp. 117–133.

[A62] **Reddaway, S. F. (1979).** "The DAP Approach". In [A60], pp. 309–329.

[A63] **Hunt, D. J. (1981).** "The ICL DAP and its Application to Image Processing". In [A18], pp. 275–282.

[A64] **Duff, M. J. B. (ed) (1983).** "Computing Structures for Image Processing". Academic Press, London.

[A65] **Fountain, T. J. (1983).** "Bit-Serial Array Processor Circuits". In [A64], pp. 1–14.

[A66] **Gerritsen, F. A. (1983).** "A Comparison of the CLIP4, DAP and MPP Processor Array Implementations". In [A64], pp. 15–30.

[A67] **Smith, K. A. (1983).** "DIMP: DAP Image Manipulation Package". DAP Support Unit Internal Report, Queen Mary College, London.

[A68] **Paddon, D. J. (ed) (1984).** "Supercomputers and Parallel Computation". Oxford University Press, Oxford.

[A69] **Davies, S. T. (1984).** "The Implementation of the FFT on the DAP". In [A68], pp. 195–208.

[A70] **Slotnick, D.L., Borck, W. C., McReynolds, R. C. (1962).** "The Solomon Computer". *AFIPS FJCC* **22**, pp. 97–107.

[A71] **Gregory, J. and McReynolds, R. (1963).** "The SOLOMON computer". *IEEE Transactions on Electronic Computers* **EC-12**, pp. 774–781.

[A72] **Slotnick, D. L. (1981).** "Centrally-Controlled Parallel Processors". In [A10], pp. 16–24.

[A73] **Barnes, G. H., Brown, R. M., Kato, M., Kuck, D. J., Slotnick, D. L. and Stokes, R. A. (1968).** "The ILLIAC IV Computer". *IEEE Transactions on Computers* **C-17**, pp. 746–757.

[A74] **Davis, R. L. (1969).** "The ILLIAC IV Processing Element". *IEEE Transactions on Computers* **C-18**, pp. 800–816.

[A75] **Barnes, G. H. (1972).** "The Use of ILLIAC IV". In [A8], pp. 295–29 6.

[A76] **Bouknight, W. J., Denenberg, S. A., McIntyre, D. E., Randall, J. M., Sameh, A. H. and Slotnick, D. L. (1972).** "The Illiac IV System". *Proceedings of the IEEE* **60**, pp. 369–388.

[A77] **Stokes, R. and Cantarella, R. (1981).** "The History of Parallel Processing at Burroughs". In [A10], pp. 25–32.

[A78] **Githens, J. A. (1970).** "An Associative, Highly-Parallel Computer for Radar Data Processing". In [A44], pp. 71–86.

[A79] **Crane, B. A., Gilmartin, M. J., Huttenhoff, J. H., Rux, P.T. and Shively, R. R. (1972).** "PEPE Computer Architecture". In [A8], pp. 57–60.

[A80] **Wilson, D. E. (1972).** "The PEPE Support Software System". In [A8], pp. 61–64.

[A81] **Cornell, J. A. (1976).** "PEPE: Parallel Element Processing Ensemble". In [A52], pp. 171–190.

[A82] **Mariani, M. P. and Henry, E. J. (1978).** "PEPE — A Users Viewpoint — A Powerful Real Time Adjunct". *AFIPS NCC* **47**, pp. 993–1002.

[A83] **Vick, C. R. and Cornell, J. (1978).** "PEPE Architecture — Present and Future". *AFIPS NCC* **47**, pp. 981–992.

[A84] **Onoe, M., Preston K. Jr. and Rosenfeld, A. (eds) (1980).** "Real-Time Medical Image Processing". Plenum Press, New York.

[A85] **Graham, M. D. and Norgren, P. E. (1980).** "The Diff3 Analyzer: A Parallel/Serial Golay Image Processor". In [A84], pp. 163–182.

[A86] **Duff, M. J. B. (1976).** "CLIP, an Array Processor for Image Processing". In [A52], pp. 191–203.

[A87] **Duff, M. J. B. (1976).** "CLIP4: A Large Scale Integrated Circuit Array Parallel Processor". In [A33], pp. 728–733.

[A88] **Batchelor, B. G. (ed) (1978).** "Pattern Recognition. Ideas in Practice". Plenum Press, New York.

[A89] **Duff, M. J. B. (1978).** "Parallel Processing Techniques". In [A88], pp. 145–174.

[A90] **Kuck, D. J. and Stokes, R. A. (1982).** "The Burroughs Scientific Processor (BSP)". *IEEE Transactions on Computers* **C-31**, pp. 363–376.

[A91] **Duff, M. J. B. (1978).** "Review of the CLIP Image Processing System". *AFIPS NCC* 47, pp. 1055–1060.

[A92] **Stucki, P. (ed) (1979).** "Advances in Digital Image Processing". Plenum Press, New York.

[A93] **Duff, M. J. B. (1979).** "Parallel Processors for Digital Image Processing". In [A92], pp. 265–276.

[A94] **Fountain, T. J. and Goetcherian, V. (1980).** "CLIP4 Parallel Processing System". *IEE Proceedings* **127E**, pp. 219–224.

[A95] **Fountain, T. J. (1981).** "CLIP4: A Progress Report". In [A18], pp. 283–291.

[A96] **Reeves, A. P. (1980).** "A Systematically Designed Binary Array Processor". *IEEE Transactions on Computers* **C-29**, pp. 278–287.

[A97] **Reeves, A. P. (1981).** "The Anatomy of VLSI Binary Array Processors". In [A18], pp. 267–274.

[A98] **Reeves, A. P. (1981).** "Parallel Computer Architectures for Image Processing". In [A10], pp. 199–206.

[A99] **Batcher, K. E. (1980).** "Design of a Massively Parallel Processor". *IEEE Transactions on Computers* **C-29**, pp. 836–840.

[A100] **IEEE and ACM (1980).** "Proceedings of the Seventh Annual Symposium on Computer Architecture (La Baule, France)". Published as *Computer Architecture News* **8(3)**.

[A101] **Batcher, K. E. (1980).** "Architecture of a Massively Parallel Processor". In [A100], pp. 168–173.

[A102] **Potter, J. L. (1981).** "Continuous Image Processing on the MPP". In [A28], pp. 51–56.

[A103] **Batcher, K. E. (1982).** "Bit-Serial Parallel Processing Systems". *IEEE Transactions on Computers* **C-31**, pp. 377–384.

[A104] **Kushner, T., Wu, A. Y. and Rosenfeld, A. (1982).** "Image Processing on MPP: 1". *Pattern Recognition* **15(3)**, pp. 121–130.

[A105] **Potter, J. L. (1983).** "Image Processing on the Massively Parallel Processor". *IEEE Computer* **16(1)**, pp. 62–67.

[A106] **Herron, J. M., Farley, J., Preston K. Jr. and Sellner, H. (1982).** "A General Purpose High-Speed Logical Transform Image Processor". *IEEE Transactions on Computers* **C-31**, pp. 795–800.

[A107] **Parvin, B. (1980).** "A Microprogrammable Vector Processor for Image Processing Application". In [A26], pp. 287–292.

[A108] **Fountain, T. J. (1981).** "Towards CLIP6 — An Extra Dimension". In [A28], pp. 25–30.

[A109] **Tanimoto, S. L. and Pfeiffer, J. J. Jr. (1981).** "An Image Processor Based on an Array of Pipelines". In [A28], pp. 201–208.

[A110] **Danielsson, P. E. and Ericsson, T. (1983).** "LIPP — Proposals for the Design of an Image Processor Array". In [A64], pp. 157–178.

[A111] **Fountain, T. J. (1983).** "The Development of the CLIP7 Image Processing System". *Pattern Recognition Letters* **1**, pp. 331–340.

[A112] **Matthews, K. N. (1983).** "Large Window Median Filtering on CLIP7". *Pattern Recognition Letters* **1**, pp. 341–346.

[A113] **Graham, M. D. (1983).** "The Diff-4: A Second Generation Slide Analyser". In [A64], pp. 179–194.

[A114] **Jesshope, C. R. (1984).** "A Reconfigurable Processor Array for VLSI". In [A68], pp. 35–40.

[A115] **International Association for Pattern Recognition (1986).** "Proceedings of the Eighth International Conference on Pattern Recognition (Paris, France), Volume 2". IEEE Computer Society Press, Piscataway, New Jersey.

[A116] **Lindskog, B. and Danielsson, P. E. (1986).** "A Parallel Processor Tuned for 3D Image Operations". In [A115], pp. 1248–1250.

[A117] **Thompson, C. D. and Kung, H. T. (1977).** "Sorting on a Mesh-Connected Parallel Computer". *Communications of the ACM* **20**, pp. 263–270.

[A118] **Kung, H. T. (1980).** "The Structure of Algorithms". *Advances in Computers* **19**, pp. 65–112.

[A119] **Kung, H. T. and Picard, R. L. (1981).** "Hardware Pipelines for Multi-Dimensional Convolution and Resampling". In [A28], pp. 273–278.

[A120] **Kung, H.T. (1982).** "Why Systolic Architectures?". *IEEE Computer* **15(1)**, pp. 37–46.

[A121] **Gelsema, E. S. and Kanal, L. N. (eds) (1980).** "Pattern Recognition in Practice". North-Holland Publishing Company, Amsterdam.

[A122] **Sternberg, S. R. (1980).** "Language and Architecture for Parallel Image Processing". In [A121], pp. 35–44.

[A123] **Sternberg, S. R. (1980).** "Cellular Computers and Biomedical Image Processing". In [A84], pp. 11–22.

[A124] **Lougheed, R. M., McCubbrey, D. L. and Sternberg, S. R. (1980).** "Cytocomputers: Architectures for Image Processing". In [A26], pp. 281–286.

[A125] **Lougheed, R. M. and McCubbrey, D. L. (1980).** "The Cytocomputer: A Practical Pipelined Image Processor". In [A100], pp. 271–278.

[A126] **Sternberg, S. R. (1983).** "Biomedical Image Processing". *IEEE Computer* **16(1)**, pp. 22–34.

[A127] **Snyder, L., Jamieson, L. J., Gannon, D. B. and Siegel, H. J. (eds) (1985).** "Algorithmically Specialised Parallel Computers: Proceedings of the Workshop on Algorithmically Specialised Computer Organisations (Purdue University, September 1982)". Academic Press, Orlando, Florida.

[A128] **Sternberg, S. R. (1985).** "Computer Architectures Specialised for Mathematical Morphology". In [A127], pp. 169–176.

[A129] **Yen, D. W. L. and Kulkarni, A. V. (1981).** "The ESL Systolic Array Processor for Signal and Image Processing". In [A28], pp. 265–272.

[A130] **Kulkarni, A. V. and Yen, D. W. L. (1982).** "Systolic Processing and an Implementation for Signal and Image Processing". *IEEE Transactions on Computers* **C-31**, pp. 1000–1009.

[A131] **Luetjen, K., Gemmar, P. and Ischen, H. (1980).** "FLIP: A Flexible Multiprocessor System for Image Processing". In [A16], pp. 326–328.

[A132] **Gemmar, P., Ischen, H. and Luetjen, K. (1981).** "FLIP: A Multiprocessor System For Image Analysis.". In [A18], pp. 245–256.

[A133] **Snyder, L. (1982).** "Introduction to the Configurable Highly Parallel Computer". *IEEE Computer* **15(1)**, pp. 47–64.

[A134] **Snyder, L. (1984).** "Supercomputers and VLSI". *Advances in Computers* **23**, pp. 2–33.

[A135] **Shapiro, E. Y. (1983).** "The Bagel: A Systolic Concurrent Prolog Machine". ICOT Research Centre Technical Memorandum TM-0031.

[A136] **Inmos Limited (1984).** "IMS T424 Transputer". Product information, Inmos Limited, Bristol.

[A137] **IEEE and ACM (1985).** "Proceedings of the Twelfth Annual Symposium on Computer Architecture". Published as *Computer Architecture News* **13(3)**.

[A138] **Whitby-Strevens, C. (1985).** "The Transputer". In [A137], pp. 292–300.

[A139] **Lipovski, G. J. (1970).** "The Architecture of a Large Associative Processor". *AFIPS SJCC* **36**, pp. 385–396.

[A140] **Manara, R. and Stringa, L. (1981).** "The EMMA System: An Industrial Experience on a Multiprocessor". In [A18], pp. 215–227.

[A141] **Peterson, J. L. (1977).** "Petri Nets". *Computing Surveys* **9**, pp. 223–252.

[A142] **Tucker, J. H. (1976).** "CERVISCAN: An Image Analysis System for Experiments in Automatic Cervical Smear Prescreening". *Computers and Biomedical Research* **9**, pp. 93–107.

[A143] **IEEE and ACM (1977).** "Proceedings of the Fourth Annual Symposium on Computer Architecture". Published as *Computer Architecture News* **5(7)**.

[A144] **Harris, J. A. and Smith, D. R. (1977).** "Hierarchical Multiprocessor Organisations". In [A143], pp. 41–48.

[A145] **IEEE and ACM (1979).** "Proceedings of the Sixth Annual Symposium on Computer Architecture". Published as *Computer Architecture News* **7(6)**.

[A146] **Harris, J. A. and Smith, D. R. (1979).** "Simulation Experiments of a Tree Organised Computer". In [A145], pp. 68–74.

[A147] **Ichikawa, T., Sakamura, K. and Aiso, H. (1978).** "A Multi-Microprocessor ARES with Associative Processing Capability on Semantic Databases". *AFIPS NCC* **47**, pp. 1033–1039.

[A148] **Bracken, P. A., Dalton, J. T., Quann, J. J. and Billingsley, J. B. (1978).** "AOIPS — An Interactive Image Processing System". *AFIPS NCC* **47**, pp. 159–171.

[A149] **Edwards, D. G. B., Knowles, A. E. and Woods, J. V. (1980).** "MU6-G. A New Design to Achieve Mainframe Performance From a Mini-Sized Computer". In [A100], pp. 161–167.

[A150] **Dawson, J. M., Huff, R. W. and Wu, C. (1978).** "Plasma Simulation on the UCLA CHI Computer System". *AFIPS NCC* **47**, pp. 395–407.

[A151] **IEEE and ACM (1978).** "Proceedings of the Fifth Annual Symposium on Computer Architecture". Published as *Computer Architecture News* **6(5)**.

[A152] **Despain, A. M. and Patterson, D. A. (1978).** "X-tree: A Tree Structured Multi-Processor Computer Architecture". In [A151], pp. 144–151.

[A153] **Patterson, D. A., Fehr, E. S. and Sequin, C. H. (1979).** "Design Considerations for the VLSI Processor of X-TREE". In [A145], pp. 90–101.

[A154] **Treleaven, P. C. (1982).** "VLSI Processor Architectures". *IEEE Computer* **15(6)**, pp. 33–45.

[A155] **Seitz, C. L. (ed) (1981).** "Proceedings of the Second Caltech Conference on Very Large Scale Integration (California Institute of Technology, Pasadena)". California Institute of Technology, Pasadena.

[A156] **Browning, S. A. and Seitz, C. L. (1981).** "Communication in a Tree Machine". In [A155], pp. 509–526.

[A157] **Rieger, C., Bane, J. and Trigg, R. (1980).** "ZMOB: A Highly Parallel Multiprocessor". Technical Report TR911, Department of Computer Science, University of Maryland.

[A158] **Rieger, C., Bane, J. and Trigg, R. (1980).** "ZMOB: A Highly Parallel Multiprocessor". In [A26], pp. 298–304.

[A159] **Bane, R., Stanfill, C. and Weiser, M. (1981).** "Operating System Strategy on ZMOB". In [A28], pp. 125–132.

[A160] **Trigg, R. (1981).** "Software on ZMOB: An Object-Oriented Approach". In [A28], pp. 133–140.

[A161] **Kushner, T., Wu, A. Y. and Rosenfeld, A. (1982).** "Image Processing on ZMOB". *IEEE Transactions on Computers* **C-31**, pp. 943–951.

[A162] **Granlund, G. H. (1981).** "GOP: A Fast and Flexible Processor for Image Analysis". In [A18], pp. 179–188.

[A163] **Granlund, G. H., Antonsson, D., Arvidsson, J., Hedlund, M., Henden, P., Knutsson, H., Lundgren, K., Nilsson, B., von Post, B. and Wilson, R. (1981).** "The GOP Image Processor". In [A28], pp. 195–200.

[A164] **Granlund, G. H. (1978).** "In Search of a General Picture Processing Operator". *Computer Graphics and Image Processing* **8**, pp. 155–173.

[A165] **Hillis, W. D. (1981).** "The Connection Machine (Computer Architecture for the New Wave)". Massachusetts Institute of Technology, Artificial Intelligence Laboratory A. I. Memorandum 646.

[A166] **Hillis, W. D. (1985).** "The Connection Machine". MIT Press, Cambridge, Massachusetts.

[A167] **Hillis, W. D. (1987).** "The Connection Machine". *Scientific American* **256(6)**, pp. 86–93.

[A168] **Joshi, A. (ed) (1976).** "Proceedings of the Ninth International Joint Conference on Artificial Intelligence (Los Angeles, California), Volume 1". Morgan Kaufmann Publishers, Los Altos, California.

[A169] **Touretzky, D. S. and Hinton, G. E. (1985).** "Symbols Among the Neurons: Details of a Connectionist Inference Architecture". In [A168], pp. 238–243.

[A170] **Uhr, L., Thompson, M. and Lackey, J. (1981).** "A 2-Layered SIMD/MIMD Parallel Pyramidal 'Array Net"'. In [A28], pp. 209–216.

[A171] **Siegel, L. J. (1981).** "Image Processing on a Partitionable SIMD Machine". In [A18], pp. 294–300.

[A172] **Kuehn, J. T. and Siegel, H. J. (1981).** "Simulation Studies of PASM in SIMD Mode". In [A28], pp. 43–50.

[A173] **Siegel, L. J., Siegel, H. J. and Feather, A. E. (1982).** "Parallel Approaches to Image Correlation". *IEEE Transactions on Computers* **C-31**, pp. 208–218.

[A174] **Warpenburg, M. R. and Siegel, L. J. (1982).** "SIMD Image Resampling". *IEEE Transactions on Computers* **C-31**, pp. 934–942.

[A175] **Siegel, H. J., Schwederski, T., Davis, N. J. IV and Kuehn, J. T. (1984).** "PASM: A Reconfigurable Parallel System for Image Processing". *Computer Architecture News* **12(4)**, pp. 7–19.

[A176] **Adams, G. B. and Siegel, H. J. (1982).** "On the Number of Permutations Performable on the Augmented Data Manipulator Network". *IEEE Transactions on Computers* **C-31**, pp. 270–277.

[A177] **Preston, K. Jr. and Uhr, L. (eds) (1982).** "Multicomputers and Image Processing". Academic Press, New York.

[A178] **Tanimoto, S. L. (1982).** "Programming Techniques for Hierarchical Parallel Image Processors". In [A177], pp. 421–429.

[A179] **Kung, S. Y., Arun, K. S., Gal-Ezer, R. J. and Rao, D. V. B. (1982).** "Wavefront Array Processor: Language, Architecture and Applications". *IEEE Transactions on Computers* **C-31**, pp. 1054–1066.

[A180] **Levialdi, S. (ed) (1985).** "Integrated Technology for Parallel Image Processing". Academic Press, London.

[A181] **Cantoni, V., Ferretti, M., Levialdi, S. and Maloberti, F. (1985).** "A Pyramid Project Using Integrated Technology". In [A180], pp. 121–132.

[A182] **Merigot, A., Garda, P., Zavidovique, B. and Devos, F. (1986).** "Interlayer Communication in MIMD Pyramidal Computer". In [A115], pp. 954–957.

[A183] **Thornton, J. E. (1970).** "Design of a Computer: The Control Data 6600". Scott Foresman and Company, Glenview, Illinois.

[A184] **Watson, W. J. (1972).** "The TI ASC — A Highly Modular and Flexible Super Computer Architecture". *AFIPS FJCC* **41**, pp. 221–228.

[A185] **Watson, W. J. (1972).** "The Texas Instruments Advanced Scientific Computer". In [A8], pp. 291–293.

[A186] **Bell, C. G. and Freeman, P. (1972).** "C.ai — A Computer Architecture For AI Research". *AFIPS FJCC* 41, pp. 779–790.

[A187] **Wulf, W. A. and Bell, C. G. (1972).** "C.mmp: A Multi-Mini-Processor". *AFIPS FJCC* 41, pp. 765–777.

[A188] **Guzmán, A. (1981).** "A Parallel Heterarchical Machine for High-Level Language Processing". In [A18], pp. 229–244.

[A189] **Guzmán, A. (1981).** "A Heterarchical, Multi-Microprocessor Lisp Machine". In [A28], pp. 309–317.

[A190] **Swan, R. J., Fuller, S. H. and Siewiorek, D. P. (1977).** "Cm* — A Modular Multi-Microprocessor". *AFIPS NCC* 46, pp. 637–663.

[A191] **Swan, R. J., Bechtolsheim, A., Lai, K. W. and Ousterhout, J. K. (1977).** "The Implementation of the Cm* Multi-Microprocessor". *AFIPS NCC* 46, pp. 645–655.

[A192] **Jones, A. K., Chansler, R. J. Jr., Durham, I., Feiler, P. and Schwans, K. (1977).** "Software Management of Cm* — A Distributed Multiprocessor". *AFIPS NCC* 46, pp. 657–663.

[A193] **Deminet, J. (1982).** "Experience with Multiprocessor Algorithms". *IEEE Transactions on Computers* C-31, pp. 278–288.

[A194] **Sullivan, H. and Bashkov, T. (1977).** "A Large Scale, Homogeneous, Fully Distributed Parallel Machine, I". In [A143], pp. 105–117.

[A195] **Sullivan, H., Bashkov, T. and Klappholz, D. (1977).** "A Large Scale, Homogeneous, Fully Distributed Parallel Machine, II". In [A143], pp. 118–124.

[A196] **Jordan, H. F. (1984).** "Experience with Pipelined Multiple Instruction Streams". *Proceedings of the IEEE* 72, pp. 113–123.

[A197] **Hiromoto, R. E., Lubek, O. M. and Moore, J. (1984).** "Experiences with the Denlecor HEP". *Parallel Computing* 1, pp. 197–206.

[A198] **Keller, R. M., Lindstrom, G. and Patil, S. (1979).** "A Loosely-Coupled Applicative Multi-Processing System". *AFIPS NCC* 48, pp. 613–622.

[A199] **Gottlieb, A. and Schwartz, J. T. (1982).** "Networks and Algorithms for Very-Large-Scale Parallel Computation". *IEEE Computer* 15(1), pp. 27–36.

[A200] **Schwartz, J. T. (1980).** "Ultracomputers". *ACM Transactions on Programming Languages and Systems* 2, pp. 484–521.

[A201] **Premkumar, U. V., Kapur, R., Malek, M., Lipovski, G. J. and Horne, P. (1980).** "Design and Implementation of the Banyan Interconnection Network in TRAC". *AFIPS NCC* 49, pp. 643–653.

[A202] **Sejnowski, M. C., Upchurch, E. T., Kapur, R. N., Charlu, D. P. S. and Lipovski, G. J. (1980).** "An Overview of the Texas Reconfigurable Array Computer". *AFIPS NCC* 49, pp. 631–641.

[A203] **Association for Computing Machinery (1981).** "Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture (Portsmouth, New Hampshire)". ACM, New York.

[A204] **Darlington, J. and Reeve, M. (1981).** "ALICE: A Multi-Processor Reduction Machine for the Parallel Evaluation of Applicative Languages". In [A203], pp. 65–75.

[A205] **Reeve, M. J. (1982).** "The ALICE Compiler Target Language". Department of Computing Internal Report, Imperial College, London.

[A206] **Basille, J. L., Castan, S. and Latil, J. Y. (1981).** "Système Multiprocesseur Adapté au Traitement d'Images". In [A18], pp. 205–213.

[A207] **Granum, E. (ed) (1981).** "Proceedings of the IVth European Chromosome Analysis Workshop (Edinburgh)". MRC Clinical and Population Cytogenetics Unit, Edinburgh.

[A208] **Basille, J. L. and Castan, S. (1981).** "A Two-Level Parallel Structure: SY.MP.A.T.I., Application to Chromosome Analysis". In [A207], pp. 1.4.1–1.4.4.

[A209] **Basille, J. L., Castan, S. and Latil, J. Y. (1982).** "A Typical Apropogation Algorithm on the Line Processor SY.MP.A.T.I.. The Region Labelling". In [A177], pp. 99–110.

[A210] **Briggs, F. A., Fu, K. S., Hwang, K. and Wah, B. W. (1982).** "PUMPS Architecture for Pattern Analysis and Database Management". *IEEE Transactions on Computers* **C-31**, pp. 969–983.

[A211] **IEEE and ACM (1982).** "Proceedings of the Ninth Annual Symposium on Computer Architecture (Austin, Texas)". Published as *Computer Architecture News* **10(3)**.

[A212] **Gottlieb, A., Grishman, R., Kruskal, C. P., McAuliffe, K. P., Rudolph, L. and Snir, M. (1982).** "The NYU Ultracomputer — Designing a MIMD, Shared-Memory Parallel Machine". In [A211], pp. 27–42.

[A213] **Gottlieb, A., Grishman, R., Kruskal, C. P., McAuliffe, K. P., Rudolph, L. and Snir, M. (1983).** "The NYU Ultracomputer — Designing an MIMD Shared Memory Parallel Computer". *IEEE Transactions on Computers* **C-32**, pp. 175–189.

[A214] **Edler, J., Gottlieb, A., Kruskal, C. P., McAuliffe, K. P., Rudolph, L., Snir, M., Teller, P. J. and Wilson, J. (1985).** "Issues Related to MIMD Shared-Memory Computers: the NYU Ultracomputer Approach". In [A137], pp. 126–135.

[A215] **Visual Machines Limited (1985).** "C-VAS 3000 Computer Vision System". Product information, Visual Machines Limited, Manchester.

[A216] **Ibbett, R. N., Capon, P. C. and Topham, N. P. (1985).** "MU6V: A Parallel Vector Processing System". In [A137], pp. 136–143.

[A217] **IEEE and ACM (1975).** "Proceedings of the Second Annual Symposium on Computer Architecture". Published as *Computer Architecture News* **3(4)**.

[A218] **Dennis, J. B. and Misunas, D. P. (1975).** "A Preliminary Architecture for a Basic Data-Flow Processor". In [A217], pp. 126–132.

[A219] **Feng, T. (ed) (1975).** "Proceedings of the 1975 Sagamore Computer Conference on Parallel Processing (Sagamore, New York)". IEEE Computer Society Press, Long Beach, California.

[A220] **Misunas, D. P. (1975).** "Structure Processing in a Data-Flow Computer". In [A219], pp. 230–235.

[A221] **Gilchrist, B. (ed) (1977).** "Information Processing 77 — Proceedings of IFIP Congress 77". North-Holland Publishing Company, Amsterdam.

[A222] **Arvind and Gostelow, K. P. (1977).** "A Computer Capable of Exchanging Processors for Time". In [A221], pp. 849–853.

[A223] **Arvind, Culler, D. E., Iannucci, R. A., Kathail, V., Pingali, K. and Thomas, R. E. (1983).** "The Tagged Token Dataflow Architecture Parallel Machine". Lecture Material for subject 6.83s, Massachusetts Institute of Technology.

[A224] **Woods, J. V. (ed) (1986).** "Fifth Generation Computer Architectures". North-Holland, Amsterdam.

[A225] **Arvind and Culler, D. E. (1986).** "Managing Resources in a Parallel Machine". In [A224], pp. 103–121.

[A226] **Farrel, E. P., Ghani, N. and Treleaven, P. C. (1979).** "A Concurrent Computer Architecture and a Ring Based Implementation". In [A145], pp. 1–11.

[A227] **Watson, I. and Gurd, J. R. (1979).** "A Prototype Dataflow Computer With Token Labelling". *AFIPS NCC* **48**, pp. 623–628.

[A228] **Gurd, J. R. and Watson, I. (1980).** "Data Driven System for High Speed Computing — Part 1: Structuring Software for Parallel Execution". *Computer Design* **19(6)**, pp. 91–100.

[A229] **Gurd, J. R. and Watson, I. (1980).** "Data Driven System for High Speed Computing — Part 2: Hardware Design". *Computer Design* **19(7)**, pp. 97–106.

[A230] **da Silva, J. G. D. and Woods, J. V. (1981).** "Design of a Processing Subsystem for the Manchester Data-Flow Computer". *IEE Proceedings* **127E**, pp. 218–224.

[A231] **Watson, I. and Gurd, J. R. (1982).** "A Practical Data Flow Computer". *IEEE Computer* **15(2)**, pp. 51–57.

[A232] **Gurd, J. R., Kirkham, C. C. and Watson, I. (1985).** "The Manchester Prototype Dataflow Computer". *Communications of the ACM* **28**, pp. 34–52.

[A233] **Aleksander, I. (1978).** "Pattern Recognition with Networks of Memory Elements". In [A88], pp. 43–64.

[A234] **Aleksander, I., Thomas, W. V. and Bowden, P. A. (1984).** "WISARD — A Radical Step Forward in Image Recognition". *Sensor Review,* July 1984.

Appendix 3:

Test Programs

This appendix contains a number of MIMD-Pascal programs which were used, in conjunction with the system configuration files listed in appendix 5, to produce the results presented in the body of this thesis. The programs listed are:

Program B801B

Task – Region filling (8 regions in a 64 × 64 image).
Algorithm – Scan-line algorithm, with preferential vertical propogation whenever idle processing units are available.
Scheduler – Pre-waiting run-time scheduler, with private job lists and dynamic private list to global list job re-allocation whenever some processing units are idle.

Program B802B

Task – Region filling (8 regions in a 64 × 64 image).
Algorithm – Scan-line algorithm, with preferential vertical propogation whenever idle processing units are available.
Scheduler – Run-time scheduler with private job lists and dynamic private list to global list job re-allocation whenever some processing units are idle.

Program B803B

Task – Region filling (8 regions in a 64 × 64 image).
Algorithm – Scan-line algorithm, with preferential vertical propogation whenever idle processing units are available.
Scheduler – Pre-waiting run-time scheduler, with global job lists only.

Program B804B

Task – Region filling (8 regions in a 64 × 64 image).
Algorithm – Scan-line algorithm, with preferential vertical propogation whenever idle processing units are available.
Scheduler – Simple run-time scheduler, with global job lists only.

Program B805B

Task – Region filling (8 regions in a 64 × 64 image).
Algorithm – Wildfire algorithm.
Scheduler – Pre-waiting run-time scheduler, with private job lists and dynamic private list to global list job re-allocation whenever some processing units are idle.

Program B806B

Task – Region filling (8 regions in a 64 × 64 image).
Algorithm – Wildfire algorithm.
Scheduler – Run-time scheduler with private job lists and dynamic private list to global list job re-allocation whenever some processing units are idle.

Program B807B

Task – Region filling (8 regions in a 64 × 64 image).
Algorithm – Wildfire algorithm.
Scheduler – Pre-waiting run-time scheduler, with global job lists only.

Program B808B

Task – Region filling (8 regions in a 64 × 64 image).
Algorithm – Wildfire algorithm.

Scheduler – Simple run-time scheduler, with global job lists only.

Program Q801B

Task – Integer sorting (256 random numbers).
Algorithm – Quicksort algorithm.
Scheduler – Pre-waiting run-time scheduler, with private job lists and dynamic private list to global list job re-allocation whenever some processing units are idle.

Program Q802B

Task – Integer sorting (256 random numbers).
Algorithm – Quicksort algorithm.
Scheduler – Run-time scheduler with private job lists and dynamic private list to global list job re-allocation whenever some processing units are idle.

Program Q803B

Task – Integer sorting (256 random numbers).
Algorithm – Quicksort algorithm.
Scheduler – Pre-waiting run-time scheduler, with global job lists only.

Program Q804B

Task – Integer sorting (256 random numbers).
Algorithm – Quicksort algorithm.
Scheduler – Simple run-time scheduler, with global job lists only.

Program G16P

Task – Linear filtering (3×3 mask on a 16×16 image).
Algorithm – Spatial-domain convolution with a Gaussian mask.
Scheduler – None.

A3.1 Region Filler B801B

```
($ COMPRESSCODE + $)
($ SUPERCOMPRESS + $)
($ CONFIGURATION CON8P $)

PROGRAM B801B;

(* Blob filling task - 8 blobs, 64 x 64 pixel image *)

(* Scan line algorithm *)
(* Private and global lists *)
(* Wait interference reduced by pre-checking *)
(* Vertical propogation if idle units available *)
(* Private to global job transfer if idle units available *)

GLOBAL TASK;

VAR     IMAGE           :ARRAY[0..4095] OF INTEGER;

        (* Global job list *)
        GLISTX,GLISTY   :ARRAY[0..63] OF INTEGER;
        TOPOFGLIST      :INTEGER;

        (* Global job list semaphore *)
        LISTACCESS      :INTEGER;

        (* Count of processors with nothing to do *)
        IDLEUNITS       :INTEGER;

        (* Count of active jobs *)
        JOBSINSYSTEM    :INTEGER;


PROCEDURE GIGNITE(X,Y:INTEGER;);

(* Create a job on the global job list *)

  BEGIN
  INCREMENT(JOBSINSYSTEM);
  WAIT(LISTACCESS);
  INCREMENT(TOPOFGLIST);
  GLISTX[TOPOFGLIST]:=X;
  GLISTY[TOPOFGLIST]:=Y;
  SIGNAL(LISTACCESS);
  END;


PROCEDURE PROCESS(N:INTEGER;);

(* Perform main process *)

VAR (* Private job list *)
    PLISTX,PLISTY   :ARRAY[0..63] OF INTEGER;
    TOPOFPLIST      :INTEGER;

    (* Misc.  *)
    COUNTER         :INTEGER;
    PX,PY           :INTEGER;

  PROCEDURE PIGNITE(X,Y:INTEGER;);

  (* Create a job on a private job list *)

    BEGIN
    INCREMENT(JOBSINSYSTEM);
    INCREMENT(TOPOFPLIST);
    PLISTX[TOPOFPLIST]:=X;
    PLISTY[TOPOFPLIST]:=Y;
    END;
```

```
PROCEDURE SCANLINESPREAD(X,Y:INTEGER;);

(* Delete and propogate from a point *)
(* using a scan line algorithm *)

VAR XX,LEFT,RIGHT :INTEGER;
    THISISZERO,
    LASTWASZERO    :BOOLEAN;


  PROCEDURE VPROPOGATE(VPX,VPY:INTEGER;);

  (* Propogate (globally) vertically  *)

      BEGIN
      IF VPY<63 THEN
          IF IMAGE[VPX+64*(VPY+1)]<>0 THEN
              GIGNITE(VPX,VPY+1);

      IF VPY>0 THEN
          IF IMAGE[VPX+64*(VPY-1)]<>0 THEN
              GIGNITE(VPX,VPY-1);
      END;

  BEGIN
  (* 63>=X>=0,  63>=Y>=0 *)

  IF IMAGE[X+64*Y]<>0 THEN
      BEGIN
      (* Delete the point *)
      IMAGE[X+64*Y]:=0;

      ($ ENTER Vpropogate $)
      (* Immediately propogate (globally) *)
      (* vertically if other units are idle *)
      IF IDLEUNITS>TOPOFGLIST+1 THEN
          VPROPOGATE(X,Y);
      ($ EXIT Vpropogate $)

      ($ ENTER Hfill $)
      (* Delete to left extremity *)
      LEFT:=X;
      IF NOT( (LEFT<=0) OR (IMAGE[LEFT-1+64*Y]=0) ) THEN
          BEGIN
          REPEAT
              LEFT:=LEFT-1;
              IMAGE[LEFT+64*Y]:=0;
          UNTIL (LEFT<=0) OR (IMAGE[LEFT-1+64*Y]=0);
          END;

      (* Delete to right extremity *)
      RIGHT:=X;
      IF NOT( (RIGHT>=63) OR (IMAGE[RIGHT+1+64*Y]=0) ) THEN
          BEGIN
          REPEAT
              RIGHT:=RIGHT+1;
              IMAGE[RIGHT+64*Y]:=0;
          UNTIL (RIGHT>=63) OR (IMAGE[RIGHT+1+64*Y]=0);
          END;
      ($ EXIT Hfill $)

      ($ ENTER Hscan $)
      (* Check line above for required new seed points *)
      IF Y<63 THEN
          BEGIN
          LASTWASZERO:=TRUE;
          FOR XX:=LEFT TO RIGHT DO
              BEGIN
              THISISZERO:=(IMAGE[XX+64*(Y+1)]=0);
              IF (NOT THISISZERO) AND LASTWASZERO THEN
                  PIGNITE(XX,Y+1);
              LASTWASZERO:=THISISZERO;
              END;
          END;
```

```
      (* Check line below for required new seed points *)
      IF Y>0 THEN
          BEGIN
          LASTWASZERO:=TRUE;
          FOR XX:=LEFT TO RIGHT DO
              BEGIN
              THISISZERO:=(IMAGE[XX+64*(Y-1)]=0);
              IF (NOT THISISZERO) AND LASTWASZERO THEN
                  PIGNITE(XX,Y-1);
              LASTWASZERO:=THISISZERO;
              END;
          END;
      ($ EXIT Hscan $)
      END;

  (* One job now done *)
  DECREMENT(JOBSINSYSTEM);
  END;


BEGIN
TOPOFPLIST:=-1;
INCREMENT(IDLEUNITS);
REPEAT

  ($ ENTER Prewait $)
  REPEAT
  (* Wait until there is a good chance of a successful *)
  (* WAIT, and of there being a job on the global job *)
  (* list *)
  UNTIL ((TOPOFGLIST>=0) AND (LISTACCESS>0))
            OR (JOBSINSYSTEM=0);
  ($ EXIT Prewait $)

  ($ ENTER Wait1 $)
  WAIT(LISTACCESS);
  ($ EXIT Wait1 $)

  IF TOPOFGLIST>=0 THEN
    BEGIN
    (* This processor is now busy *)
    DECREMENT(IDLEUNITS);

    ($ ENTER Getglbl $)
    (* Move Job from Global list to private list *)
    (* Decrement first, so that global list doesnt *)
    (* look fuller than it is, to other processors. *)
    DECREMENT(TOPOFGLIST);
    INCREMENT(TOPOFPLIST);
    PLISTX[TOPOFPLIST]:=GLISTX[TOPOFGLIST+1];
    PLISTY[TOPOFPLIST]:=GLISTY[TOPOFGLIST+1];
    SIGNAL(LISTACCESS);
    ($ EXIT Getglbl $)

    ($ ENTER Private $)
    REPEAT
      (* Get a job from the private list, and do it *)
      PX:=PLISTX[TOPOFPLIST];
      PY:=PLISTY[TOPOFPLIST];
      DECREMENT(TOPOFPLIST);
      SCANLINESPREAD(PX,PY);
```

```
        ($ ENTER Sendglbl $)

        (* If there are more processes idle than there are *)
        (* jobs for them on the global job list, AND *)
        (* this private list is not empty, transfer *)
        (* half of the private list to the global list *)

        IF IDLEUNITS>TOPOFGLIST+1 THEN
          IF TOPOFPLIST>=1 THEN
            BEGIN

            ($ ENTER Wait2 $)
            WAIT(LISTACCESS);
            ($ EXIT Wait2 $)

            FOR COUNTER:=0 TO (TOPOFPLIST-1)/2 DO
              BEGIN
              INCREMENT(TOPOFGLIST);
              GLISTX[TOPOFGLIST]:=PLISTX[TOPOFPLIST];
              GLISTY[TOPOFGLIST]:=PLISTY[TOPOFPLIST];
              DECREMENT(TOPOFPLIST);
              END;
            SIGNAL(LISTACCESS);
            END;
      ($ EXIT Sendglbl $)

      UNTIL TOPOFPLIST=-1;
      ($ EXIT Private $)

      (* Nothing to do now, until a job can be *)
      (* found on the global job list *)
      INCREMENT(IDLEUNITS);
      END
    ELSE
      BEGIN
      ($ ENTER Oops $)
      (* Nothing to do, return to Prewait *)
      SIGNAL(LISTACCESS);
      ($ EXIT Oops $)
      END;
  UNTIL JOBSINSYSTEM=0;
  END;


BEGIN
(* Initialisation *)
TOPOFGLIST:=-1;
LISTACCESS:=1;

IDLEUNITS:=0;
JOBSINSYSTEM:=0;

(* Initialise Job list with known positions of blobs *)
 GIGNITE( 8, 8);GIGNITE(24, 8);GIGNITE(40, 8);GIGNITE(56, 8);
 GIGNITE( 8,24);GIGNITE(24,24);GIGNITE(40,24);GIGNITE(56,24);
{GIGNITE( 8,40);GIGNITE(24,40);GIGNITE(40,40);GIGNITE(56,40);}
{GIGNITE( 8,56);GIGNITE(24,56);GIGNITE(40,56);GIGNITE(56,56);}

END;


GENERAL TASK;

  BEGIN
  END;

END.
```

A3.2 Region Filler B802B

```
($ COMPRESSCODE + $)
($ SUPERCOMPRESS + $)
($ CONFIGURATION CON8P $)

PROGRAM B802B;

(* Blob filling task - 8 blobs, 64 x 64 pixel image *)

(* Scan line algorithm *)
(* Private and global lists *)
(* Vertical propogation if idle units available *)
(* Private to global job transfer if idle units available *)

GLOBAL TASK;

VAR     IMAGE           :ARRAY[0..4095] OF INTEGER;

        (* Global job list *)
        GLISTX,GLISTY   :ARRAY[0..63] OF INTEGER;
        TOPOFGLIST      :INTEGER;

        (* Global job list semaphore *)
        LISTACCESS      :INTEGER;

        (* Count of processors with nothing to do *)
        IDLEUNITS       :INTEGER;

        (* Count of active jobs *)
        JOBSINSYSTEM    :INTEGER;


PROCEDURE GIGNITE(X,Y:INTEGER;);

(* Create a job on the global job list *)

  BEGIN
  INCREMENT(JOBSINSYSTEM);
  WAIT(LISTACCESS);
  INCREMENT(TOPOFGLIST);
  GLISTX[TOPOFGLIST]:=X;
  GLISTY[TOPOFGLIST]:=Y;
  SIGNAL(LISTACCESS);
  END;


PROCEDURE PROCESS(N:INTEGER;);

(* Perform main process *)

VAR (* Private job list *)
    PLISTX,PLISTY   :ARRAY[0..63] OF INTEGER;
    TOPOFPLIST      :INTEGER;

    (* Misc.  *)
    COUNTER         :INTEGER;
    PX,PY           :INTEGER;

  PROCEDURE PIGNITE(X,Y:INTEGER;);

  (* Create a job on a private job list *)

    BEGIN
    INCREMENT(JOBSINSYSTEM);
    INCREMENT(TOPOFPLIST);
    PLISTX[TOPOFPLIST]:=X;
    PLISTY[TOPOFPLIST]:=Y;
    END;
```

```
PROCEDURE SCANLINESPREAD(X,Y:INTEGER;);

(* Delete and propogate from a point *)
(* using a scan line algorithm *)

VAR XX,LEFT,RIGHT :INTEGER;
    THISISZERO,
    LASTWASZERO    :BOOLEAN;


  PROCEDURE VPROPOGATE(VPX,VPY:INTEGER;);

  (* Propogate (globally) vertically  *)

      BEGIN
      IF VPY<63 THEN
          IF IMAGE[VPX+64*(VPY+1)]<>0 THEN
              GIGNITE(VPX,VPY+1);

      IF VPY>0 THEN
          IF IMAGE[VPX+64*(VPY-1)]<>0 THEN
              GIGNITE(VPX,VPY-1);
      END;

  BEGIN
  (* 63>=X>=0,  63>=Y>=0 *)

  IF IMAGE[X+64*Y]<>0 THEN
      BEGIN
      (* Delete the point *)
      IMAGE[X+64*Y]:=0;

      ($ ENTER Vpropogate $)
      (* Immediately propogate (globally) *)
      (* vertically if other units are idle *)
      IF IDLEUNITS>TOPOFGLIST+1 THEN
            VPROPOGATE(X,Y);
      ($ EXIT Vpropogate $)

      ($ ENTER Hfill $)
      (* Delete to left extremity *)
      LEFT:=X;
      IF NOT( (LEFT<=0) OR (IMAGE[LEFT-1+64*Y]=0) ) THEN
          BEGIN
          REPEAT
              LEFT:=LEFT-1;
              IMAGE[LEFT+64*Y]:=0;
          UNTIL (LEFT<=0) OR (IMAGE[LEFT-1+64*Y]=0);
          END;

      (* Delete to right extremity *)
      RIGHT:=X;
      IF NOT( (RIGHT>=63) OR (IMAGE[RIGHT+1+64*Y]=0) ) THEN
          BEGIN
          REPEAT
              RIGHT:=RIGHT+1;
              IMAGE[RIGHT+64*Y]:=0;
          UNTIL (RIGHT>=63) OR (IMAGE[RIGHT+1+64*Y]=0);
          END;
      ($ EXIT Hfill $)

      ($ ENTER Hscan $)
      (* Check line above for required new seed points *)
      IF Y<63 THEN
          BEGIN
          LASTWASZERO:=TRUE;
          FOR XX:=LEFT TO RIGHT DO
              BEGIN
              THISISZERO:=(IMAGE[XX+64*(Y+1)]=0);
              IF (NOT THISISZERO) AND LASTWASZERO THEN
                  PIGNITE(XX,Y+1);
              LASTWASZERO:=THISISZERO;
              END;
          END;
```

```
      (* Check line below for required new seed points *)
      IF Y>0 THEN
          BEGIN
          LASTWASZERO:=TRUE;
          FOR XX:=LEFT TO RIGHT DO
              BEGIN
              THISISZERO:=(IMAGE[XX+64*(Y-1)]=0);
              IF (NOT THISISZERO) AND LASTWASZERO THEN
                  PIGNITE(XX,Y-1);
              LASTWASZERO:=THISISZERO;
              END;
          END;
      ($ EXIT Hscan $)
      END;

  (* One job now done *)
  DECREMENT(JOBSINSYSTEM);
  END;



BEGIN
TOPOFPLIST:=-1;
INCREMENT(IDLEUNITS);
REPEAT

  ($ ENTER Wait1 $)
  WAIT(LISTACCESS);
  ($ EXIT Wait1 $)

  IF TOPOFGLIST>=0 THEN
    BEGIN
    (* This processor is now busy *)
    DECREMENT(IDLEUNITS);

    ($ ENTER Getglbl $)
    (* Move Job from Global list to private list *)
    (* Decrement first, so that global list doesnt *)
    (* look fuller than it is, to other processors. *)
    DECREMENT(TOPOFGLIST);
    INCREMENT(TOPOFPLIST);
    PLISTX[TOPOFPLIST]:=GLISTX[TOPOFGLIST+1];
    PLISTY[TOPOFPLIST]:=GLISTY[TOPOFGLIST+1];
    SIGNAL(LISTACCESS);
    ($ EXIT Getglbl $)

    ($ ENTER Private $)
    REPEAT
      (* Get a job from the private list, and do it *)
      PX:=PLISTX[TOPOFPLIST];
      PY:=PLISTY[TOPOFPLIST];
      DECREMENT(TOPOFPLIST);
      SCANLINESPREAD(PX,PY);

      ($ ENTER Sendglbl $)

      (* If there are more processes idle than there are *)
      (* jobs for them on the global job list, AND *)
      (* this private list is not empty, transfer *)
      (* half of the private list to the global list *)

      IF IDLEUNITS>TOPOFGLIST+1 THEN
        IF TOPOFPLIST>=1 THEN
          BEGIN

          ($ ENTER Wait2 $)
          WAIT(LISTACCESS);
          ($ EXIT Wait2 $)
```

```
            FOR COUNTER:=0 TO (TOPOFPLIST-1)/2 DO
              BEGIN
              INCREMENT(TOPOFGLIST);
              GLISTX[TOPOFGLIST]:=PLISTX[TOPOFPLIST];
              GLISTY[TOPOFGLIST]:=PLISTY[TOPOFPLIST];
              DECREMENT(TOPOFPLIST);
              END;
            SIGNAL(LISTACCESS);
            END;
      ($ EXIT Sendglbl $)

      UNTIL TOPOFPLIST=-1;
      ($ EXIT Private $)

      (* Nothing to do now, until a job can be *)
      (* found on the global job list *)
      INCREMENT(IDLEUNITS);
      END
    ELSE
      BEGIN
      ($ ENTER Oops $)
      (* Nothing to do *)
      SIGNAL(LISTACCESS);
      ($ EXIT Oops $)
      END;
  UNTIL JOBSINSYSTEM=0;
  END;


BEGIN
(* Initialisation *)
TOPOFGLIST:=-1;
LISTACCESS:=1;

IDLEUNITS:=0;
JOBSINSYSTEM:=0;

(* Initialise Job list with known positions of blobs *)
 GIGNITE( 8, 8);GIGNITE(24, 8);GIGNITE(40, 8);GIGNITE(56, 8);
 GIGNITE( 8,24);GIGNITE(24,24);GIGNITE(40,24);GIGNITE(56,24);
{GIGNITE( 8,40);GIGNITE(24,40);GIGNITE(40,40);GIGNITE(56,40);}
{GIGNITE( 8,56);GIGNITE(24,56);GIGNITE(40,56);GIGNITE(56,56);}

END;


GENERAL TASK;

  BEGIN
  END;

END.
```

A3.3 Region Filler B803B

```
($ COMPRESSCODE + $)
($ SUPERCOMPRESS + $)
($ CONFIGURATION CONSP $)

PROGRAM B803B;

(* Blob filling task - 8 blobs, 64 x 64 pixel image *)

(* Scan line algorithm *)
(* Global lists only *)
(* Wait interference reduced by pre-checking *)
(* Vertical propogation if idle units available *)

GLOBAL TASK;

VAR     IMAGE           :ARRAY[0..4095] OF INTEGER;

        (* Global job list *)
        GLISTX,GLISTY   :ARRAY[0..63] OF INTEGER;
        TOPOFGLIST      :INTEGER;

        (* Global job list semaphore *)
        LISTACCESS      :INTEGER;

        (* Count of processors with nothing to do *)
        IDLEUNITS       :INTEGER;

        (* Count of active jobs *)
        JOBSINSYSTEM    :INTEGER;


PROCEDURE GIGNITE(X,Y:INTEGER;);

(* Create a job on the global job list *)

  BEGIN
  INCREMENT(JOBSINSYSTEM);
  WAIT(LISTACCESS);
  INCREMENT(TOPOFGLIST);
  GLISTX[TOPOFGLIST]:=X;
  GLISTY[TOPOFGLIST]:=Y;
  SIGNAL(LISTACCESS);
  END;


PROCEDURE PROCESS(N:INTEGER;);

(* Perform main process *)

VAR (* Misc.  *)
    COUNTER         :INTEGER;
    PX,PY           :INTEGER;

  PROCEDURE SCANLINESPREAD(X,Y:INTEGER;);

  (* Delete and propogate from a point *)
  (* using a scan line algorithm *)

  VAR XX,LEFT,RIGHT :INTEGER;
      THISISZERO,
      LASTWASZERO    :BOOLEAN;
```

```
PROCEDURE VPROPOGATE(VPX,VPY:INTEGER;);

(* Propogate (globally) vertically  *)

    BEGIN
    IF VPY<63 THEN
        IF IMAGE[VPX+64*(VPY+1)]<>0 THEN
            GIGNITE(VPX,VPY+1);

    IF VPY>0 THEN
        IF IMAGE[VPX+64*(VPY-1)]<>0 THEN
            GIGNITE(VPX,VPY-1);
    END;

BEGIN
(* 63>=X>=0,  63>=Y>=0 *)

IF IMAGE[X+64*Y]<>0 THEN
    BEGIN
    (* Delete the point *)
    IMAGE[X+64*Y]:=0;

    ($ ENTER Vpropogate $)
    (* Immediately propogate (globally) *)
    (* vertically if other units are idle *)
    IF IDLEUNITS>TOPOFGLIST+1 THEN
          VPROPOGATE(X,Y);
    ($ EXIT Vpropogate $)

    ($ ENTER Hfill $)
    (* Delete to left extremity *)
    LEFT:=X;
    IF NOT( (LEFT<=0) OR (IMAGE[LEFT-1+64*Y]=0) ) THEN
        BEGIN
        REPEAT
            LEFT:=LEFT-1;
            IMAGE[LEFT+64*Y]:=0;
        UNTIL (LEFT<=0) OR (IMAGE[LEFT-1+64*Y]=0);
        END;

    (* Delete to right extremity *)
    RIGHT:=X;
    IF NOT( (RIGHT>=63) OR (IMAGE[RIGHT+1+64*Y]=0) ) THEN
        BEGIN
        REPEAT
            RIGHT:=RIGHT+1;
            IMAGE[RIGHT+64*Y]:=0;
        UNTIL (RIGHT>=63) OR (IMAGE[RIGHT+1+64*Y]=0);
        END;
    ($ EXIT Hfill $)

    ($ ENTER Hscan $)
    (* Check line above for required new seed points *)
    IF Y<63 THEN
        BEGIN
        LASTWASZERO:=TRUE;
        FOR XX:=LEFT TO RIGHT DO
            BEGIN
            THISISZERO:=(IMAGE[XX+64*(Y+1)]=0);
            IF (NOT THISISZERO) AND LASTWASZERO THEN
                GIGNITE(XX,Y+1);
            LASTWASZERO:=THISISZERO;
            END;
        END;
```

```
        (* Check line below for required new seed points *)
        IF Y>0 THEN
            BEGIN
            LASTWASZERO:=TRUE;
            FOR XX:=LEFT TO RIGHT DO
                BEGIN
                THISISZERO:=(IMAGE[XX+64*(Y-1)]=0);
                IF (NOT THISISZERO) AND LASTWASZERO THEN
                    GIGNITE(XX,Y-1);
                LASTWASZERO:=THISISZERO;
                END;
            END;
        ($ EXIT Hscan $)
        END;

    (* One job now done *)
    DECREMENT(JOBSINSYSTEM);
    END;


  BEGIN
  INCREMENT(IDLEUNITS);
  REPEAT

    ($ ENTER Prewait $)
    REPEAT
    (* Wait until there is a good chance of a successful *)
    (* WAIT, and of there being a job on the global job *)
    (* list *)
    UNTIL ((TOPOFGLIST>=0) AND (LISTACCESS>0))
                OR (JOBSINSYSTEM=0);
    ($ EXIT Prewait $)

    ($ ENTER Wait1 $)
    WAIT(LISTACCESS);
    ($ EXIT Wait1 $)

    IF TOPOFGLIST>=0 THEN
      BEGIN
      (* This processor is now busy *)
      DECREMENT(IDLEUNITS);

      (* Get a job from the global list, and do it *)

      ($ ENTER Getglbl $)
      PX:=GLISTX[TOPOFGLIST];
      PY:=GLISTY[TOPOFGLIST];
      DECREMENT(TOPOFGLIST);
      SIGNAL(LISTACCESS);
      ($ EXIT Getglbl $)

      SCANLINESPREAD(PX,PY);

      (* Nothing to do now, until a job can be *)
      (* found on the global job list *)
      INCREMENT(IDLEUNITS);
      END
    ELSE
      BEGIN
      ($ ENTER Oops $)
      (* Nothing to do, return to Prewait *)
      SIGNAL(LISTACCESS);
      ($ EXIT Oops $)
      END;
  UNTIL JOBSINSYSTEM=0;
  END;
```

```
BEGIN
(* Initialisation *)
TOPOFGLIST:=-1;
LISTACCESS:=1;

IDLEUNITS:=0;
JOBSINSYSTEM:=0;

(* Initialise Job list with known positions of blobs *)
 GIGNITE( 8, 8);GIGNITE(24, 8);GIGNITE(40, 8);GIGNITE(56, 8);
 GIGNITE( 8,24);GIGNITE(24,24);GIGNITE(40,24);GIGNITE(56,24);
{GIGNITE( 8,40);GIGNITE(24,40);GIGNITE(40,40);GIGNITE(56,40);}
{GIGNITE( 8,56);GIGNITE(24,56);GIGNITE(40,56);GIGNITE(56,56);}

END;


GENERAL TASK;

  BEGIN
  END;

END.
```

A3.4 Region Filler B804B

```
($ COMPRESSCODE + $)
($ SUPERCOMPRESS + $)
($ CONFIGURATION CONSP $)

PROGRAM B804B;

(* Blob filling task - 8 blobs, 64 x 64 pixel image *)

(* Scan line algorithm *)
(* Global lists only *)
(* Vertical propogation if idle units available *)

GLOBAL TASK;

VAR     IMAGE           :ARRAY[0..4095] OF INTEGER;

        (* Global job list *)
        GLISTX,GLISTY   :ARRAY[0..63] OF INTEGER;
        TOPOFGLIST      :INTEGER;

        (* Global job list semaphore *)
        LISTACCESS      :INTEGER;

        (* Count of processors with nothing to do *)
        IDLEUNITS       :INTEGER;

        (* Count of active jobs *)
        JOBSINSYSTEM    :INTEGER;


PROCEDURE GIGNITE(X,Y:INTEGER;);

(* Create a job on the global job list *)

  BEGIN
  INCREMENT(JOBSINSYSTEM);
  WAIT(LISTACCESS);
  INCREMENT(TOPOFGLIST);
  GLISTX[TOPOFGLIST]:=X;
  GLISTY[TOPOFGLIST]:=Y;
  SIGNAL(LISTACCESS);
  END;

PROCEDURE PROCESS(N:INTEGER;);

(* Perform main process *)

VAR (* Misc.  *)
    COUNTER         :INTEGER;
    PX,PY           :INTEGER;

  PROCEDURE SCANLINESPREAD(X,Y:INTEGER;);

  (* Delete and propogate from a point *)
  (* using a scan line algorithm *)

  VAR XX,LEFT,RIGHT :INTEGER;
      THISISZERO,
      LASTWASZERO    :BOOLEAN;


    PROCEDURE VPROPOGATE(VPX,VPY:INTEGER;);

    (* Propogate (globally) vertically  *)

        BEGIN
        IF VPY<63 THEN
            IF IMAGE[VPX+64*(VPY+1)]<>0 THEN
                GIGNITE(VPX,VPY+1);

        IF VPY>0 THEN
            IF IMAGE[VPX+64*(VPY-1)]<>0 THEN
                GIGNITE(VPX,VPY-1);
        END;
```

```
BEGIN
(* 63>=X>=0,  63>=Y>=0 *)

IF IMAGE[X+64*Y]<>0 THEN
    BEGIN
    (* Delete the point *)
    IMAGE[X+64*Y]:=0;

    ($ ENTER Vpropogate $)
    (* Immediately propogate (globally) *)
    (* vertically if other units are idle *)
    IF IDLEUNITS>TOPOFGLIST+1 THEN
        VPROPOGATE(X,Y);
    ($ EXIT Vpropogate $)

    ($ ENTER Hfill $)
    (* Delete to left extremity *)
    LEFT:=X;
    IF NOT( (LEFT<=0) OR (IMAGE[LEFT-1+64*Y]=0) ) THEN
        BEGIN
        REPEAT
            LEFT:=LEFT-1;
            IMAGE[LEFT+64*Y]:=0;
        UNTIL (LEFT<=0) OR (IMAGE[LEFT-1+64*Y]=0);
        END;

    (* Delete to right extremity *)
    RIGHT:=X;
    IF NOT( (RIGHT>=63) OR (IMAGE[RIGHT+1+64*Y]=0) ) THEN
        BEGIN
        REPEAT
            RIGHT:=RIGHT+1;
            IMAGE[RIGHT+64*Y]:=0;
        UNTIL (RIGHT>=63) OR (IMAGE[RIGHT+1+64*Y]=0);
        END;
    ($ EXIT Hfill $)

    ($ ENTER Hscan $)
    (* Check line above for required new seed points *)
    IF Y<63 THEN
        BEGIN
        LASTWASZERO:=TRUE;
        FOR XX:=LEFT TO RIGHT DO
            BEGIN
            THISISZERO:=(IMAGE[XX+64*(Y+1)]=0);
            IF (NOT THISISZERO) AND LASTWASZERO THEN
                GIGNITE(XX,Y+1);
            LASTWASZERO:=THISISZERO;
            END;
        END;

    (* Check line below for required new seed points *)
    IF Y>0 THEN
        BEGIN
        LASTWASZERO:=TRUE;
        FOR XX:=LEFT TO RIGHT DO
            BEGIN
            THISISZERO:=(IMAGE[XX+64*(Y-1)]=0);
            IF (NOT THISISZERO) AND LASTWASZERO THEN
                GIGNITE(XX,Y-1);
            LASTWASZERO:=THISISZERO;
            END;
        END;
    ($ EXIT Hscan $)
    END;

(* One job now done *)
DECREMENT(JOBSINSYSTEM);
END;
```

```
  BEGIN
  INCREMENT(IDLEUNITS);
  REPEAT

    ($ ENTER Wait1 $)
    WAIT(LISTACCESS);
    ($ EXIT Wait1 $)

    IF TOPOFGLIST>=0 THEN
      BEGIN
      (* This processor is now busy *)
      DECREMENT(IDLEUNITS);

      (* Get a job from the global list, and do it *)

      ($ ENTER Getglbl $)
      PX:=GLISTX[TOPOFGLIST];
      PY:=GLISTY[TOPOFGLIST];
      DECREMENT(TOPOFGLIST);
      SIGNAL(LISTACCESS);
      ($ EXIT Getglbl $)

      SCANLINESPREAD(PX,PY);

      (* Nothing to do now, until a job can be *)
      (* found on the global job list *)
      INCREMENT(IDLEUNITS);
      END
    ELSE
      BEGIN
      ($ ENTER Oops $)
      (* Nothing to do *)
      SIGNAL(LISTACCESS);
      ($ EXIT Oops $)
      END;
  UNTIL JOBSINSYSTEM=0;
  END;


BEGIN
(* Initialisation *)
TOPOFGLIST:=-1;
LISTACCESS:=1;

IDLEUNITS:=0;
JOBSINSYSTEM:=0;

(* Initialise Job list with known positions of blobs *)
 GIGNITE( 8, 8);GIGNITE(24, 8);GIGNITE(40, 8);GIGNITE(56, 8);
 GIGNITE( 8,24);GIGNITE(24,24);GIGNITE(40,24);GIGNITE(56,24);
{GIGNITE( 8,40);GIGNITE(24,40);GIGNITE(40,40);GIGNITE(56,40);}
{GIGNITE( 8,56);GIGNITE(24,56);GIGNITE(40,56);GIGNITE(56,56);}

END;


GENERAL TASK;

  BEGIN
  END;

END.
```

A3.5 Region Filler B805B

```
($ COMPRESSCODE + $)
($ SUPERCOMPRESS + $)
($ CONFIGURATION CON8P $)

PROGRAM B805B;

(* Blob filling task - 8 blobs, 64 x 64 pixel image *)

(* Wildfire algorithm *)
(* Private and global lists *)
(* Wait interference reduced by pre-checking *)
(* Private to global job transfer if idle units available *)

GLOBAL TASK;

VAR     IMAGE           :ARRAY[0..4095] OF INTEGER;

        (* Global job list *)
        GLISTX,GLISTY   :ARRAY[0..2047] OF INTEGER;
        TOPOFGLIST      :INTEGER;

        (* Global job list semaphore *)
        LISTACCESS      :INTEGER;

        (* Count of processors with nothing to do *)
        IDLEUNITS       :INTEGER;

        (* Count of active jobs *)
        JOBSINSYSTEM    :INTEGER;


PROCEDURE GIGNITE(X,Y:INTEGER;);

(* Create a job on the global job list *)

  BEGIN
  INCREMENT(JOBSINSYSTEM);
  WAIT(LISTACCESS);
  INCREMENT(TOPOFGLIST);
  GLISTX[TOPOFGLIST]:=X;
  GLISTY[TOPOFGLIST]:=Y;
  SIGNAL(LISTACCESS);
  END;


PROCEDURE PROCESS(N:INTEGER;);

(* Perform main process *)

VAR (* Private job list *)
    PLISTX,PLISTY   :ARRAY[0..255] OF INTEGER;
    TOPOFPLIST      :INTEGER;

    (* Misc.  *)
    COUNTER         :INTEGER;
    PX,PY           :INTEGER;


  PROCEDURE PIGNITE(X,Y:INTEGER;);

  (* Create a job on a private job list *)

    BEGIN
    INCREMENT(JOBSINSYSTEM);
    INCREMENT(TOPOFPLIST);
    PLISTX[TOPOFPLIST]:=X;
    PLISTY[TOPOFPLIST]:=Y;
    END;
```

```
PROCEDURE SPREAD(X,Y:INTEGER;);

(* Spread vertically and horizontally.                *)
(* Each point is checked before it is PIGNITEd, since *)
(* this will have to be done eventually, and checking *)
(* first reduces job list sizes and overheads.        *)

  BEGIN
  (* 63>=X>=0,  63>=Y>=0 *)
  IF IMAGE[X+64*Y]=1 THEN
      BEGIN
      IMAGE[X+64*Y]:=0;
      IF (X>0) THEN IF (IMAGE[X-1+64*Y]=1) THEN
          PIGNITE(X-1,Y);
      IF (X<63) THEN IF (IMAGE[X+1+64*Y]=1) THEN
          PIGNITE(X+1,Y);
      IF (Y>0) THEN IF (IMAGE[X+64*(Y-1)]=1) THEN
          PIGNITE(X,Y-1);
      IF (Y<63) THEN IF (IMAGE[X+64*(Y+1)]=1) THEN
          PIGNITE(X,Y+1);
      END;
  DECREMENT(JOBSINSYSTEM);
  END;


BEGIN
TOPOFPLIST:=-1;
INCREMENT(IDLEUNITS);
REPEAT

  ($ ENTER Prewait $)
  REPEAT
  (* Wait until there is a good chance of a successful *)
  (* WAIT, and of there being a job on the global job *)
  (* list *)
  UNTIL ((TOPOFGLIST>=0) AND (LISTACCESS>0))
              OR (JOBSINSYSTEM=0);
  ($ EXIT Prewait $)

  ($ ENTER Wait1 $)
  WAIT(LISTACCESS);
  ($ EXIT Wait1 $)

  IF TOPOFGLIST>=0 THEN
    BEGIN
    (* This processor is now busy *)
    DECREMENT(IDLEUNITS);

    ($ ENTER Getglbl $)
    (* Move Job from Global list to private list *)
    (* Decrement first, so that global list doesnt *)
    (* look fuller than it is, to other processors. *)
    DECREMENT(TOPOFGLIST);
    INCREMENT(TOPOFPLIST);
    PLISTX[TOPOFPLIST]:=GLISTX[TOPOFGLIST+1];
    PLISTY[TOPOFPLIST]:=GLISTY[TOPOFGLIST+1];
    SIGNAL(LISTACCESS);
    ($ EXIT Getglbl $)

    ($ ENTER Private $)
    REPEAT
      (* Get a job from the private list, and do it *)
      PX:=PLISTX[TOPOFPLIST];
      PY:=PLISTY[TOPOFPLIST];
      DECREMENT(TOPOFPLIST);
      SPREAD(PX,PY);
```

```
        ($ ENTER Sendglbl $)

        (* If there are more processes idle than there are *)
        (* jobs for them on the global job list, AND *)
        (* this private list is not empty, transfer *)
        (* half of the private list to the global list *)

        IF IDLEUNITS>TOPOFGLIST+1 THEN
          IF TOPOFPLIST>=1 THEN
            BEGIN

            ($ ENTER Wait2 $)
            WAIT(LISTACCESS);
            ($ EXIT Wait2 $)

            FOR COUNTER:=0 TO (TOPOFPLIST-1)/2 DO
              BEGIN
              INCREMENT(TOPOFGLIST);
              GLISTX[TOPOFGLIST]:=PLISTX[TOPOFPLIST];
              GLISTY[TOPOFGLIST]:=PLISTY[TOPOFPLIST];
              DECREMENT(TOPOFPLIST);
              END;
            SIGNAL(LISTACCESS);
            END;
      ($ EXIT Sendglbl $)

      UNTIL TOPOFPLIST=-1;
      ($ EXIT Private $)

      (* Nothing to do now, until a job can be *)
      (* found on the global job list *)
      INCREMENT(IDLEUNITS);
      END
    ELSE
      BEGIN
      ($ ENTER Oops $)
      (* Nothing to do, return to Prewait *)
      SIGNAL(LISTACCESS);
      ($ EXIT Oops $)
      END;
  UNTIL JOBSINSYSTEM=0;
  END;


BEGIN
(* Initialisation *)
TOPOFGLIST:=-1;
LISTACCESS:=1;

IDLEUNITS:=0;
JOBSINSYSTEM:=0;

(* Initialise Job list with known positions of blobs *)
 GIGNITE( 8, 8);GIGNITE(24, 8);GIGNITE(40, 8);GIGNITE(56, 8);
 GIGNITE( 8,24);GIGNITE(24,24);GIGNITE(40,24);GIGNITE(56,24);
{GIGNITE( 8,40);GIGNITE(24,40);GIGNITE(40,40);GIGNITE(56,40);}
{GIGNITE( 8,56);GIGNITE(24,56);GIGNITE(40,56);GIGNITE(56,56);}

END;


GENERAL TASK;

  BEGIN
  END;

END.
```

A3.6 Region Filler B806B

```
($ COMPRESSCODE + $)
($ SUPERCOMPRESS + $)
($ CONFIGURATION CON8P $)

PROGRAM B806B;

(* Blob filling task - 8 blobs, 64 x 64 pixel image *)

(* Wildfire algorithm *)
(* Private and global lists *)
(* Private to global job transfer if idle units available *)

GLOBAL TASK;

VAR     IMAGE           :ARRAY[0..4095] OF INTEGER;

        (* Global job list *)
        GLISTX,GLISTY   :ARRAY[0..2047] OF INTEGER;
        TOPOFGLIST      :INTEGER;

        (* Global job list semaphore *)
        LISTACCESS      :INTEGER;

        (* Count of processors with nothing to do *)
        IDLEUNITS       :INTEGER;

        (* Count of active jobs *)
        JOBSINSYSTEM    :INTEGER;


PROCEDURE GIGNITE(X,Y:INTEGER;);

(* Create a job on the global job list *)

  BEGIN
  INCREMENT(JOBSINSYSTEM);
  WAIT(LISTACCESS);
  INCREMENT(TOPOFGLIST);
  GLISTX[TOPOFGLIST]:=X;
  GLISTY[TOPOFGLIST]:=Y;
  SIGNAL(LISTACCESS);
  END;


PROCEDURE PROCESS(N:INTEGER;);

(* Perform main process *)

VAR (* Private job list *)
    PLISTX,PLISTY   :ARRAY[0..255] OF INTEGER;
    TOPOFPLIST      :INTEGER;

    (* Misc.  *)
    COUNTER         :INTEGER;
    PX,PY           :INTEGER;

  PROCEDURE PIGNITE(X,Y:INTEGER;);

  (* Create a job on a private job list *)

    BEGIN
    INCREMENT(JOBSINSYSTEM);
    INCREMENT(TOPOFPLIST);
    PLISTX[TOPOFPLIST]:=X;
    PLISTY[TOPOFPLIST]:=Y;
    END;
```

```
PROCEDURE SPREAD(X,Y:INTEGER;);

(* Spread vertically and horizontally.                *)
(* Each point is checked before it is PIGNITEd, since *)
(* this will have to be done eventually, and checking *)
(* first reduces job list sizes and overheads.        *)

  BEGIN
  (* 63>=X>=0,  63>=Y>=0 *)
  IF IMAGE[X+64*Y]=1 THEN
      BEGIN
      IMAGE[X+64*Y]:=0;
      IF (X>0) THEN IF (IMAGE[X-1+64*Y]=1) THEN
          PIGNITE(X-1,Y);
      IF (X<63) THEN IF (IMAGE[X+1+64*Y]=1) THEN
          PIGNITE(X+1,Y);
      IF (Y>0) THEN IF (IMAGE[X+64*(Y-1)]=1) THEN
          PIGNITE(X,Y-1);
      IF (Y<63) THEN IF (IMAGE[X+64*(Y+1)]=1) THEN
          PIGNITE(X,Y+1);
      END;
  DECREMENT(JOBSINSYSTEM);
  END;


BEGIN
TOPOFPLIST:=-1;
INCREMENT(IDLEUNITS);
REPEAT

  ($ ENTER Wait1 $)
  WAIT(LISTACCESS);
  ($ EXIT Wait1 $)

  IF TOPOFGLIST>=0 THEN
    BEGIN
    (* This processor is now busy *)
    DECREMENT(IDLEUNITS);

    ($ ENTER Getglbl $)
    (* Move Job from Global list to private list *)
    (* Decrement first, so that global list doesnt *)
    (* look fuller than it is, to other processors. *)
    DECREMENT(TOPOFGLIST);
    INCREMENT(TOPOFPLIST);
    PLISTX[TOPOFPLIST]:=GLISTX[TOPOFGLIST+1];
    PLISTY[TOPOFPLIST]:=GLISTY[TOPOFGLIST+1];
    SIGNAL(LISTACCESS);
    ($ EXIT Getglbl $)

    ($ ENTER Private $)
    REPEAT
      (* Get a job from the private list, and do it *)
      PX:=PLISTX[TOPOFPLIST];
      PY:=PLISTY[TOPOFPLIST];
      DECREMENT(TOPOFPLIST);
      SPREAD(PX,PY);

      ($ ENTER Sendglbl $)

      (* If there are more processes idle than there are *)
      (* jobs for them on the global job list, AND *)
      (* this private list is not empty, transfer *)
      (* half of the private list to the global list *)

      IF IDLEUNITS>TOPOFGLIST+1 THEN
        IF TOPOFPLIST>=1 THEN
          BEGIN

          ($ ENTER Wait2 $)
          WAIT(LISTACCESS);
          ($ EXIT Wait2 $)
```

```
            FOR COUNTER:=0 TO (TOPOFPLIST-1)/2 DO
              BEGIN
              INCREMENT(TOPOFGLIST);
              GLISTX[TOPOFGLIST]:=PLISTX[TOPOFPLIST];
              GLISTY[TOPOFGLIST]:=PLISTY[TOPOFPLIST];
              DECREMENT(TOPOFPLIST);
              END;
            SIGNAL(LISTACCESS);
            END;
      ($ EXIT Sendglbl $)

      UNTIL TOPOFPLIST=-1;
      ($ EXIT Private $)

      (* Nothing to do now, until a job can be *)
      (* found on the global job list *)
      INCREMENT(IDLEUNITS);
      END
    ELSE
      BEGIN
      ($ ENTER Oops $)
      (* Nothing to do *)
      SIGNAL(LISTACCESS);
      ($ EXIT Oops $)
      END;
  UNTIL JOBSINSYSTEM=0;
  END;


BEGIN
(* Initialisation *)
TOPOFGLIST:=-1;
LISTACCESS:=1;

IDLEUNITS:=0;
JOBSINSYSTEM:=0;

(* Initialise Job list with known positions of blobs *)
 GIGNITE( 8, 8);GIGNITE(24, 8);GIGNITE(40, 8);GIGNITE(56, 8);
 GIGNITE( 8,24);GIGNITE(24,24);GIGNITE(40,24);GIGNITE(56,24);
{GIGNITE( 8,40);GIGNITE(24,40);GIGNITE(40,40);GIGNITE(56,40);}
{GIGNITE( 8,56);GIGNITE(24,56);GIGNITE(40,56);GIGNITE(56,56);}

END;


GENERAL TASK;

  BEGIN
  END;

END.
```

A3.7 Region Filler B807B

```
($ COMPRESSCODE + $)
($ SUPERCOMPRESS + $)
($ CONFIGURATION CON8P $)

PROGRAM B807B;

(* Blob filling task - 8 blobs, 64 x 64 pixel image *)

(* Wildfire algorithm *)
(* Global lists only *)
(* Wait interference reduced by pre-checking *)

GLOBAL TASK;

VAR     IMAGE           :ARRAY[0..4095] OF INTEGER;

        (* Global job list *)
        GLISTX,GLISTY   :ARRAY[0..2047] OF INTEGER;
        TOPOFGLIST      :INTEGER;

        (* Global job list semaphore *)
        LISTACCESS      :INTEGER;

        (* Count of processors with nothing to do *)
        IDLEUNITS       :INTEGER;

        (* Count of active jobs *)
        JOBSINSYSTEM    :INTEGER;


PROCEDURE GIGNITE(X,Y:INTEGER;);

(* Create a job on the global job list *)

  BEGIN
  INCREMENT(JOBSINSYSTEM);
  WAIT(LISTACCESS);
  INCREMENT(TOPOFGLIST);
  GLISTX[TOPOFGLIST]:=X;
  GLISTY[TOPOFGLIST]:=Y;
  SIGNAL(LISTACCESS);
  END;

PROCEDURE PROCESS(N:INTEGER;);

(* Perform main process *)

VAR (* Misc.  *)
    COUNTER         :INTEGER;
    PX,PY           :INTEGER;

  PROCEDURE SPREAD(X,Y:INTEGER;);

  (* Spread vertically and horizontally.                 *)
  (* Each point is checked before it is GIGNITEd, since *)
  (* this will have to be done eventually, and checking *)
  (* first reduces job list sizes and overheads.         *)

    BEGIN
    (* 63>=X>=0,  63>=Y>=0 *)
    IF IMAGE[X+64*Y]=1 THEN
        BEGIN
        IMAGE[X+64*Y]:=0;
        IF (X>0) THEN IF (IMAGE[X-1+64*Y]=1) THEN
            GIGNITE(X-1,Y);
        IF (X<63) THEN IF (IMAGE[X+1+64*Y]=1) THEN
            GIGNITE(X+1,Y);
        IF (Y>0) THEN IF (IMAGE[X+64*(Y-1)]=1) THEN
            GIGNITE(X,Y-1);
        IF (Y<63) THEN IF (IMAGE[X+64*(Y+1)]=1) THEN
            GIGNITE(X,Y+1);
        END;
    DECREMENT(JOBSINSYSTEM);
    END;
```

```
   BEGIN
   INCREMENT(IDLEUNITS);
   REPEAT

     ($ ENTER Prewait $)
     REPEAT
     (* Wait until there is a good chance of a successful *)
     (* WAIT, and of there being a job on the global job *)
     (* list *)
     UNTIL ((TOPOFGLIST>=0) AND (LISTACCESS>0))
                OR (JOBSINSYSTEM=0);
     ($ EXIT Prewait $)

     ($ ENTER Wait1 $)
     WAIT(LISTACCESS);
     ($ EXIT Wait1 $)

     IF TOPOFGLIST>=0 THEN
       BEGIN
       (* This processor is now busy *)
       DECREMENT(IDLEUNITS);

       (* Get a job from the global list, and do it *)

       ($ ENTER Getglbl $)
       PX:=GLISTX[TOPOFGLIST];
       PY:=GLISTY[TOPOFGLIST];
       DECREMENT(TOPOFGLIST);
       SIGNAL(LISTACCESS);
       ($ EXIT Getglbl $)

       SPREAD(PX,PY);

       (* Nothing to do now, until a job can be *)
       (* found on the global job list *)
       INCREMENT(IDLEUNITS);
       END
ELSE
       BEGIN
       ($ ENTER Oops $)
       (* Nothing to do, return to Prewait *)
       SIGNAL(LISTACCESS);
       ($ EXIT Oops $)
       END;
  UNTIL JOBSINSYSTEM=0;
  END;



BEGIN
(* Initialisation *)
TOPOFGLIST:=-1;
LISTACCESS:=1;

IDLEUNITS:=0;
JOBSINSYSTEM:=0;

(* Initialise Job list with known positions of blobs *)
 GIGNITE( 8, 8);GIGNITE(24, 8);GIGNITE(40, 8);GIGNITE(56, 8);
 GIGNITE( 8,24);GIGNITE(24,24);GIGNITE(40,24);GIGNITE(56,24);
{GIGNITE( 8,40);GIGNITE(24,40);GIGNITE(40,40);GIGNITE(56,40);}
{GIGNITE( 8,56);GIGNITE(24,56);GIGNITE(40,56);GIGNITE(56,56);}

END;



GENERAL TASK;

  BEGIN
  END;

END.
```

A3.8 Region Filler B808B

```
($ COMPRESSCODE + $)
($ SUPERCOMPRESS + $)
($ CONFIGURATION CONSP $)

PROGRAM B808B;

(* Blob filling task - 8 blobs, 64 x 64 pixel image *)

(* Wildfire algorithm *)
(* Global lists only *)

GLOBAL TASK;

VAR     IMAGE           :ARRAY[0..4095] OF INTEGER;

        (* Global job list *)
        GLISTX,GLISTY   :ARRAY[0..2047] OF INTEGER;
        TOPOFGLIST      :INTEGER;

        (* Global job list semaphore *)
        LISTACCESS      :INTEGER;

        (* Count of processors with nothing to do *)
        IDLEUNITS       :INTEGER;

        (* Count of active jobs *)
        JOBSINSYSTEM    :INTEGER;


PROCEDURE GIGNITE(X,Y:INTEGER;);

(* Create a job on the global job list *)

  BEGIN
  INCREMENT(JOBSINSYSTEM);
  WAIT(LISTACCESS);
  INCREMENT(TOPOFGLIST);
  GLISTX[TOPOFGLIST]:=X;
  GLISTY[TOPOFGLIST]:=Y;
  SIGNAL(LISTACCESS);
  END;


PROCEDURE PROCESS(N:INTEGER;);

(* Perform main process *)

VAR (* Misc.  *)
    COUNTER         :INTEGER;
    PX,PY           :INTEGER;

  PROCEDURE SPREAD(X,Y:INTEGER;);

  (* Spread vertically and horizontally.                *)
  (* Each point is checked before it is GIGNITEd, since *)
  (* this will have to be done eventually, and checking *)
  (* first reduces job list sizes and overheads.        *)

    BEGIN
    (* 63>=X>=0,  63>=Y>=0 *)
    IF IMAGE[X+64*Y]=1 THEN
        BEGIN
        IMAGE[X+64*Y]:=0;
        IF (X>0) THEN IF (IMAGE[X-1+64*Y]=1) THEN
            GIGNITE(X-1,Y);
        IF (X<63) THEN IF (IMAGE[X+1+64*Y]=1) THEN
            GIGNITE(X+1,Y);
        IF (Y>0) THEN IF (IMAGE[X+64*(Y-1)]=1) THEN
            GIGNITE(X,Y-1);
        IF (Y<63) THEN IF (IMAGE[X+64*(Y+1)]=1) THEN
            GIGNITE(X,Y+1);
        END;
    DECREMENT(JOBSINSYSTEM);
    END;
```

```
  BEGIN
  INCREMENT(IDLEUNITS);
  REPEAT

    ($ ENTER Wait1 $)
    WAIT(LISTACCESS);
    ($ EXIT Wait1 $)

    IF TOPOFGLIST>=0 THEN
      BEGIN
      (* This processor is now busy *)
      DECREMENT(IDLEUNITS);

      (* Get a job from the global list, and do it *)

      ($ ENTER Getglbl $)
      PX:=GLISTX[TOPOFGLIST];
      PY:=GLISTY[TOPOFGLIST];
      DECREMENT(TOPOFGLIST);
      SIGNAL(LISTACCESS);
      ($ EXIT Getglbl $)

      SPREAD(PX,PY);

      (* Nothing to do now, until a job can be *)
      (* found on the global job list *)
      INCREMENT(IDLEUNITS);
      END
    ELSE
      BEGIN
      ($ ENTER Oops $)
      (* Nothing to do *)
      SIGNAL(LISTACCESS);
      ($ EXIT Oops $)
      END;
  UNTIL JOBSINSYSTEM=0;
  END;


BEGIN
(* Initialisation *)
TOPOFGLIST:=-1;
LISTACCESS:=1;

IDLEUNITS:=0;
JOBSINSYSTEM:=0;

(* Initialise Job list with known positions of blobs *)
 GIGNITE( 8, 8);GIGNITE(24, 8);GIGNITE(40, 8);GIGNITE(56, 8);
 GIGNITE( 8,24);GIGNITE(24,24);GIGNITE(40,24);GIGNITE(56,24);
{GIGNITE( 8,40);GIGNITE(24,40);GIGNITE(40,40);GIGNITE(56,40);}
{GIGNITE( 8,56);GIGNITE(24,56);GIGNITE(40,56);GIGNITE(56,56);}

END;


GENERAL TASK;

  BEGIN
  END;

END.
```

A3.9 Integer Sorter Q801B

```
($ COMPRESSCODE + $)
($ SUPERCOMPRESS + $)
($ CONFIGURATION CON8P $)

PROGRAM Q801B;

(* Quicksort *)

(* Private and global lists *)
(* Wait interference reduced by pre-checking *)
(* Private to global job transfer if idle units available *)

GLOBAL TASK;

VAR     (* List of numbers to be sorted *)
        A               :ARRAY[0..255] OF INTEGER;

        (* Global Job lists *)
        GLISTT,GLISTB   :ARRAY[0..127] OF INTEGER;

        (* Pointers *)
        APTR,
        TOPOFGLIST      :INTEGER;

        (* Global job list semaphore *)
        LISTACCESS      :INTEGER;

        (* Count of processors with nothing to do *)
        IDLEUNITS       :INTEGER;

        (* Termination count *)
        JOBSINSYSTEM    :INTEGER;


PROCEDURE GLOBALJOB(L,H:INTEGER;);

(* Create a job on the global job list *)

    BEGIN
    INCREMENT(JOBSINSYSTEM);
    WAIT(LISTACCESS);
    TOPOFGLIST:=TOPOFGLIST+1;
    GLISTT[TOPOFGLIST]:=H;
    GLISTB[TOPOFGLIST]:=L;
    SIGNAL(LISTACCESS);
    END;


PROCEDURE PROCESS(N:INTEGER;);

VAR (* Private Job list *)
    PLISTT,PLISTB       :ARRAY[0..127] OF INTEGER;

    (* Pointer *)
    TOPOFPLIST          :INTEGER;

    (* Misc. *)
    COUNTER,
    B,T                 :INTEGER;


    PROCEDURE PRIVATEJOB(L,H:INTEGER;);

    (* Create a job on the private job list *)

        BEGIN
        INCREMENT(JOBSINSYSTEM);
        TOPOFPLIST:=TOPOFPLIST+1;
        PLISTT[TOPOFPLIST]:=H;
        PLISTB[TOPOFPLIST]:=L;
        END;
```

```
PROCEDURE SORT(LO,HI:INTEGER;);

(* Sort the numbers in the global array from *)
(* element LO to element HI inclusive, using *)
(* the QUICKSORT algorithm. *)

VAR SPLITVAL,TEMP,
    LPTR,HPTR,
    SPLITPT          :INTEGER;

    BEGIN
    IF HI>LO THEN
        BEGIN
        ($ ENTER Setptrs $)
        (* Choose value to split at *)
        (* from centre of list *)
        SPLITVAL:=A[(LO+HI)/2];
        LPTR:=LO;
        HPTR:=HI;
        ($ EXIT Setptrs $)

        REPEAT
            (* Move LPTR up the list *)
            ($ ENTER Scanup $)
            IF (A[LPTR]<SPLITVAL) THEN
               BEGIN
               REPEAT
                   LPTR:=LPTR+1;
               UNTIL (A[LPTR]>=SPLITVAL);
               END;
            ($ EXIT Scanup $)

            (* Move HPTR up the list *)
            ($ ENTER Scandown $)
            IF (A[HPTR]>SPLITVAL) THEN
                BEGIN
                REPEAT
                    HPTR:=HPTR-1;
                UNTIL (A[HPTR]<=SPLITVAL);
                END;
            ($ EXIT Scandown $)

            (* Swap elements, provided pointers *)
            (* have not passed each other.      *)
            ($ ENTER Exchange $)
            IF LPTR<=HPTR THEN
                BEGIN
                TEMP:=A[LPTR];
                A[LPTR]:=A[HPTR];
                A[HPTR]:=TEMP;
                HPTR:=HPTR-1;
                LPTR:=LPTR+1;
                END;
            ($ EXIT Exchange $)

        UNTIL LPTR>HPTR;

        (* Create new jobs *)
        ($ ENTER Createjobs $)
        IF LO<HPTR THEN
            PRIVATEJOB(LO,HPTR);
        IF LPTR<HI THEN
            PRIVATEJOB(LPTR,HI);
        ($ EXIT Createjobs $)
        END;
    DECREMENT(JOBSINSYSTEM);
    END;
```

```
BEGIN
TOPOFPLIST:=-1;
INCREMENT(IDLEUNITS);
REPEAT

  ($ ENTER Prewait $)
  REPEAT
  (* Wait until there is a good chance of a successful *)
  (* WAIT, and of there being a job on the global job *)
  (* list *)
  UNTIL ((TOPOFGLIST>=0) AND (LISTACCESS>0))
            OR (JOBSINSYSTEM=0);
  ($ EXIT Prewait $)

  ($ ENTER Wait1 $)
  WAIT(LISTACCESS);
  ($ EXIT Wait1 $)

  IF TOPOFGLIST>=0 THEN
    BEGIN
    (* This processor is now busy *)
    DECREMENT(IDLEUNITS);

    ($ ENTER Getglbl $)
    (* Move Job from Global list to private list *)
    (* Decrement first, so that global list doesnt *)
    (* look fuller than it is, to other processors. *)
    DECREMENT(TOPOFGLIST);
    INCREMENT(TOPOFPLIST);
    PLISTT[TOPOFPLIST]:=GLISTT[TOPOFGLIST+1];
    PLISTB[TOPOFPLIST]:=GLISTB[TOPOFGLIST+1];
    SIGNAL(LISTACCESS);
    ($ EXIT Getglbl $)

    ($ ENTER Private $)
    REPEAT
      (* Get a job from the private list, and do it *)
      T:=PLISTT[TOPOFPLIST];
      B:=PLISTB[TOPOFPLIST];
      DECREMENT(TOPOFPLIST);
      SORT(B,T);

      ($ ENTER Sendglbl $)

      (* If there are more processes idle than there are *)
      (* jobs for them on the global job list, AND *)
      (* this private list is not empty, transfer *)
      (* half of the private list to the global list *)

      IF IDLEUNITS>TOPOFGLIST+1 THEN
        IF TOPOFPLIST>=1 THEN
          BEGIN

          ($ ENTER Wait2 $)
          WAIT(LISTACCESS);
          ($ EXIT Wait2 $)

          FOR COUNTER:=0 TO (TOPOFPLIST-1)/2 DO
            BEGIN
            INCREMENT(TOPOFGLIST);
            GLISTT[TOPOFGLIST]:=PLISTT[TOPOFPLIST];
            GLISTB[TOPOFGLIST]:=PLISTB[TOPOFPLIST];
            DECREMENT(TOPOFPLIST);
            END;
          SIGNAL(LISTACCESS);
          END;
    ($ EXIT Sendglbl $)

    UNTIL TOPOFPLIST=-1;
    ($ EXIT Private $)
```

```
      (* Nothing to do now, until a job can be *)
      (* found on the global job list *)
      INCREMENT(IDLEUNITS);
      END
    ELSE
      BEGIN
      ($ ENTER Oops $)
      (* Nothing to do, return to Prewait *)
      SIGNAL(LISTACCESS);
      ($ EXIT Oops $)
      END;
  UNTIL JOBSINSYSTEM=0;
  END;


BEGIN
LISTACCESS:=1;

(* A list of pre-calculated random numbers *)
(*     will be loaded at run time.         *)
(*     All that is required here is to set *)
(*     the list pointer, to indicate the   *)
(*     number of elements to be sorted.    *)

APTR:=255;

TOPOFGLIST:=-1;
JOBSINSYSTEM:=0;
GLOBALJOB(0,APTR);
END;



GENERAL TASK;

  BEGIN
  END;

END.
```

A3.10 Integer Sorter Q802B

```
($ COMPRESSCODE + $)
($ SUPERCOMPRESS + $)
($ CONFIGURATION CON8P $)

PROGRAM Q802B;

(* Quicksort *)

(* Private and global lists *)
(* Private to global job transfer if idle units available *)

GLOBAL TASK;

VAR     (* List of numbers to be sorted *)
        A               :ARRAY[0..255] OF INTEGER;

        (* Global Job lists *)
        GLISTT,GLISTB   :ARRAY[0..127] OF INTEGER;

        (* Pointers *)
        APTR,
        TOPOFGLIST      :INTEGER;

        (* Global job list semaphore *)
        LISTACCESS      :INTEGER;

        (* Count of processors with nothing to do *)
        IDLEUNITS       :INTEGER;

        (* Termination count *)
        JOBSINSYSTEM    :INTEGER;


PROCEDURE GLOBALJOB(L,H:INTEGER;);

(* Create a job on the global job list *)

    BEGIN
    INCREMENT(JOBSINSYSTEM);
    WAIT(LISTACCESS);
    TOPOFGLIST:=TOPOFGLIST+1;
    GLISTT[TOPOFGLIST]:=H;
    GLISTB[TOPOFGLIST]:=L;
    SIGNAL(LISTACCESS);
    END;


PROCEDURE PROCESS(N:INTEGER;);

VAR (* Private Job list *)
    PLISTT,PLISTB       :ARRAY[0..127] OF INTEGER;

    (* Pointer *)
    TOPOFPLIST          :INTEGER;

    (* Misc. *)
    COUNTER,
    B,T                 :INTEGER;


    PROCEDURE PRIVATEJOB(L,H:INTEGER;);

    (* Create a job on the private job list *)

        BEGIN
        INCREMENT(JOBSINSYSTEM);
        TOPOFPLIST:=TOPOFPLIST+1;
        PLISTT[TOPOFPLIST]:=H;
        PLISTB[TOPOFPLIST]:=L;
        END;
```

```
PROCEDURE SORT(LO,HI:INTEGER;);

(* Sort the numbers in the global array from *)
(* element LO to element HI inclusive, using *)
(* the QUICKSORT algorithm. *)

VAR SPLITVAL,TEMP,
    LPTR,HPTR,
    SPLITPT          :INTEGER;

    BEGIN
    IF HI>LO THEN
        BEGIN
        ($ ENTER Setptrs $)
        (* Choose value to split at *)
        (* from centre of list *)
        SPLITVAL:=A[(LO+HI)/2];
        LPTR:=LO;
        HPTR:=HI;
        ($ EXIT Setptrs $)

        REPEAT
            (* Move LPTR up the list *)
            ($ ENTER Scanup $)
            IF (A[LPTR]<SPLITVAL) THEN
               BEGIN
               REPEAT
                   LPTR:=LPTR+1;
               UNTIL (A[LPTR]>=SPLITVAL);
               END;
            ($ EXIT Scanup $)

            (* Move HPTR up the list *)
            ($ ENTER Scandown $)
            IF (A[HPTR]>SPLITVAL) THEN
                BEGIN
                REPEAT
                    HPTR:=HPTR-1;
                UNTIL (A[HPTR]<=SPLITVAL);
                END;
            ($ EXIT Scandown $)

            (* Swap elements, provided pointers *)
            (* have not passed each other.      *)
            ($ ENTER Exchange $)
            IF LPTR<=HPTR THEN
                BEGIN
                TEMP:=A[LPTR];
                A[LPTR]:=A[HPTR];
                A[HPTR]:=TEMP;
                HPTR:=HPTR-1;
                LPTR:=LPTR+1;
                END;
            ($ EXIT Exchange $)

        UNTIL LPTR>HPTR;

        (* Create new jobs *)
        ($ ENTER Createjobs $)
        IF LO<HPTR THEN
            PRIVATEJOB(LO,HPTR);
        IF LPTR<HI THEN
            PRIVATEJOB(LPTR,HI);
        ($ EXIT Createjobs $)
        END;
    DECREMENT(JOBSINSYSTEM);
    END;
```

```
BEGIN
TOPOFPLIST:=-1;
INCREMENT(IDLEUNITS);
REPEAT

  ($ ENTER Wait1 $)
  WAIT(LISTACCESS);
  ($ EXIT Wait1 $)

  IF TOPOFGLIST>=0 THEN
    BEGIN
    (* This processor is now busy *)
    DECREMENT(IDLEUNITS);

    ($ ENTER Getglbl $)
    (* Move Job from Global list to private list *)
    (* Decrement first, so that global list doesnt *)
    (* look fuller than it is, to other processors. *)
    DECREMENT(TOPOFGLIST);
    INCREMENT(TOPOFPLIST);
    PLISTB[TOPOFPLIST]:=GLISTB[TOPOFGLIST+1];
    PLISTT[TOPOFPLIST]:=GLISTT[TOPOFGLIST+1];
    SIGNAL(LISTACCESS);
    ($ EXIT Getglbl $)

    ($ ENTER Private $)
    REPEAT
      (* Get a job from the private list, and do it *)
      B:=PLISTB[TOPOFPLIST];
      T:=PLISTT[TOPOFPLIST];
      DECREMENT(TOPOFPLIST);
      SORT(B,T);

      ($ ENTER Sendglbl $)

      (* If there are more processes idle than there are *)
      (* jobs for them on the global job list, AND *)
      (* this private list is not empty, transfer *)
      (* half of the private list to the global list *)

      IF IDLEUNITS>TOPOFGLIST+1 THEN
        IF TOPOFPLIST>=1 THEN
          BEGIN

          ($ ENTER Wait2 $)
          WAIT(LISTACCESS);
          ($ EXIT Wait2 $)

          FOR COUNTER:=0 TO (TOPOFPLIST-1)/2 DO
            BEGIN
            INCREMENT(TOPOFGLIST);
            GLISTB[TOPOFGLIST]:=PLISTB[TOPOFPLIST];
            GLISTT[TOPOFGLIST]:=PLISTT[TOPOFPLIST];
            DECREMENT(TOPOFPLIST);
            END;
          SIGNAL(LISTACCESS);
          END;
    ($ EXIT Sendglbl $)

    UNTIL TOPOFPLIST=-1;
    ($ EXIT Private $)
```

```
      (* Nothing to do now, until a job can be *)
      (* found on the global job list *)
      INCREMENT(IDLEUNITS);
      END
    ELSE
      BEGIN
      ($ ENTER Oops $)
      (* Nothing to do *)
      SIGNAL(LISTACCESS);
      ($ EXIT Oops $)
      END;
  UNTIL JOBSINSYSTEM=0;
  END;


BEGIN
LISTACCESS:=1;

(* A list of pre-calculated random numbers *)
(*     will be loaded at run time.         *)
(*     All that is required here is to set *)
(*     the list pointer, to indicate the   *)
(*     number of elements to be sorted.    *)

APTR:=255;

TOPOFGLIST:=-1;
JOBSINSYSTEM:=0;
GLOBALJOB(0,APTR);
END;



GENERAL TASK;

  BEGIN
  END;

END.
```

A3.11 Integer Sorter Q803B

```
($ COMPRESSCODE + $)
($ SUPERCOMPRESS + $)
($ CONFIGURATION CON8P $)

PROGRAM Q803B;

(* Quicksort *)

(* Global lists only *)
(* Wait interference reduced by pre-checking *)

GLOBAL TASK;

VAR     (* List of numbers to be sorted *)
        A               :ARRAY[0..255] OF INTEGER;

        (* Global Job lists *)
        GLISTT,GLISTB   :ARRAY[0..127] OF INTEGER;

        (* Pointers *)
        APTR,
        TOPOFGLIST      :INTEGER;

        (* Global job list semaphore *)
        LISTACCESS      :INTEGER;

        (* Count of processors with nothing to do *)
        IDLEUNITS       :INTEGER;

        (* Termination count *)
        JOBSINSYSTEM    :INTEGER;


PROCEDURE GLOBALJOB(L,H:INTEGER;);

(* Create a job on the global job list *)

    BEGIN
    INCREMENT(JOBSINSYSTEM);
    WAIT(LISTACCESS);
    TOPOFGLIST:=TOPOFGLIST+1;
    GLISTT[TOPOFGLIST]:=H;
    GLISTB[TOPOFGLIST]:=L;
    SIGNAL(LISTACCESS);
    END;


PROCEDURE PROCESS(N:INTEGER;);

VAR (* Misc. *)
    COUNTER,
    B,T                 :INTEGER;


    PROCEDURE SORT(LO,HI:INTEGER;);

    (* Sort the numbers in the global array from *)
    (* element LO to element HI inclusive, using *)
    (* the QUICKSORT algorithm. *)

    VAR SPLITVAL,TEMP,
        LPTR,HPTR,
        SPLITPT         :INTEGER;

        BEGIN
        IF HI>LO THEN
            BEGIN
            ($ ENTER Setptrs $)
            (* Choose value to split at *)
            (* from centre of list *)
            SPLITVAL:=A[(LO+HI)/2];
            LPTR:=LO;
            HPTR:=HI;
            ($ EXIT Setptrs $)
```

```
            REPEAT
                (* Move LPTR up the list *)
                ($ ENTER Scanup $)
                IF (A[LPTR]<SPLITVAL) THEN
                   BEGIN
                   REPEAT
                       LPTR:=LPTR+1;
                   UNTIL (A[LPTR]>=SPLITVAL);
                   END;
                ($ EXIT Scanup $)

                (* Move HPTR up the list *)
                ($ ENTER Scandown $)
                IF (A[HPTR]>SPLITVAL) THEN
                    BEGIN
                    REPEAT
                        HPTR:=HPTR-1;
                    UNTIL (A[HPTR]<=SPLITVAL);
                    END;
                ($ EXIT Scandown $)

                (* Swap elements, provided pointers *)
                (* have not passed each other.      *)
                ($ ENTER Exchange $)
                IF LPTR<=HPTR THEN
                    BEGIN
                    TEMP:=A[LPTR];
                    A[LPTR]:=A[HPTR];
                    A[HPTR]:=TEMP;
                    HPTR:=HPTR-1;
                    LPTR:=LPTR+1;
                    END;
                ($ EXIT Exchange $)

            UNTIL LPTR>HPTR;

            (* Create new jobs *)
            ($ ENTER Createjobs $)
            IF LO<HPTR THEN
                GLOBALJOB(LO,HPTR);
            IF LPTR<HI THEN
                GLOBALJOB(LPTR,HI);
            ($ EXIT Createjobs $)
            END;
        DECREMENT(JOBSINSYSTEM);
        END;


BEGIN
INCREMENT(IDLEUNITS);
REPEAT

  ($ ENTER Prewait $)
  REPEAT
  (* Wait until there is a good chance of a successful *)
  (* WAIT, and of there being a job on the global job *)
  (* list *)
  UNTIL ((TOPOFGLIST>=0) AND (LISTACCESS>0))
              OR (JOBSINSYSTEM=0);
  ($ EXIT Prewait $)

  ($ ENTER Wait1 $)
  WAIT(LISTACCESS);
  ($ EXIT Wait1 $)
```

```
    IF TOPOFGLIST>=0 THEN
      BEGIN
      (* This processor is now busy *)
      DECREMENT(IDLEUNITS);

      (* Get a job from the global list, and do it *)

      ($ ENTER Getglbl $)
      B:=GLISTB[TOPOFGLIST];
      T:=GLISTT[TOPOFGLIST];
      DECREMENT(TOPOFGLIST);
      SIGNAL(LISTACCESS);
      ($ EXIT Getglbl $)

      SORT(B,T);

      (* Nothing to do now, until a job can be *)
      (* found on the global job list *)
      INCREMENT(IDLEUNITS);
      END
    ELSE
      BEGIN
      ($ ENTER Oops $)
      (* Nothing to do, return to Prewait *)
      SIGNAL(LISTACCESS);
      ($ EXIT Oops $)
      END;
  UNTIL JOBSINSYSTEM=0;
  END;


BEGIN
LISTACCESS:=1;

(* A list of pre-calculated random numbers *)
(*     will be loaded at run time.         *)
(*     All that is required here is to set *)
(*     the list pointer, to indicate the   *)
(*     number of elements to be sorted.    *)

APTR:=255;

TOPOFGLIST:=-1;
JOBSINSYSTEM:=0;
GLOBALJOB(0,APTR);
END;


GENERAL TASK;

  BEGIN
  END;

END.
```

A3.12 Integer Sorter Q804B

```
($ COMPRESSCODE + $)
($ SUPERCOMPRESS + $)
($ CONFIGURATION CON8P $)

PROGRAM Q804B;

(* Quicksort *)

(* Global lists only *)

GLOBAL TASK;

VAR     (* List of numbers to be sorted *)
        A               :ARRAY[0..255] OF INTEGER;

        (* Global Job lists *)
        GLISTT,GLISTB   :ARRAY[0..127] OF INTEGER;

        (* Pointers *)
        APTR,
        TOPOFGLIST      :INTEGER;

        (* Global job list semaphore *)
        LISTACCESS      :INTEGER;

        (* Count of processors with nothing to do *)
        IDLEUNITS       :INTEGER;

        (* Termination count *)
        JOBSINSYSTEM    :INTEGER;


PROCEDURE GLOBALJOB(L,H:INTEGER;);

(* Create a job on the global job list *)

    BEGIN
    INCREMENT(JOBSINSYSTEM);
    WAIT(LISTACCESS);
    TOPOFGLIST:=TOPOFGLIST+1;
    GLISTT[TOPOFGLIST]:=H;
    GLISTB[TOPOFGLIST]:=L;
    SIGNAL(LISTACCESS);
    END;


PROCEDURE PROCESS(N:INTEGER;);

VAR (* Misc. *)
    COUNTER,
    B,T                 :INTEGER;


    PROCEDURE SORT(LO,HI:INTEGER;);

    (* Sort the numbers in the global array from *)
    (* element LO to element HI inclusive, using *)
    (* the QUICKSORT algorithm. *)

    VAR SPLITVAL,TEMP,
        LPTR,HPTR,
        SPLITPT         :INTEGER;

        BEGIN
        IF HI>LO THEN
            BEGIN
            ($ ENTER Setptrs $)
            (* Choose value to split at *)
            (* from centre of list *)
            SPLITVAL:=A[(LO+HI)/2];
            LPTR:=LO;
            HPTR:=HI;
            ($ EXIT Setptrs $)
```

```
    REPEAT
        (* Move LPTR up the list *)
        ($ ENTER Scanup $)
        IF (A[LPTR]<SPLITVAL) THEN
           BEGIN
           REPEAT
               LPTR:=LPTR+1;
           UNTIL (A[LPTR]>=SPLITVAL);
           END;
        ($ EXIT Scanup $)

        (* Move HPTR up the list *)
        ($ ENTER Scandown $)
        IF (A[HPTR]>SPLITVAL) THEN
            BEGIN
            REPEAT
                HPTR:=HPTR-1;
            UNTIL (A[HPTR]<=SPLITVAL);
            END;
        ($ EXIT Scandown $)

        (* Swap elements, provided pointers *)
        (* have not passed each other.      *)
        ($ ENTER Exchange $)
        IF LPTR<=HPTR THEN
            BEGIN
            TEMP:=A[LPTR];
            A[LPTR]:=A[HPTR];
            A[HPTR]:=TEMP;
            HPTR:=HPTR-1;
            LPTR:=LPTR+1;
            END;
        ($ EXIT Exchange $)

    UNTIL LPTR>HPTR;

    (* Create new jobs *)
    ($ ENTER Createjobs $)
    IF LO<HPTR THEN
        GLOBALJOB(LO,HPTR);
    IF LPTR<HI THEN
        GLOBALJOB(LPTR,HI);
    ($ EXIT Createjobs $)
    END;
DECREMENT(JOBSINSYSTEM);
END;
```

```
  BEGIN
  INCREMENT(IDLEUNITS);
  REPEAT

    ($ ENTER Wait1 $)
    WAIT(LISTACCESS);
    ($ EXIT Wait1 $)

    IF TOPOFGLIST>=0 THEN
      BEGIN
      (* This processor is now busy *)
      DECREMENT(IDLEUNITS);

      (* Get a job from the global list, and do it *)

      ($ ENTER Getglbl $)
      B:=GLISTB[TOPOFGLIST];
      T:=GLISTT[TOPOFGLIST];
      DECREMENT(TOPOFGLIST);
      SIGNAL(LISTACCESS);
      ($ EXIT Getglbl $)

      SORT(B,T);

      (* Nothing to do now, until a job can be *)
      (* found on the global job list *)
      INCREMENT(IDLEUNITS);
      END
    ELSE
      BEGIN
      ($ ENTER Oops $)
      (* Nothing to do *)
      SIGNAL(LISTACCESS);
      ($ EXIT Oops $)
      END;
  UNTIL JOBSINSYSTEM=0;
  END;


BEGIN
LISTACCESS:=1;

(* A list of pre-calculated random numbers *)
(*     will be loaded at run time.         *)
(*     All that is required here is to set *)
(*     the list pointer, to indicate the   *)
(*     number of elements to be sorted.    *)

APTR:=255;

TOPOFGLIST:=-1;
JOBSINSYSTEM:=0;
GLOBALJOB(0,APTR);
END;



GENERAL TASK;

  BEGIN
  END;

END.
```

A3.13 Linear Filter G16P

```
($ COMPRESSCODE + $)
($ SUPERCOMPRESS + $)
($ CONFIGURATION CON8P $)

PROGRAM G;

(* Gaussian filter *)

GLOBAL TASK;

VAR     (* Images *)
        SOURCE,DEST             :ARRAY[0..255] OF INTEGER;

        (* Misc. *)
        IMSIZE,HEIGHT,WIDTH,
        NOPROCESSORS            :INTEGER;


PROCEDURE PROCESS(N:INTEGER;);

VAR P,K :INTEGER;

    BEGIN
    K:=N*IMSIZE;
    FOR P:=K/NOPROCESSORS TO (K+IMSIZE)/NOPROCESSORS-1 DO
        BEGIN
        DEST[P]:=(4*SOURCE[P]
                 +2*(SOURCE[P-WIDTH]+SOURCE[P-1]+SOURCE[P+1]
                                    +SOURCE[P+WIDTH])
                 +SOURCE[P-WIDTH-1]+SOURCE[P-WIDTH+1]
                 +SOURCE[P+WIDTH-1]+SOURCE[P+WIDTH+1]) / 16
        END;
    END;


BEGIN
(* Initialise job list *)
NOPROCESSORS:=8;
HEIGHT:=16;
WIDTH:=16;

IMSIZE:=HEIGHT*WIDTH;
END;



GENERAL TASK;

  BEGIN
  END;

END.
```

Appendix 4:

Descriptions of Low-Level Simulator Runs

This appendix lists each series of low-level simulator runs, with a short description of the hardware and software configurations used. Only deviations from the standard machine configuration (described in chapter 14) are listed here. Programs are listed in appendix 3, and system configuration files for the standard 2-repeated partitioned indirect binary 2-tube machine (CON1P and CON8P) and the 8-repeated partitioned indirect binary 2-tube machine (C101) are listed in appendix 5.

Run No.	Program	Task	Scheduler and System Software	Hardware Configuration
R100	B801B	Region filling (Scan-line)	Private list, pre-waiting, dynamic job re-allocation, improved WAIT algorithm.	Standard hardware configuration
R101	B802B	Region filling (Scan-line)	Private list, no pre-waiting, dynamic job re-allocation, improved WAIT algorithm.	Standard hardware configuration
R102	B803B	Region filling (Scan-line)	Global list, pre-waiting, dynamic job re-allocation, improved WAIT algorithm.	Standard hardware configuration
R103	B804B	Region filling (Scan-line)	Global list, no pre-waiting, dynamic job re-allocation, improved WAIT algorithm.	Standard hardware configuration
R104	B805B	Region filling (Wildfire)	Private list, pre-waiting, dynamic job re-allocation, improved WAIT algorithm.	Standard hardware configuration
R105	B806B	Region filling (Wildfire)	Private list, no pre-waiting, dynamic job re-allocation, improved WAIT algorithm.	Standard hardware configuration
R106	B807B	Region filling (Wildfire)	Global list, pre-waiting, dynamic job re-allocation, improved WAIT algorithm.	Standard hardware configuration
R107	B808B	Region filling (Wildfire)	Global list, no pre-waiting, dynamic job re-allocation, improved WAIT algorithm.	Standard hardware configuration
R108	Q601B	Integer sorting (Quicksort 64)	Private list, pre-waiting, dynamic job re-allocation, improved WAIT algorithm.	Standard hardware configuration
R109	Q602B	Integer sorting (Quicksort 64)	Private list, no pre-waiting, dynamic job re-allocation, improved WAIT algorithm.	Standard hardware configuration
R110	Q603B	Integer sorting (Quicksort 64)	Global list, pre-waiting, dynamic job re-allocation, improved WAIT algorithm.	Standard hardware configuration

Run No.	Program	Task	Scheduler and System Software	Hardware Configuration
R111	Q604B	Integer sorting (Quicksort 64)	Global list, no pre-waiting, dynamic job re-allocation, improved WAIT algorithm.	Standard hardware configuration
R112	Q801B	Integer sorting (Quicksort 256)	Private list, pre-waiting, dynamic job re-allocation, improved WAIT algorithm.	Standard hardware configuration
R113	Q802B	Integer sorting (Quicksort 256)	Private list, no pre-waiting, dynamic job re-allocation, improved WAIT algorithm.	Standard hardware configuration
R114	Q803B	Integer sorting (Quicksort 256)	Global list, pre-waiting, dynamic job re-allocation, improved WAIT algorithm.	Standard hardware configuration
R115	Q804B	Integer sorting (Quicksort 256)	Global list, no pre-waiting, dynamic job re-allocation, improved WAIT algorithm.	Standard hardware configuration
R116	G16P	Linear filtering (16×16 image)	No scheduler required	Standard hardware configuration
R117	G16P	Linear filtering (16×16 image)	No scheduler required	50ns network cycle time
R118	G16P	Linear filtering (16×16 image)	No scheduler required	150ns network cycle time
R119	G16P	Linear filtering (16×16 image)	No scheduler required	200ns network cycle time
R120	G16P	Linear filtering (16×16 image)	No scheduler required	250ns network cycle time
R121	G16P	Linear filtering (16×16 image)	No scheduler required	Unidirectional ring network
R122	G16P	Linear filtering (16×16 image)	No scheduler required	Toggling strategy for packet-routing clashes
R123	G16P	Linear filtering (16×16 image)	No scheduler required	No local instruction store in DPU
R124	G16P	Linear filtering (16×16 image)	No scheduler required	8-repeated 2-tube network (32 processors)
R125	G16P	Linear filtering (16×16 image)	No scheduler required	7-repeated 2-tube network (28 processors)
R126	G16P	Linear filtering (16×16 image)	No scheduler required	6-repeated 2-tube network (24 processors)

Run No.	Program	Task	Scheduler and System Software	Hardware Configuration
R127	G16P	Linear filtering (16 × 16 image)	No scheduler required	5-repeated 2-tube network (20 processors)
R128	G16P	Linear filtering (16 × 16 image)	No scheduler required	4-repeated 2-tube network (16 processors)
R129	G16P	Linear filtering (16 × 16 image)	No scheduler required	3-repeated 2-tube network (12 processors)
R130	G16P	Linear filtering (16 × 16 image)	No scheduler required	2-repeated 2-tube network (8 processors)
R131	G16P	Linear filtering (16 × 16 image)	No scheduler required	2-cube network (4 processors)
R132	G16P	Linear filtering (16 × 16 image)	No scheduler required	2-repeated 3-tube network (24 processors)
R133	G16P	Linear filtering (16 × 16 image)	No scheduler required	3-cube network (12 processors)
R134	G16P	Linear filtering (16 × 16 image)	Simple WAIT algorithm	Standard hardware configuration
R135	B801C	Region filling (Scan-line)	Private list, pre-waiting, dynamic job re-allocation, simple WAIT algorithm.	Standard hardware configuration
R136	B802C	Region filling (Scan-line)	Private list, no pre-waiting, dynamic job re-allocation, simple WAIT algorithm.	Standard hardware configuration
R137	B803C	Region filling (Scan-line)	Global list, pre-waiting, dynamic job re-allocation, simple WAIT algorithm.	Standard hardware configuration
R138	B804C	Region filling (Scan-line)	Global list, no pre-waiting, dynamic job re-allocation, simple WAIT algorithm.	Standard hardware configuration
R139	B805C	Region filling (Wildfire)	Private list, pre-waiting, dynamic job re-allocation, simple WAIT algorithm.	Standard hardware configuration
R140	B806C	Region filling (Wildfire)	Private list, no pre-waiting, dynamic job re-allocation, simple WAIT algorithm.	Standard hardware configuration
R141	B807C	Region filling (Wildfire)	Global list, pre-waiting, dynamic job re-allocation, simple WAIT algorithm.	Standard hardware configuration

Run No.	Program	Task	Scheduler and System Software	Hardware Configuration
R142	B808C	Region filling (Wildfire)	Global list, no pre-waiting, dynamic job re-allocation, simple WAIT algorithm.	Standard hardware configuration
R143	G16P	Linear filtering (16 × 16 image)	No scheduler required	4-cube network (32 processors)
R144	G16P	Linear filtering (12 × 12 image)	No scheduler required	Standard hardware configuration
R145	G16P	Linear filtering (8 × 8 image)	No scheduler required	Standard hardware configuration
R146	G16P	Linear filtering (16 × 16 image)	No scheduler required	No memory address translation.
R147	G16P	Linear filtering (16 × 16 image)	No scheduler required	Central horizontal split placement.
R148	G16P	Linear filtering (16 × 16 image)	No scheduler required	Central vertical split placement.
R149	G16P	Linear filtering (16 × 16 image)	No scheduler required	Layered vertical split placement.
R150	G16P	Linear filtering (16 × 16 image)	No scheduler required	Chessboard placement.

Appendix 5:

Simulated Machine Configurations

This appendix contains system configuration files which were used, in conjunction with the programs listed in appendix 3, to produce the results presented in the body of this thesis. The configuration file listed are:

CON1P – This file describes a machine based on a 2-repeated partitioned indirect binary 2-tube network. This machine has eight memory units placed in even-numbered positions, and one processor placed in slot 1. This configuration was used for most simulations involving only one processor (detailed in appendix 4).

CON8P – This file describes a similar machine to CON1P, but with eight memory units placed in even-numbered positions, and eight processors in the odd-numbered positions. This configuration was used for most simulations involving eight processors (detailed in appendix 4).

C101 – This file describes a machine based on an 8-repeated partitioned indirect binary 2-tube network. This machine has 32 memory units placed in even-numbered positions, and 32 processors in odd-numbered positions. This configuration was used for all simulations involving 32 processors (detailed in appendix 4).

A5.1 Configuration CON1P

```
Configuration file CON1P
========================

NOUNITS:=16

BOARDSIZE:=4096
TOPOLOGY:=0
BUSWIDTH:=4

BUSCONFLICTSYS:=3
NOVERALLOWED:=FALSE(0)
CODELOCAL:=TRUE(1)
HASHSYSTEM:=2

SLOTTIME:=100
CYCLETIME:=100
FETCHTIME:=200
MEMOPTIME:=500
MIRRTIME:=400
READTIME:=250
WRITETIME:=250

STACKTABADDR:=31000
DISPTABADDR:=31100
EXADDTABADDR:=31200

SYSVARPTR:=20000
GLOBVARPTR:=19000

MIRROR:=32000
READPORT:=32001
WRITEPORT:=32002

PROCSET:=[1,-1]
MEMSET:=[0,2,4,6,8,10,12,14,-1]

Default addresses

    Unit Execute   Stack Display
       0       0       0       0
       1       0   23000   31900
       2       0       0       0
       3       0   24000   31910
       4       0       0       0
       5       0   25000   31920
       6       0       0       0
       7       0   26000   31930
       8       0       0       0
       9       0   27000   31940
      10       0       0       0
      11       0   28000   31950
      12       0       0       0
      13       0   29000   31960
      14       0       0       0
      15       0   30000   31970

      -1      -1      -1      -1
```

A5.2 Configuration CON8P

```
Configuration file CON8P
========================

NOUNITS:=16

BOARDSIZE:=4096
TOPOLOGY:=0
BUSWIDTH:=4

BUSCONFLICTSYS:=3
HOVERALLOWED:=FALSE(0)
CODELOCAL:=TRUE(1)
HASHSYSTEM:=2

SLOTTIME:=100
CYCLETIME:=100
FETCHTIME:=200
MEMOPTIME:=500
MIRRTIME:=400
READTIME:=250
WRITETIME:=250

STACKTABADDR:=31000
DISPTABADDR:=31100
EXADDTABADDR:=31200

SYSVARPTR:=20000
GLOBVARPTR:=19000

MIRROR:=32000
READPORT:=32001
WRITEPORT:=32002

PROCSET:=[1,3,5,7,9,11,13,15,-1]
MEMSET:=[0,2,4,6,8,10,12,14,-1]

Default addresses
```

Unit	Execute	Stack	Display
0	0	0	0
1	0	23000	31900
2	0	0	0
3	0	24000	31910
4	0	0	0
5	0	25000	31920
6	0	0	0
7	0	26000	31930
8	0	0	0
9	0	27000	31940
10	0	0	0
11	0	28000	31950
12	0	0	0
13	0	29000	31960
14	0	0	0
15	0	30000	31970
-1	-1	-1	-1

A5.3 Configuration C101

```
Configuration file C_101
========================

NOUNITS:=64

BOARDSIZE:=1024
TOPOLOGY:=0
BUSWIDTH:=4

BUSCONFLICTSYS:=3
NOVERALLOWED:=FALSE(0)
CODELOCAL:=TRUE(1)
HASHSYSTEM:=2

SLOTTIME:=100
CYCLETIME:=100
FETCHTIME:=200
MEMOPTIME:=500
MIRRTIME:=400
READTIME:=250
WRITETIME:=250

STACKTABADDR:=31000
DISPTABADDR:=31100
EXADDTABADDR:=31200

SYSVARPTR:=10000
GLOBVARPTR:=9000

MIRROR:=32000
READPORT:=32001
WRITEPORT:=32002

PROCSET:=[1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31,
        33,35,37,39,41,43,45,47,49,51,53,55,57,59,61,63,-1]
MEMSET:=[0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,
        32,34,36,38,40,42,44,46,48,50,52,54,56,58,60,62,-1]
Default addresses
```

Unit	Execute	Stack	Display
0	0	0	0
1	0	10500	31500
2	0	0	0
3	0	11000	31510
4	0	0	0
5	0	11500	31520
6	0	0	0
7	0	12000	31530
8	0	0	0
9	0	12500	31540
10	0	0	0
11	0	13000	31550
12	0	0	0
13	0	13500	31560
14	0	0	0
15	0	14000	31570
16	0	0	0
17	0	14500	31580
18	0	0	0
19	0	15000	31590
20	0	0	0
21	0	15500	31600
22	0	0	0
23	0	16000	31610
24	0	0	0
25	0	16500	31620
26	0	0	0
27	0	17000	31630
28	0	0	0
29	0	17500	31640
30	0	0	0
31	0	18000	31650
32	0	0	0
33	0	18500	31660

34	0	0	0
35	0	19000	31670
36	0	0	0
37	0	19500	31680
38	0	0	0
39	0	20000	31690
40	0	0	0
41	0	20500	31700
42	0	0	0
43	0	21000	31710
44	0	0	0
45	0	21500	31720
46	0	0	0
47	0	22000	31730
48	0	0	0
49	0	22500	31740
50	0	0	0
51	0	23000	31750
52	0	0	0
53	0	23500	31760
54	0	0	0
55	0	24000	31770
56	0	0	0
57	0	24500	31780
58	0	0	0
59	0	25000	31790
60	0	0	0
61	0	25500	31800
62	0	0	0
63	0	26000	31810
-1	-1	-1	-1

Appendix 6:

Sample Result Files

This appendix contains a number of result files which were generated by the low-level simulator. Parts of these files were used to generate the graphs presented in the body of this thesis. The result files listed are:

R116|1/8 – This is a result file generated by simulation of the linear filtering program, G16P, which is listed in appendix 3. To generate this result file, this program was run using the system configuration file CON1P, listed in appendix 5, which defines a machine based on the 2-repeated partitioned indirect binary 2-tube network, with eight memory units and one processing unit.

R116|8/8 – This is also a result file generated by simulation of the linear filtering program, G16P. To generate this result file, the program was run using the system configuration file CON8P, also listed in appendix 5, which defines a similar machine, based on the 2-repeated partitioned indirect binary 2-tube network, but with eight memory units and eight processing units.

A6.1 Results of Run R116|1/8

```
Results of program #4:G16P1/8
Compiler date : 26-May-1986 V.1
Run date      : 29th May 1986


Processor status :

                                                NWFM / Total NWFM (%)                Efficiency (%)
 128598.  Slots   12859.8 us          Us   In   Sy   Su   Sd      Us   In   Sy   Su   Sd
      0  M   Idle 0  1   1664.0
      1 P  F 0 71  91918.0  220    99.6  0.1    .   0.2    .    71.5 76.1 47.4 62.1 69.5
      2  M   Idle 0  1   1672.0
      3
      4  M   Idle 0  0    642.0
      5
      6  M   Idle 0  2   2436.0
      7
      8  M   Idle 0  1    650.0
      9
     10  M   Idle 0  1   1678.0
     11
     12  M   Idle 0  1    671.0
     13
     14  M   Idle 0  1    656.0
     15

                          Means :  99.6  0.1    .   0.2    .    71.5 76.1 47.4 62.1 69.5

Overall : Proc :     71.5%  Mem :      1.0%  Bus :      1.5%
Counts  : Proc :  91918.0   Mem :  10069.0   Bus :  31252.0


Processor distribution (Elapsed time) :

    Name Strt  Fin       1     Mean

Active      0  220   100.0   100.0
Startup     0   19     0.1     0.1
Startwt    21   26       .       .
Initial    28   41     0.2     0.2
Wait       43   62       .       .
Signal     63   66       .       .
Read       67   72       .       .
Write      73   77       .       .
PROCESS    78  188    99.6    99.6
GLOBAL    189  200     0.1     0.1
TASK   0  201  209    99.6    99.6
Shutdown  211  220     0.1     0.1


Processor efficiency :

    Name Strt  Fin       1     Mean

Active      0  220    71.5    71.5
Startup     0   19    77.1    77.1
Startwt    21   26       -       -
Initial    28   41    63.4    63.4
Wait       43   62       -       -
Signal     63   66    47.4    47.4
Read       67   72       -       -
Write      73   77       -       -
PROCESS    78  188    71.5    71.5
GLOBAL    189  200    76.1    76.1
TASK   0  201  209    71.5    71.5
Shutdown  211  220    58.6    58.6
```

```
Shutdown Times :

Unit     Time
   1 128597.


Bus activity :

Bus clashes    :       0.0
Memory clashes :       0.0
Bus transfers :
  Unit   Transfers Mean time
    0 :     832.0       4.0
    1 :    4649.0       3.3
    2 :     836.0       4.0
    3 :       0.0         -
    4 :     321.0       3.0
    5 :       0.0         -
    6 :     834.0       3.0
    7 :       0.0         -
    8 :     325.0       2.0
    9 :       0.0         -
   10 :     839.0       2.0
   11 :       0.0         -
   12 :     334.0       5.0
   13 :       0.0         -
   14 :     328.0       5.0
   15 :       0.0         -
Total :    9298.0       3.4


Compiler status :

Compressed code -  Yes
Supercompressed -  Yes
Wait algorithm  -    3
Optimised DAG   -  Yes
Regs re-used    -  Yes
 saved in procs -   No
Code moved      -   No



Configuration : CON1P

Boardsize - 4096
Topology - 0
Width of bus - 4
Arbitration system - 3
Hovering -  No
Code local - Yes
Hashsystem - 2
Slot time (ns) - 100
Memory time (ns) - 200
Memory op time (ns) - 500
Mirror time (ns) - 400
Processor cycle time (ns) - 100
Memsize - 32767
No of procs  - 1
No of mems  - 8
Memory time (slots) - 2
Memory op time (slots) - 5
Mirror time (slots) - 4
Memory boards -   0  2  4  6  8 10 12 14
Processors    -   1
Machine       -  M P0 M - M - M - M - M - M - M -
```

A6.2 Results of Run R116|8/8

```
Results of program #4:G16P8/8
Compiler date : 26-May-1986 V.1
Run date      : 29th May 1986


Processor status :

                                                  NWFM / Total NWFM (%)                Efficiency (%)
 18255.0 Slots     1825.5 us                  Us    In    Sy    Su    Sd     Us    In    Sy    Su    Sd
      0  M   Idle 0 10   1792.0
      1 P  F 0 67  12300.0  318              96.3   0.9   0.2   2.3   0.3   68.2  74.5  46.6  54.9  67.2
      2  M   Idle 0  9   1712.0
      3 P  F 0 68  12353.0  318              95.9     .   2.8   0.9   0.3   68.4     -  74.8  62.2  67.2
      4  M   Idle 0  4    784.0
      5 P  F 0 68  12353.0  318              95.9     .   2.8   0.9   0.3   68.0     -  72.6  62.5  70.7
      6  M   Idle 0 14   2480.0
      7 P  F 0 68  12353.0  318              95.9     .   2.8   0.9   0.3   67.8     -  74.3  58.4  77.4
      8  M   Idle 0  4    818.0
      9 P  F 0 69  12519.0  318              94.7     .   4.1   0.9   0.3   68.7     -  79.7  61.2  73.2
     10  M   Idle 0  9   1718.0
     11 P  F 0 69  12519.0  318              94.7     .   4.1   0.9   0.3   68.5     -  75.6  58.1  67.2
     12  M   Idle 0  5    826.0
     13 P  F 0 69  12519.0  318              94.7     .   4.1   0.9   0.3   68.5     -  75.9  58.1  63.1
     14  M   Idle 0  4    770.0
     15 P  F 0 69  12519.0  318              94.7     .   4.1   0.9   0.3   68.4     -  75.0  61.8  69.5

                                    Means :   95.3   0.1   3.1   1.1   0.3   68.3  74.5  75.2  58.8  69.2

Overall : Proc :     68.1%  Mem :      7.5%  Bus :     12.0%
Counts  : Proc :  99435.0   Mem :  10900.0   Bus :  35040.0


Processor distribution (Elapsed time) :

     Name  Strt  Fin       1       3       5       7       9      11      13      15    Mean

Active        0  318   100.0   100.0   100.0   100.0   100.0   100.0   100.0   100.0   100.0
Startup       0   19     0.6     0.7     0.6     0.7     0.6     0.6     0.6     0.6     0.6
Startwt      21   26       .     2.6     2.7     2.7     3.6     3.9     3.8     3.8     2.9
Initial      28   69     3.0       .       .       .       .       .       .       .     0.4
Wait         71   90       .     2.3     2.4     2.3     3.2     3.4     3.4     3.4     2.6
Signal       91   94     0.3     0.3     0.3     0.3     0.3     0.3     0.3     0.3     0.3
Read         95  100       .       .       .       .       .       .       .       .       .
Write       101  105       .       .       .       .       .       .       .       .       .
PROCESS     106  216    95.4    95.7    95.7    95.7    94.7    94.5    94.5    94.5    95.1
GLOBAL      217  228     0.8       .       .       .       .       .       .       .     0.1
TASK   0    229  237    95.8       .       .       .       .       .       .       .    12.0
TASK   1    239  247       .    96.1       .       .       .       .       .       .    11.9
TASK   2    249  257       .       .    96.0       .       .       .       .       .    12.0
TASK   3    259  267       .       .       .    96.1       .       .       .       .    12.0
TASK   4    269  277       .       .       .       .    95.1       .       .       .    11.9
TASK   5    279  287       .       .       .       .       .    94.9       .       .    11.9
TASK   6    289  297       .       .       .       .       .       .    94.8       .    11.9
TASK   7    299  307       .       .       .       .       .       .       .    94.9    11.9
Shutdown     309  318     0.7     0.6     0.6     0.6     0.6     0.6     0.7     0.7     0.6
```

Processor efficiency :

Name	Strt	Fin	1	3	5	7	9	11	13	15	Mean
Active	0	318	67.8	68.5	68.1	67.9	69.0	68.6	68.7	68.6	68.4
Startup	0	19	74.3	66.4	69.8	65.9	69.2	69.2	68.6	73.0	69.5
Startwt	21	26	-	74.8	71.4	72.0	78.8	73.6	74.2	74.5	74.3
Initial	28	69	56.1	-	-	-	-	-	-	-	56.1
Wait	71	90	-	78.0	74.8	76.6	82.5	77.6	78.3	78.0	78.2
Signal	91	94	46.6	50.0	54.0	55.1	49.1	50.9	49.1	44.3	49.7
Read	95	100	-	-	-	-	-	-	-	-	-
Write	101	105	-	-	-	-	-	-	-	-	-
PROCESS	106	216	68.2	68.4	68.0	67.9	68.8	68.5	68.6	68.5	68.3
GLOBAL	217	228	74.5	-	-	-	-	-	-	-	74.5
TASK 0	229	237	68.2	-	-	-	-	-	-	-	68.2
TASK 1	239	247	-	68.4	-	-	-	-	-	-	68.4
TASK 2	249	257	-	-	68.0	-	-	-	-	-	68.0
TASK 3	259	267	-	-	-	67.8	-	-	-	-	67.8
TASK 4	269	277	-	-	-	-	68.7	-	-	-	68.7
TASK 5	279	287	-	-	-	-	-	68.5	-	-	68.5
TASK 6	289	297	-	-	-	-	-	-	68.5	-	68.5
TASK 7	299	307	-	-	-	-	-	-	-	68.4	68.4
Shutdown	309	318	57.1	59.1	63.0	66.7	61.3	59.6	56.7	56.7	59.8

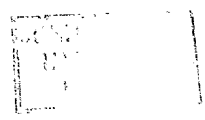
Shutdown Times :

Unit	Time
1	18154.0
3	18039.0
5	18159.0
7	18193.0
9	18138.0
11	18254.0
13	18236.0
15	18248.0

Bus activity :

Bus clashes : 983.0
Memory clashes : 335.0
Bus transfers :

Unit	Transfers	Mean time
0 :	866.0	3.2
1 :	631.0	3.7
2 :	856.0	3.1
3 :	617.0	3.4
4 :	362.0	3.5
5 :	617.0	4.1
6 :	856.0	3.5
7 :	617.0	4.1
8 :	379.0	3.1
9 :	619.0	3.5
10 :	859.0	3.1
11 :	619.0	3.5
12 :	395.0	3.6
13 :	619.0	4.0
14 :	385.0	3.7
15 :	619.0	4.0
Total :	9916.0	3.5



Compiler status :

Compressed code - Yes
Supercompressed - Yes
Wait algorithm - 3
Optimised DAG - Yes
Regs re-used - Yes
saved in procs - No
Code moved - No

```
Configuration : CON8P

Boardsize - 4096
Topology - 0
Width of bus - 4
Arbitration system - 3
Hovering -  No
Code local - Yes
Hashsystem - 2
Slot time (ns) - 100
Memory time (ns) - 200
Memory op time (ns) - 500
Mirror time (ns) - 400
Processor cycle time (ns) - 100
Memsize - 32767
No of procs  - 8
No of mems  - 8
Memory time (slots) - 2
Memory op time (slots) - 5
Mirror time (slots) - 4
Memory boards -   0  2  4  6  8 10 12 14
Processors    -   1  3  5  7  9 11 13 15
Machine       -  M P0 M P1 M P2 M P3 M P4 M P5 M P6 M P7
```

1