An Investigation into the Parallel Implementation of JPEG for Image Compression

by

Paul Darbyshire

Thesis submitted for the degree of

Master of Engineering

in the Department of

Electrical and Electronic Engineering

Victoria University of Technology

Declaration of Candidate:

I

Paul Darbyshire

declare that this thesis entitled

An Investigation into the Parallel Implementation of JPEG for Image Compression

is my own work and has not been submitted previously, in whole or in part in respect of any other academic award.

Signature of Candidate

18/4/98

Date

ACKNOWLEDGEMENTS

This thesis has been a long time in the making, and during that time I have had help from many people, to whom I owe much. First, to my original principal supervisor, Ann Pleasants, who became my second supervisor after a career move to Fiji. Thanks to Ann's help and persistence over the years, and particularly in the last three months via constant emails to and from Fiji, this thesis has finally been assembled in a presentable form. I would also like to thank Ji Dong Wang who took over the role of principal supervisor for a period of about a year and showed me how to write papers. I would also like to thank Elizabeth Haywood who has been my principal supervisor for only a short while but who has helped greatly with content suggestions and editing. This thesis would not have been possible without them.

Finally, I have saved this to last, but perhaps it should have been first. To my wife Kay and son Adam, who have both put up with me for the duration of this thesis. We have missed much together over the time that I have spent on this. To Adam, who always wanted to do something with me, but would quietly shut the door and let dad work, thanks. To Kay who would not complain when we missed doing things together so I could get this done, thanks.

I dedicate this thesis to both of you.

An Investigation into the Parallel Implementation of JPEG for Image Compression

Abstract

This research develops a parallel algorithm to implement the JPEG standard for continuous tone still picture compression to be run on a group of transputers. The processor-farm paradigm is adopted. This research shows this to be the best paradigm for use with the JPEG baseline algorithm on the measured component times within the algorithm. The speedup of the parallel algorithm is investigated and measured against a single processor version. An optimal distribution of JPEG components on the processors within the processor farm is established.

The research focuses on the investigation of the optimal number of processors, which can be used effectively for a JPEG implementation adopting the processor-farm paradigm. This optimal number is termed the saturation point. Once the saturation point has been reached, it is shown that the parallel algorithm's speedup cannot be improved without the redistribution of tasks in the farm, regardless of how many extra processors are used. Further distributions of processing tasks are investigated with the aim of extending the saturation point. It is shown that the saturation point can be extended, and the distributions of tasks to achieve this are demonstrated. It is also shown that while the saturation point can be increased, the gains are minimal and may not be worth the cost of the extra processor. In fact, the algorithm speedup diminishes after the addition of the third processor, up to saturation point.

A simulation algorithm is devised using Java, which takes advantage of the multi-threaded nature of the language. A technique is developed for simulating the processor-farm paradigm. This technique uses the concept of the Java threadgroup as a basis for a simulated processor, and a Java thread allocated to that group, as a process belonging to this processor. A process scheduling scheme is refined which allows the simulated parallel system to be monitored over simulated scheduling rounds. A scheme is also shown that simulates the message passing of the transputer.

This simulated system allows the investigation of the saturation point regardless of the number of processors physically available. Further data on the saturation point supports the hypothesis that the saturation point is around seven processors. The hypothesis is based on the extrapolation of the results obtained using a limited number of processors. Using the simulation, the behaviour of a parallel system can be observed with an arbitrary number of processors. Since this simulation is written in Java, it is also platform independent, and defines an algorithm suitable for a distributed system.

CONTENTS

CHAPTER 1 INTRODUCTION **1**

CHAPTER 2 A SURVEY OF IMAGE COMPRESSION TECHNIQUES **6**

2.1 INTRODUCTION 6
2.2 IMAGE REPRESENTATION 7
2.3 ENTROPY 9
2.3.1 Measuring Entropy *10*
2.4 ENTROPY CODERS 12
2.4.1 Huffman Coding *12*
2.4.2 Arithmetic Coding *15*
2.5 IMAGE COMPRESSION 17
2.5.1 Predictive Coding *18*
2.5.2 Transform Coding *21*
2.5.2.1 Fourier Transform 24
2.5.2.2 Discrete Cosine Transform 26
2.5.2.3 Karhunen-Loeve Transform 27
2.5.2.4 Walsh-Hadamard Transform 28
2.5.2.5 Summary 29
2.5.3 Vector Quantization *32*
2.6 SELECTION OF INTERNATIONAL STANDARD 34
2.7 RECENT COMPRESSION TECHNIQUES 35
2.7.1 Wavelet Compression *35*
2.7.2 Fractal Compression *39*
2.7.3 Comparison with JPEG *43*
2.8 SUMMARY 45

CHAPTER 3 THE JPEG STANDARD AND PARALLEL SYSTEMS 46

3.1 INTRODUCTION 46

3.2 JPEG STANDARD 46

3.2.1 Lossless Vs. Lossy Compression 47

3.2.2 DCT-based Compression Overview 49

3.2.3 JPEG Steps 54

3.2.3.1 Level Shift 55

3.2.3.2 DCT 55

3.2.3.3 Quantization 57

3.2.3.4 Baseline Coder 58

3.2.4 Compressed Data Formats 62

3.3 PARALLEL SYSTEMS AND PROCESSING 62

3.3.1 Supporting Architecture 65

3.3.1.1 MIMD vs. SIMD 65

3.3.1.2 Memory Organization 67

3.3.1.3 The Transputer 69

3.3.2 Parallel Processing Software Paradigms 72

3.3.2.1 Homogeneous Parallelization 73

3.3.2.2 Heterogeneous Parallelization 74

3.3.2.3 Processor Farm Paradigm 75

3.3.3 Performance 77

3.4 SUMMARY 82

CHAPTER 4 PARALLEL JPEG IMPLEMENTATION 83

4.1 INTRODUCTION 83

4.2 METHODOLOGY 84

4.3 PARALLEL PROCESSING ENVIRONMENT 85

4.4 SEQUENTIAL ALGORITHM 87

4.5 PARALLEL ALGORITHM 90

4.6 COMPONENT TIMING ISSUES AND COMMUNICATION 96

4.6.1 Component Timing 96

4.6.1.1 Hardware Issues 96

4.6.1.2 Software Issues 98

4.6.1.3 Expected Time Savings of PV1 99

4.6.2 Communication Timing 100

4.6.2.1 Expected Time Savings of PV1 Recalculated 105

4.6.2.2 Impact of Communication Times on Optimal Placement 106

4.7 DEVELOPMENT OF A GENERAL PARALLEL ALGORITHM 108

4.7.1 Pipeline Implementation 108

4.7.2 Processor Farm Implementation *110*
4.7.3 Development of Processor Farm Algorithm *111*
4.7.3.1 Saturation Point 114
4.7.3.2 Expected vs Measured times of PV2 118
4.7.3.3 Impact of Saturation Point on Optimal Task Placement 121
4.8 CONCLUSION 124

CHAPTER 5 PARALLEL JPEG SIMULATION USING JAVA 126

5.1 INTRODUCTION 126
5.2 IMPLEMENTATION OF TASKS AS JAVA THREADS 127
5.3 THREAD SCHEDULING IN JAVA 129
5.3.1 Thread states in Java *129*
5.3.2 Scheduling in the Java VM *131*
5.4 DEVELOPMENT OF PARALLEL JPEG SIMULATION ALGORITHM 133
5.4.1 Thread Groups *135*
5.4.2 Construction of Simulation Objects *137*
5.4.3 Simulation Algorithm *139*
5.4.4 Processor Communication *142*
5.5 JAVA SIMULATION PROBLEMS 144
5.5.1 Garbage Collector *144*
5.5.2 Inappropriate Behaviour of Master Processor Send Thread *145*
5.5.3 Network Timing Considerations *146*
5.5.4 Processor Speed When Running Simulation *149*
5.6 SIMULATION RESULTS 151
5.7 CONCLUSIONS 159

CHAPTER 6 CONCLUSION 161

6.1 SUMMARY 161
6.2 CRITICAL APPRAISAL 162
6.3 FURTHER RESEARCH 164

BIBLIOGRAPHY 168

APPENDIX A IMAGES AND TABLES 177

A.1 INTRODUCTION 177

A.2 IMAGES 178
A.3 CHAPTER 4 TABLES 179
A.4 JPEG PROCEDURE DIAGRAMS 182
A.5 SIMULATION RUN DATA 186

APPENDIX B PARALLEL C TRANSPUTER CODE 190

B.1 INTRODUCTION 190
B.2 CONFIGURATION FILE FOR SEQUENTIAL PROGRAM SV1 191
B.3 PARALLEL C CODE OF PROGRAM SV1 192
B.4 CONFIGURATION FILE FOR PARALLEL PROGRAM PV1 198
B.5 PARALLEL C CODE OF PROGRAM PV1 199
B.5.1 Code for Processor P_0 199
B.5.2 Code for Processor P_1 204
B.6 CONFIGURATION FILE FOR PROCESSOR FARM PROGRAM PV2 206
B.7 PARALLEL C CODE OF PROCESSOR FARM PROGRAM PV2 207
B.7.1 Code for Master task 207
B.7.2 Code for Worker task 212
B.8 INCLUDE FILES 214

APPENDIX C JAVA PARALLEL SIMULATION CODE 216

C.1 INTRODUCTION 216
C.2 IMAGE FILE HEADER CLASS 217
C.3 COMPRESSED FILE HEADER CLASS 218
C.4 TOKEN OBJECT CLASS 219
C.5 WORKER THREAD OBJECT CLASS 220
C.6 SEND THREAD OBJECT CLASS OF MASTER PROCESSOR 223
C.7 RECEIVE THREAD OBJECT CLASS OF MASTER PROCESSOR 225
C.8 CONTROLLING THREAD AND SIMULATION ALGORITHM CLASS 228

List of Figures

FIGURE 2.1 CONSTRUCTION OF DIGITAL IMAGE 8
FIGURE 2.2 DEPICTION OF ENTROPY VS. REDUNDANCY IN A DATA SOURCE [90] 10
FIGURE 2.3 HUFFMAN TREE BUILT FROM FREQUENCIES IN TABLE 2.2 13
FIGURE 2.4 INITIAL INTERVAL DIVISIONS ACCORDING TO PROBABILITIES 15
FIGURE 2.5 SUBSEQUENT DIVISION OF PREVIOUS SYMBOL INTERVALS 16
FIGURE 2.6 PIXEL NEIGHBOURHOOD 18
FIGURE 2.7 DPCM LOSSY ENCODER BLOCK DIAGRAM 20
FIGURE 2.8 GENERAL STEPS IN TRANSFORM CODING 21
FIGURE 2.9 TYPICAL I/O CHARACTERISTICS OF A QUANTIZER [48] 22
FIGURE 2.10 ENERGY PACKING EFFICIENCY AS A FUNCTION OF CORRELATION COEFFICIENT [15] 30
FIGURE 2.11 ENERGY PACKING EFFICIENCY AS A FUNCTION OF TRANSFORM BLOCK SIZE [15] 30
FIGURE 2.12 DECORRELATION EFFICIENCY AS A FUNCTION OF THE CORRELATION COEFFICIENT [15] ... 31
FIGURE 2.13 DECORRELATION EFFICIENCY AS A FUNCTION OF TRANSFORM BLOCK SIZE [15] 31
FIGURE 2.14 VECTOR QUANTIZATION BLOCK DIAGRAM 33
FIGURE 2.15 COMPARISON OF SINE WAVE AND DAUBECHIES WAVELET 36
FIGURE 2.16 SOME WELL KNOWN WAVELET FAMILIES 36
FIGURE 2.17 DISCRETE FOURIER BASIS FUNCTIONS VS. SCALED WAVELET BASIS FUNCTIONS 37
FIGURE 2.18 OUTLIERS AND TRENDS COEFFICIENTS 38
FIGURE 2.19 FRACTAL GENERATED FERN LEAF 40
FIGURE 2.20 SIERPINSKY'S TRIANGLE CONSTRUCTION 40
FIGURE 2.21 BARNSLEY'S PHOTOCOPY MACHINE 41

FIGURE 3.1 STRUCTURE OF LOSSLESS ENCODER 47
FIGURE 3.2 PREDICTION OF PIXEL *X* FROM 3 NEIGHBOURS 48
FIGURE 3.3 GENERAL STRUCTURE OF DCT BASED ENCODER 48
FIGURE 3.4 IMAGE BLOCKS 49
FIGURE 3.5 APPLICATION OF ONE-DIMENSIONAL DCT 50
FIGURE 3.6 APPLICATION OF A TWO-DIMENSIONAL DCT 51
FIGURE 3.7 TRANSFORMED BLOCK COEFFICIENTS 52
FIGURE 3.8 TWO-DIMENSIONAL 8×8 DCT BASIS FUNCTIONS [70] 52
FIGURE 3.9 ZIG-ZAG SEQUENCE FOR 8×8 BLOCK OF TRANSFORM COEFFICIENTS 53
FIGURE 3.10 AC COEFFICIENT PROBABILITY IN ZIG-ZAG INDEX [69] 54
FIGURE 3.11 SIGNAL FLOW GRAPH FOR FAST DCT [14], [70] 56
FIGURE 3.12 BASELINE SEQUENTIAL-DCT ENCODER MODEL 59
FIGURE 3.13 DPCM MODEL FOR DC COEFFICIENT ENCODING 59

FIGURE 3.14 DISTRIBUTION OF PARALLEL SYSTEMS AMONG APPLICATIONS 64
FIGURE 3.15 SOME DISTRIBUTED MEMORY ARCHITECTURES 68
FIGURE 3.16 SOME SHARED MEMORY ARCHITECTURES 68
FIGURE 3.17 GENERALIZED DIAGRAM OF A TRANSPUTER 70
FIGURE 3.18 GENERAL FORM OF TRANSPUTER SYSTEM CONNECTED TO PC HOST 71
FIGURE 3.19 SOME EXAMPLES OF TRANSPUTER TOPOLOGIES 71
FIGURE 3.20 HOMOGENEOUS PARALLELIZATION [10] 73
FIGURE 3.21 HETEROGENEOUS PARALLELIZATION [10] 74
FIGURE 3.22 PROCESSOR FARM METHODOLOGY 76
FIGURE 3.23 AMDAHL'S LAW VS. GUSTAFSON-BARSIS LAW 79
FIGURE 3.24 SPEEDUP CURVE FOR MANDELBROT DATA FROM TABLE 3.6 80
FIGURE 3.25 SIMPLIFIED REPRESENTATION OF H.261 ENCODING EXECUTION 80
FIGURE 3.26 IDEALIZED VS. ACTUAL PERFORMANCE FOR THE H.261 ENCODER [92] 81

FIGURE 4.1 PHYSICAL TRANSPUTER CONFIGURATION WITH THREE PROCESSORS 86
FIGURE 4.2 SEQUENTIAL ALGORITHM SV1 MAIN COMPONENTS 87
FIGURE 4.3 AVERAGE PROCESSING TIMES PER IMAGE BLOCK 90
FIGURE 4.4 TOTAL PROCESSING TIMES PER PROCESSOR 91
FIGURE 4.5 STRUCTURE OF PARALLEL ALGORITHM PV1 94
FIGURE 4.6 COMPARATIVE CHART REPRESENTATION OF TABLE 4.5 97
FIGURE 4.7 COMPARISON OF COMMUNICATION TIMING BETWEEN DIFFERENT PROCESSORS 101
FIGURE 4.8 INTER-PROCESSOR COMMUNICATION MODELS 102
FIGURE 4.9 SUM OF PROCESSING TIMES WITH COMMUNICATION ALLOWANCE 104
FIGURE 4.10 PIPELINE IMPLEMENTATION OF SV1 WITH 3 PROCESSORS 109
FIGURE 4.11 CALCULATED PROCESSING TIMES OF PROCESSORS IN PIPELINE 109
FIGURE 4.12 APPLICATION OF PROCESSOR FARM TECHNIQUE TO IMAGE COMPRESSION 110
FIGURE 4.13 STRUCTURE OF PARALLEL ALGORITHM PV2 114
FIGURE 4.14 COMPARISON OF OVERALL PROCESSING TIMES OF P_0 VS P_1 116
FIGURE 4.15 EXPECTED IDLE TIME ON P_0 117
FIGURE 4.16 ASSUMED CONFIGURATION OF PROCESSOR FARM 121
FIGURE 4.17 COMPARISON OF SATURATION POINTS WITH DIFFERENT TASK CONFIGURATIONS 122
FIGURE 4.18 PROGRESSIVE ALGORITHM TIME SAVINGS VS NUMBER OF PROCESSORS 123

FIGURE 5.1 JAVA THREAD STRUCTURE 128
FIGURE 5.2 JAVA THREAD STATES 130
FIGURE 5.3 JAVA THREAD SCHEDULING HIERARCHY 132
FIGURE 5.4 ABSTRACT VIEW OF A MULTI-PROCESSOR SYSTEM 134
FIGURE 5.5 THREADS ASSIGNED TO THREAD GROUPS 135

FIGURE 5.6 OPERATIONS ON THREAD GROUPS 136
FIGURE 5.7 ALGORITHM PSEUDOCODE FOR CREATION OF SIMULATION OBJECTS 138
FIGURE 5.8 STRUCTURE OF OBJECTS IN JAVA SIMULATION 139
FIGURE 5.9 PSEUDOCODE FOR SIMULATION ALGORITHM SIM2 142
FIGURE 5.10 USE OF JAVA VECTORS TO SIMULATE MESSAGE PASSING 143
FIGURE 5.11 DELAY TIMES TO SIMULATE NETWORK COMMUNICATION DELAYS 148
FIGURE 5.12 ASSUMED SIMULATION NETWORK TOPOLOGY 148
FIGURE 5.13 COMPARATIVE CHART REPRESENTATION OF TABLE 5.1 150
FIGURE 5.14 COMPARATIVE LINE CHART REPRESENTATION OF TABLE 5.2 153
FIGURE 5.15 COMPARATIVE LINE CHART REPRESENTATION OF TABLE 5.3 156
FIGURE 5.16 COMPARATIVE LINE CHART REPRESENTATION OF TABLE 5.4 156
FIGURE 5.17 COMPARISON LINE CHARTS OF TRENDS IN INPUT QUEUE LENGTH 157

FIGURE 6.1 POSSIBLE USE OF MULTIPLE PROCESSOR FARMS 165

FIGURE A.1 GENERATION OF TABLE OF HUFFMAN CODE SIZES 182
FIGURE A.2 GENERATION OF TABLE OF HUFFMAN CODES 183
FIGURE A.3 SEQUENTIAL ENCODING OF AC COEFFICIENTS WITH HUFFMAN CODING 184
FIGURE A.4 SEQUENTIAL ENCODING OF A NON-ZERO AC COEFFICIENT 185

List of Tables

TABLE 2.1 8-BIT GREY SCALE VALUES FOR BLOCK IN FIGURE 2.1 8
TABLE 2.2 SOURCE SYMBOL FREQUENCY DISTRIBUTION 13
TABLE 2.3 SOURCE SYMBOLS AND PROBABILITIES 15
TABLE 2.4 COMPARISON OF COMPRESSION METHODOLOGIES 44
TABLE 2.5 COMPARISON OF ARTIFACT BEHAVIOUR OF METHODOLOGIES 44

TABLE 3.1 LUMINANCE QUANTIZATION TABLE 58
TABLE 3.2 LIST OF LUMINANCE CODE LENGTHS FOR HUFFMAN TABLES 60
TABLE 3.3 SET OF LUMINANCE CODE VALUES FOR CORRESPONDING CODE LENGTHS IN TABLE 3.2 60
TABLE 3.4 AMPLITUDE CATEGORIES FOR COEFFICIENTS 61
TABLE 3.5 FLYNN'S PARALLEL SYSTEMS CLASSIFICATION SCHEME 65
TABLE 3.6 EXECUTION TIMES FOR DISPLAYING MANDELBROT SET 79

TABLE 4.1 OVERALL TIMING FOR ALGORITHM SV1 89
TABLE 4.2 TIME TRIAL AVERAGES FOR SV1 COMPONENTS 89
TABLE 4.3 PROCESSING TASK SYMBOLS 93
TABLE 4.4 OVERALL TIMING RESULTS FOR PV1 95
TABLE 4.5 COMPARISON TIMING DATA PER BLOCK OF NON I/O PROCESSES ON ALL PROCESSORS 97
TABLE 4.6 RATIOS OF COMPONENT PROCESSING TIMES OF P_1 / P_0 99
TABLE 4.7 COMMUNICATION TIMING SUMMARY BETWEEN P_0 AND P_1 , P_2 101
TABLE 4.8 PROCESSOR INVOLVEMENT IN COMMUNICATION 103
TABLE 4.9 OVERALL TIMING RESULTS FOR PV2 120

TABLE 5.1 COMPARISON OF TRANSPUTER AND PENTIUM PROCESSOR TIMES 150
TABLE 5.2 OVERALL AVERAGES OF THE FIRST 100 TIME QUANTUMS 152
TABLE 5.3 AVERAGES OF FIRST 100 TIME QUANTUMS UP TO τ_{snd} DEATH 155
TABLE 5.4 AVERAGES OF FIRST 100 TIME QUANTUMS AFTER τ_{snd} DEATH 155

TABLE A.1 TIMING DATA FOR ALGORITHM SV1 COMPONENTS 179
TABLE A.2 TIMING OF SV1 NON I/O COMPONENTS ON PROCESSORS P1 AND P2 180
TABLE A.3 TRANSMISSION & RECEIPT TIMES OF 4096 BLOCKS (1 BYTE/ELEMENT) 180
TABLE A.4 TRANSMISSION & RECEIPT TIMES OF 4096 BLOCKS (4 BYTES/ELEMENT) 181
TABLE A.5 TRANSMISSION & RECEIPT TIMES OF 4096 BLOCKS (8 BYTES/ELEMENT) 181

TABLE A.6 TRANSMISSION & RECEIPT TIMES OF 4096 BLOCKS (4/8 BYTES/ELEMENT) 181
TABLE A.7 TIMING TRIALS FOR JPEG COMPRESSION COMPONENTS 186
TABLE A.8 SAMPLE DATA FOR TRIAL 1 OF SIMULATION OF 7 WORKER PROCESSORS. 187

CHAPTER 1

INTRODUCTION

The use of digital images dates back to the early 1920s when digitized pictures of world news events were first transmitted by submarine cable between New York and London [28]. It was not until the mid to late 1960s that the use of digital images became more wide spread. Then the computing power, speed and storage capacity in third generation computers were such that practical applications involving the use of digital images became feasible, for example, the storage and transmission of satellite images.

It soon became apparent that in dealing with images, computer systems would have to cope with huge amounts of data. Image compression was recognized as an important problem. For example, a small image of 640 × 480 pixels using 8 bit VGA colour mode requires about 2.4×10^6 bits. If 24-bit true colour is used then the image requires three times that amount. Digital images from Landsat satellites are taken using multispectral scanners in four spectral bands, and some images are in excess of 300 megabytes. It is not only professionals in these specialist fields that are affected by image size. The average Internet user today is well aware of the time it takes to download the large graphics embedded in web pages.

In 1992, the JPEG standard was adopted as the International Standard for continuous tone still picture image compression. The driving force behind JPEG was to deliver a standard for the coded representation of compressed image data, for interchange between telecommunication applications. The JPEG standard provides specifications for multiple modes of operation, but the most popular mode, allowing the most compression, is the expanded lossy DCT-based mode of operation. This mode of

operation loses some information that is not detectable by the human eye thus allowing far higher compressions. It is based on the Discrete Cosine Transform (DCT). By using the DCT, the image is first transformed into its frequency domain resulting in less correlated coefficients that make the quantization and coding more effective. JPEG has almost become a household word, with users of the internet familiar with the term due to the 'jpg' format images available on the web. Most image manipulation software tools also support the JPEG format, which has become a format of choice for people providing large images via the Internet.

The DCT is a computationally intensive task requiring a reasonable amount of computing power in order to code images in an acceptable time. For this reason, many implementations of JPEG are realized through hardware solutions, or JPEG chips, to achieve higher speed. However, as more demands are placed for faster implementations for real time applications, traditional serial processors have begun to encounter physical limitations which inhibit further speed increases [40]. For example, no signal can travel faster than the speed of light, which indicates an upper limit to the speed at which algorithms can be processed. To overcome this, components are being made smaller which limits the distance that signals are transmitted during processing, but then heat dissipation becomes a problem.

One strategy used to overcome these limits is parallel processing. In a parallel system, multiple processors are connected and the processing is distributed to the processors so the various tasks can be executed simultaneously. The aim of parallel processing is that if a problem takes T time units to process on a single processor system, then on an N processor system it will take T/N time units. This theoretical speedup is of course never reached due to practical considerations like communication delays between processors, and the overhead associated with the overall management of the network of processors. However, considerable speed advantages can be gained by application of a parallel programming paradigm to a suitable problem.

There are a number of parallel programming paradigms, and image compression is a good candidate for parallelization. In fact, image coding, particularly using the JPEG standard is well suited for placement on a multiprocessor system, since an appropriate paradigm can take advantage of the data parallelization inherent in the image as well

as any possible algorithmic parallelization. With the transform coding approach of JPEG, the image is broken into fixed sized sub-sections called blocks, and these are processed independently of each other. It is this independent processing of the blocks that suggest a possible data parallel approach. An algorithm parallelization is also possible, as JPEG involves a number of distinct steps that must be applied to each block.

The purpose of this research is to investigate the application of parallel programming techniques to the JPEG international standard using a suitable parallel programming paradigm. This research initially uses a transputer based network of processors for the hardware platform. A single processor algorithm, implementing the JPEG baseline sequential mode is constructed for use as a benchmark. A simple parallel version using two processors is then developed for initial testing, and finally a general parallel algorithm implementing the processor-farm paradigm is constructed.

Run time data is then gathered, and an optimal placement of processor tasks is shown for the network of processors. From the data obtainable, an extrapolation of speedup calculations indicated unexpected results, and from this, a hypothesis was formed that indicated a limit to the number of processors that could be effectively used by the processor-farm paradigm implementing JPEG. This limit on the number of processors is called the *saturation point*. This number was initially calculated at 7 processors. Further, a hypothesis is made that indicates this number may be extended by redistribution of tasks among the processors once the initial saturation point has been reached.

In order to test these hypotheses a simulation algorithm is constructed which runs on a Pentium 200 MHz processor. The simulation algorithm is implemented using Java and takes advantage of the multi-threaded nature of the language. A technique for developing a simulation of the processor-farm paradigm is shown. This technique uses the concept of the Java threadgroup as a basis for a simulated processor, and a Java thread allocated to that group as a process belonging to this processor. A process scheduling scheme is devised which allows the simulated parallel system to be monitored over simulated scheduling rounds. A scheme is also shown which simulates the message passing of the transputer.

This simulation allows the behaviour of the processor-farm paradigm to be observed when more processors are added, regardless of the number of physical processors available. The simulation is then used to obtain further results on the parallel JPEG implementation to test the hypothesis. The results collected support the original hypothesis concerning the saturation point and the estimate for its number at 7 processors. Further, it is shown with data collected from the simulation, that once reached, the saturation point can be extended by redistribution of the tasks among the processors. Optimal configurations for this extension are shown.

The Java simulation is platform independent due to the nature of Java, and thus capable of running on any platform which provides a Java Virtual Machine (VM). This results in another outcome of this research, being an algorithm, which is suitable for a distributed system.

A number of papers resulted from this research. These are:

- "Parallel Implementation of the JPEG Still Picture Compression Algorithm", Darbyshire, Pleasants and Wang [21], presented at the Australian Pattern Recognition Society Student Conference in 1996.
- "Using Java to Simulate Multi-processor Systems", Darbyshire P [20], in print with Department of Information Systems working paper series, Victoria University of Technology.
- "Implementation of a Parallel Image Compression Algorithm Using Java", Darbyshire and Wang [22], presented at the 1997 DICTA conference at Massey University, Auckland as a Technical Keynote Presentation.

Chapter 2 of this thesis investigates the development of image compression techniques over the last 30 years. This culminates in the adoption of the JPEG standard, as the international standard for still picture compression. Research has continued since, and two of the more promising recent developments are highlighted.

Chapter 3 provides an in depth look at the JPEG standard, and covers the operation of the baseline sequential mode of operation. The DCT technique, which is at the heart of the JPEG algorithms, is discussed and a fast version of this is presented. The JPEG algorithms highlighted in this chapter are used in the construction of the research parallel algorithms. This chapter also investigates the development of parallel architectures, and the parallel paradigms commonly used to build software for multi-processor systems. The selection of a suitable paradigm for application to JPEG is discussed.

In Chapter 4, a parallel implementation of JPEG is constructed in two different stages, and data collected from trial runs of these algorithms are compared with those of a JPEG algorithm constructed for a single processor system. Using the data obtained, the parallel system overhead is investigated and the results used to develop an optimal placement of tasks on processors, which maximizes parallelism. The concept of a limit to the number of processors that can be effectively used is investigated.

In Chapter 5, a simulation program that implements the JPEG parallel algorithm of Chapter 4 is constructed in the JAVA programming language. This simulation program exploits the multi-threaded nature of JAVA and allows the behaviour of the parallel algorithm constructed in Chapter 4 to be investigated with a varying number of processors. By using JAVA, this simulation is platform independent and can be run on any system providing a JAVA VM. The concept of the processor limit developed in Chapter 4 is further investigated using this simulation program.

The final chapter presents the conclusions of the research carried out in Chapters 4 and 5 and discusses associated aspects. A critical appraisal of the research in this thesis is given, and some areas for future research are presented.

CHAPTER 2

A SURVEY OF IMAGE COMPRESSION TECHNIQUES

2.1 Introduction

Image compression is needed to both reduce the storage requirements of an image and the transmission time in telecommunication applications. The goal of research into image compression techniques is to produce schemes that give high compression ratios, run efficiently, and result in a high quality reconstructed image. Image compression can be regarded as a particular instance of general data compression except that it takes advantage of characteristics unique to images and the way our visual system views them. All compression aims to minimize the number of information carrying units in the signal that represents the image [48].

There are two phases to compression, and these are the encoding and decoding processes. In some circumstances the reconstructed data must be identical with the original, for example, in textual documents distortion is unacceptable, and in medical images where patient safety is of paramount importance. When a perfect reconstruction of an image is required, compression is constrained by the entropy of the image. To increase image compression, techniques were developed that rely on some degradation to the image that is not detectable to the human eye.

In order to achieve better compression ratios for images, different techniques are used to exploit various image properties. Predictive coders are supported by a statistical model of the image elements, to reduce redundancy before coding with an entropy coder. For example, they store the most frequently occurring elements using codes having the least number of bits. In this form, predictive coding compresses without

loss. Adding quantization to this process increases compression at the expense of some loss of information and hence image quality. Transform coding first maps the image into a domain where it becomes far more amenable to compression, before quantization and entropy coding. Transform coding allows more compression to take place before serious image degradation. Vector quantization is another effective image coding technique.

To enable the transmission and exchange of images in telecommunication applications, there was a requirement for some standard for the storage and compression of photographic images. This was the driving force behind the development of the JPEG[1] International Standard. Research has not waned since the acceptance of this standard, and since then other techniques proposing higher compression while retaining image quality have attracted interest. The most notable are Wavelet and Fractal Image compression.

This chapter outlines the main areas of image compression. It starts by discussing the theoretical basis to compression. It gives an overview of the main image compression techniques, but concentrates on those that form part of the JPEG standard. Two of the more recent techniques are discussed and then compared to JPEG.

2.2 Image Representation

It is useful to first describe how images are represented digitally. The representation will depend on whether the image is greyscale or colour. This thesis is concerned only with greyscale images, hence only these are described here.

A digital image is a computer representation of a continuous tone still picture. The digital image is obtained by analog to digital conversion, which is analogous to dividing the continuous image into a number of evenly spaced discrete scan lines and taking a set number of evenly spaced samples along each scan line. This is depicted in Figure 2.1. The result is a two dimensional array of discrete samples. These

[1] Joint Photographic Expert Group

samples are then represented as numerical values for digital storage. In a greyscale image such as Figure 2.1, each sample value represents a luminance figure from a scale that normally ranges from 0-255, with 8-bit samples. Each sample's luminance value is set independent of its neighbours.

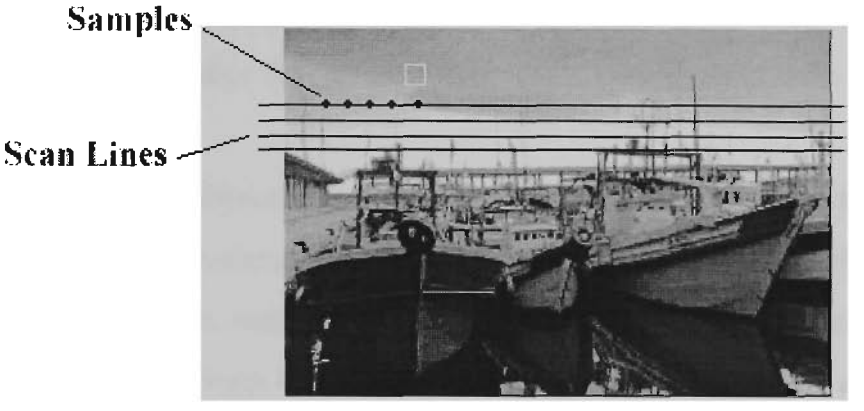


Figure 2.1 Construction of digital image

The sampling process naturally loses information, but if the samples along each scan line are close enough, a reasonable representation of a continuous tone image is obtained. A sampling rate that will obtain good quality images is 400 dots per inch (dpi). Each image sample is called a *pixel* (or *pel*), short for picture element. In the 8-bit digital image shown in Figure 2.1, the white outlined square represents a block of 8×8 pixels, whose pixel values are shown in Table 2.1. A value of zero is used for black, and 255 for white. All other values are varying shades of grey.

Table 2.1 8-bit grey scale values for block in Figure 2.1

162	162	163	164	164	164	165	166
168	168	168	168	168	168	169	169
171	171	170	170	170	170	172	174
178	177	176	175	175	175	176	176
182	178	177	177	177	178	178	178
194	191	188	186	186	185	184	183
193	189	189	189	187	184	185	186
199	196	194	192	189	186	185	186

A relatively small image of 512×512 pixels using 8-bit greyscale takes exactly 262,144 bytes of computer storage space.

2.3 Entropy

Compression techniques exploit redundancy in the data source. Redundancy was explored in detail in a landmark paper by C.E. Shannon [78], while at Bell[2] Laboratories. In this paper, Shannon investigates the transmission of information over a communication channel, and the coding process the transmitter uses to change the information into a form suitable for transmission.

Shannon investigated the difference between the information rate and the data rate of a data source. In a digital system, the bit rate is the product of the sampling rate and the number of bits in each sample, and is usually constant. However, Shannon observed the difference between the bit rate and the information rate of a real signal. This difference is the redundancy of the data source. Shannon used the term *entropy* to describe the information content of a data source.

Messages from most data sources have a degree of redundancy built into them. Where an information source produces messages by selectively selecting symbols from a discrete alphabet, the probabilities of choosing some symbols are dependent on previous choices. For example, in the English language words beginning with the letter *q* are followed by the letter *u*, so the placement of the letter *u* is governed by the statistics of the information source, not by any freedom of choice within the message. Statistically speaking the letter *u* is redundant and need not be stored or transmitted.

One way to measure the redundancy of a signal is to exploit the statistical predictability of the signal. The information content or entropy of a data source is a function of how different it is from the predicted value. According to Shannon's theory, any signal that is completely predictable carries no information [90]. For example, a sine wave is highly predictable because it is periodic. At the opposite extreme, noise is completely unpredictable.

Entropy is a term borrowed from thermodynamics, and then used as a measure of information in a random signal by Hartley [37]. Entropy is a quantification of the

[2] Bell Systems Inc.

information content of the symbols in a message from a given data source. With the entropy quantified, the redundancy rate is then known and a coding algorithm can be devised to remove this redundancy. The relationship between entropy and redundancy in a source is depicted in Figure 2.2. If the signal level and frequency of a transmission are used to denote an area, this sets a limit on the information capacity. As in Figure 2.2, most real signals occupy only a part of that area. The entropy is the actual area occupied by the signal, and is the area that must be transmitted [90].

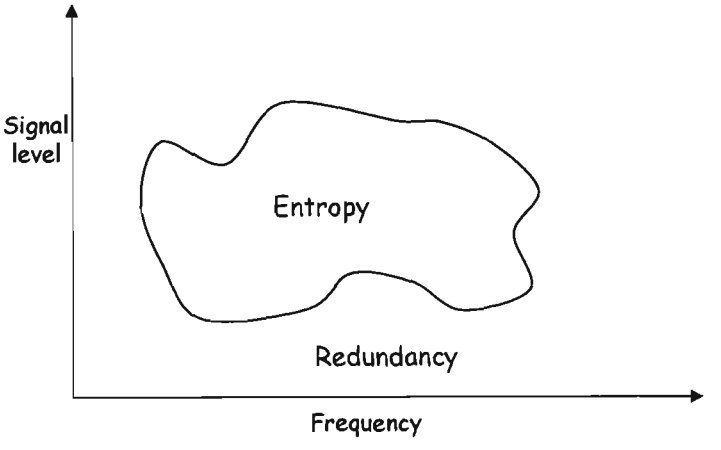


Figure 2.2 Depiction of entropy vs. redundancy in a data source [90]

2.3.1 Measuring Entropy

To calculate the entropy of a source we need to calculate the probabilities of the occurrence of the elements of the source [79]. If an arbitrary element s_i of source S, occurs with probability p_i, then the information content I, contained within that occurrence is:

$$I(s_i) = -\log p_i \quad \text{bits.} \tag{2.1}$$

When the base of the logarithm in (2.1) is 2 then $I(s_i)$ is expressed in bits. It is the minimum number of bits with which this symbol can be represented. The information per symbol $I(s_i)$ averaged for all elements over the whole alphabet of S, is the entropy $H(S)$ of the source. Then

$$H(S) = \sum_{i=1}^{M} p_i I(s_i) = -\sum_{i=1}^{M} p_i \log_2 p_i \qquad (2.2)$$

where M is the number of elements in S. In an image, the entropy is defined in terms of the probability of occurrences of the various pixel values. The entropy defined in (2.2) is the first-order entropy of the source. It takes into account only the relative probabilities of the M possible input values, and no consideration is given to the fact that a particular input may have statistical dependence on previous inputs [28] [48] [33]. If successive inputs are independent, then the first-order entropy $H_1(S)$ from (2.2) forms a lower bound on the average number of bits per input required to code a sequence of inputs from S [28].

If successive inputs of S are dependent, then higher-order entropies that give a better lower bound for a source can be calculated as:

$$H_2(S) = -\sum_{i=1}^{M} \sum_{j=i}^{M} p(w_i, w_j) \log_2 p(w_i, w_j) \qquad (2.3)$$

In (2.3), $H_2(S)$ represents the second-order entropy of S and the function $p(w_i, w_j)$ is the joint probability density function of the two random variables w_i and w_j. Third and higher order entropies can be similarly defined if the inputs have a statistical dependence on the previous two or more inputs. These higher order entropies $H_n(S)$ then become the lower bound on the number of bits required to code the sequence. It can be shown [28], that $H_1(S) \geq H_2(S) \geq \ldots \geq H_n(S)$.

Higher order entropies are usually not pursued, as the cost in computation to obtain them is prohibitive. However, for 6-bit raw image data it has been estimated [75] that the first, second and third-order entropies are approximately 4.4, 1.9 and 1.5 bits per pixel respectively.

According to Shannon's noiseless coding theorem [78] [79], a source can be losslessly encoded to an average bit rate arbitrarily close to but not less than the entropy of the source [25]. The efficiency of coders referred to above can then be measured by how

much redundancy is removed. That is, by how close to the entropy of the source the coders can get. The maximum achievable lossless compression C, is defined by:

$$C \approx \frac{average\,bit\,rate\,of\,original\,raw\,data}{entropy\,of\,the\,source\,data} = \frac{n}{H(S)} \tag{2.4}$$

2.4 Entropy Coders

The coding process removes some redundancy from the data stream by storing the symbols in fewer bits than is used in the original source. The decoding process reverses the coding by adding the redundancy back in, resulting in the restoration of the original data stream. Coders that lose no information during this process are termed *lossless*, while those that do are known as *lossy* coders.

Entropy coders aim to remove the maximum redundancy while remaining lossless. They aim to operate at the entropy bit-rate level and thus achieve near to the theoretical maximum compression in (2.4). Thus, the entropy of a source forms the dividing line between lossy and lossless coders. The best known entropy coders are the *Huffman* coder and the *arithmetic* coder. Both of these coders are used by the JPEG standard.

2.4.1 Huffman Coding

Huffman coding was devised by D.A.Huffman in 1952 [42]. Huffman compression takes advantage of the statistics of the data source by assigning variable length codes to elements of this data source. If all elements of a source were equally probable then fixed-length block coding would be an optimal strategy. In practice, different source symbols have different probabilities of occurrence [25].

The representation of a symbol that is stored by the coding process is called a codeword. Huffman coding assigns small codewords to source symbols with high

probabilities of occurrence and longer codewords to lower probability symbols. This achieves a smaller average codeword length. The probabilities of the symbols have to be known in advance in order to construct the appropriate codewords. Huffman coding requires two passes through the data source. On the first pass, the algorithm collects statistics on the frequency of the symbols and builds the codewords. On the second pass, the symbols are assigned variable length codewords using the statistics gained during the first pass.

Once the frequencies are determined, an algorithm implementing Huffman coding then builds a tree structure from the frequency array. The tree associates each element of the source with a bit string. Each node of the tree contains a data source element, its frequency, a pointer to the parent node, and pointers to the left and right child nodes. An explanation of the building of a Huffman tree is provided by [67].

Table 2.2 Source symbol frequency distribution

Symbol	space	e	o	s	t	i	m	c	h	n	p	r	w
Frequency	5	3	3	3	3	2	2	1	1	1	1	1	1

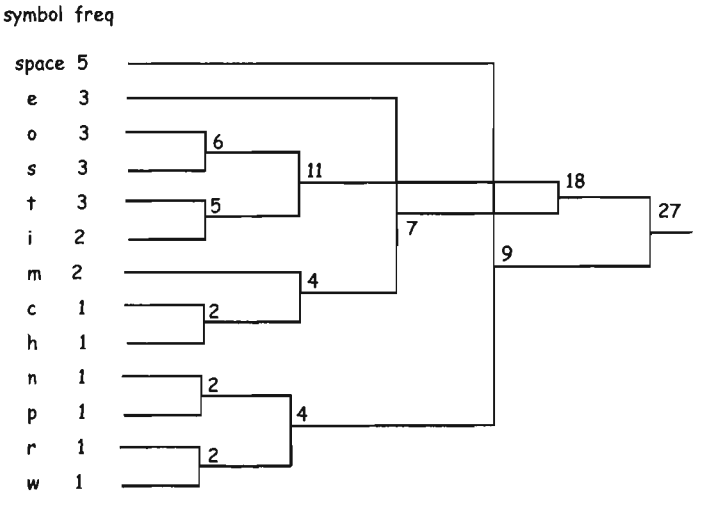


Figure 2.3 Huffman tree built from frequencies in Table 2.2

The Huffman tree grows from the leaves up, with elements of the initial frequency array forming the leaf nodes. This is done by making successive passes through the existing nodes. Each pass searches for the two nodes that do not have a parent node

and have the two lowest frequency counts. When these two nodes are located, a new node is allocated and assigned as the parent of the two nodes, and is given a frequency that is the sum of the frequencies of the two child nodes. The algorithm continues to make passes until only one node with no parent remains, and that node becomes the root of the tree. As an example, the Huffman tree can be built for the coding of the string "*now is the time to compress*". The frequency table is shown in Table 2.2. From this table a Huffman tree is constructed, and is shown in Figure 2.3.

The Huffman algorithm uses the tree to translate elements in the data source into bit-strings. By assigning a zero or one to each left and right node, then a symbol can be coded by tracing a path from its leaf node to the root of the tree. By allocating a zero or one each time you move up from a left or right child the codeword is constructed. There are fewer nodes in the path from the root to the leaves with higher frequencies, so these elements are encoded in fewer bits.

The '*space*' symbol in Figure 2.3 has codeword '*01*', while the '*w*' has codeword '*1111*'. The assignment of the zero and one to the left or right branch is completely arbitrary. The shape of the Huffman tree depends on the order of the search of the frequency array when the tree is initially built. While not unique, the Huffman tree can be shown to be optimal in the sense that no other variation will produce better compression [79] [78].

The decoding process is the reverse. You start at the root of the Huffman tree, and for each zero in the code you move to the right child, and for each one you move to the left child. When a leaf node is reached the element has been decoded.

Some Huffman algorithm variations require only one pass [81], and these fall into two categories; either adaptive algorithms that adjust the frequencies collected from the data source on the fly, or algorithms that take advantage of predetermined frequency tables built from analyzing many data sources from the same language. *Static* Huffman coding assumes the frequencies of elements of the data source are known in advance. In image compression, static Huffman tables are usually built from analyzing the statistics of many images. Tables such as these are provided in the JPEG standard.

2.4.2 Arithmetic Coding

Like Huffman coding, arithmetic coding also produces variable length codes depending on the frequencies of elements of the source. The credit for the idea of arithmetic coding is attributed to Elias [1]. Arithmetic coding generates non-block codes. That is, a one to one correspondence between symbols and codewords does not exist as it does in Huffman coding [29]. Instead, the entire sequence of source symbols is assigned a single arithmetic codeword.

The codeword defines an interval on a probability line between 0 and 1. As source symbols are read in, the interval representing the codeword of the message becomes smaller as it is progressively divided into smaller sub-intervals, dictated by the probabilities of the source symbols. The final output from an arithmetic coder after processing a message is a single number in the range [0,1). For example, the five-symbol sequence, s_1 s_2 s_3 s_3 s_4, is constructed from a four-symbol source. The probabilities of the source symbols are shown in Table 2.3.

Table 2.3 Source symbols and probabilities

Symbol	s_1	s_2	s_3	s_4
Probability	0.2	0.2	0.4	0.2

Using these probabilities the interval [0,1) can be divided into subintervals, each of which represents a source symbol, and its length determined by that symbol's probability, see Figure 2.4. Each symbol owns the half-open interval to which it has been allocated. The first symbol in the sequence s_1 is allocated the half open interval [0, 0.2), and can be represented by any number in that range except the upper limit 0.2.

Figure 2.4 Initial interval divisions according to probabilities

In Figure 2.5, as each new symbol in the sequence is input, the interval remaining from the previous input is again divided up according to the probabilities of the symbols in the source. So when the second symbol in the sequence s_2 is input, the interval [0, 0.2) is divided into the segments shown in Figure 2.5 (a). When the third symbol in the sequence s_3 is input, the interval [0.04, 0.08) is divided as in Figure 2.5 (b). Finally when the last symbol s_4 is input the interval [0.0624, 0.0688) is divided as in Figure 2.5 (d).

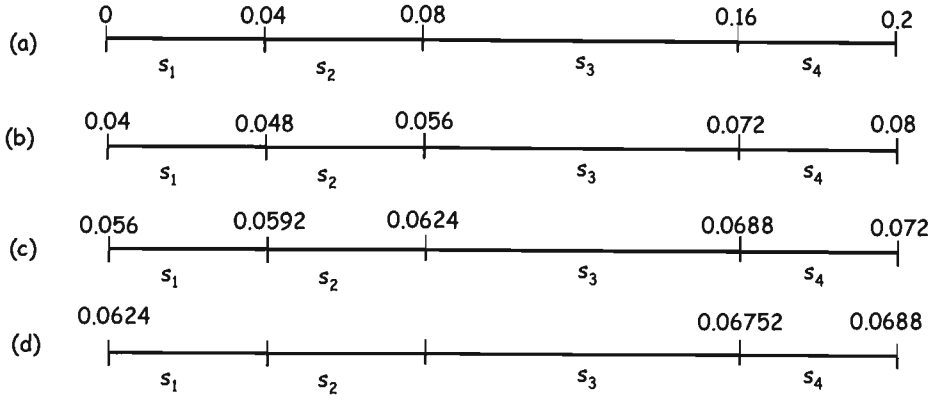


Figure 2.5 Subsequent division of previous symbol intervals

The entire symbol sequence, $s_1\ s_2\ s_3\ s_3\ s_4$, can then be represented by any number in the range [0.06752, 0.0688), e.g. 0.068. This number is then stored in binary form. However, the size of the number obtained can get arbitrarily large with the input symbol sequence, and there exists no single machine having an infinite precision. Consequently, most implementations use a scaling process, and output each leading bit as soon as it becomes known.

The order of the ranges in the probability line is irrelevant provided the coder and decoder are consistent in their assignment. The most significant component of an arithmetic coded message is due to the first symbol encoded. When encoding the above message the first element is s_1, so in order for the decoder to work properly the final coded message must be a number greater than or equal to zero, and less than 0.2. After the first element is coded, the range for the output number is bounded by the interval limits. During the rest of the coding process, each new symbol further restricts the possible range of the output number.

As the length of the input sequence increases, the resulting arithmetic code approaches the bound established by Shannon's noiseless coding theorem, the entropy of the source [29]. In practical implementations, the entropy limit is never reached because of restrictions placed on the algorithm due to underlying hardware [25]. As the number of input symbols increases, the length of the codeword interval decreases, and gets difficult to calculate due to finite arithmetic used in the calculations [29] [25]. These problems are generally overcome by the use a scaling strategy to magnify each interval prior to partitioning.

The addition of an end-of-message indicator also slightly increases the coding rate achievable. Arithmetic coding creates a clear separation between the model for representing the data and the actual coding of the data with respect to the model. This can result in high compression. In practice the amount of compression achieved by arithmetic codes and Huffman codes are comparable, and arithmetic coding is a viable alternative to Huffman [81].

2.5 Image compression

The concept behind image compression is the same as for other data sources, to remove redundancy in the image and thus reduce the required number of bits for storage. The amount of redundancy in an image can be calculated by calculating the entropy of the image using (2.2). As discussed in section 2.3, this will form a lower bound on the average bit rate used to code the image in order to affect a perfect reconstruction [15]. However, the redundancy in a digital image is generally far less than that of other sources. Although the image source data is assumed to be highly correlated, many compression techniques are very sensitive to variations in the input source, which characterizes most images.

The redundancy of the English language is approximately 50 percent [79]. However, the redundancy within a digital image will depend on a number of factors. Factors such as shadings, contrast, complexity (i.e. number of edges), and large areas of the same colour, all determine the redundancy level because they determine the statistics

of the image. Even within a section of an image that appears the same shade, there can be a wide variation in adjacent pixel values used to achieve that shading. This can be seen in the section of the image outlined in white in Figure 2.1, and the corresponding pixel values in Table 2.1. Consequently, coding algorithms that rely on the statistics of the image are less effective when coding digital images. Statistical coders like Huffman and arithmetic coding work best when applied to uncorrelated data [15].

In order to achieve far better compression ratios for images, different techniques are used to exploit various aspects of the image properties. This section starts by discussing predictive coding in its lossless form. Quantization is introduced to increase compression and give a lossy version. There is a range of transforms that could be used as transform coders and they are compared. Most compression in the transform coders is the result of quantization of the transform coefficients. Finally, Vector Quantization, which was also a candidate for the JPEG standard, is described.

2.5.1 Predictive Coding

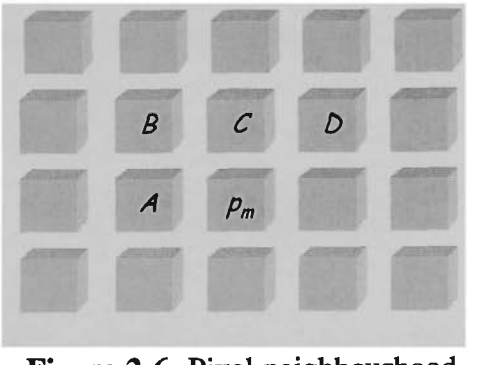


Figure 2.6 Pixel neighbourhood

Predictive coding is carried out in the data domain, and it is based on the principle of removing the mutual redundancy between adjacent pixels, by taking advantage of known pixel values from previous inputs. The image data source is assumed to be highly correlated, that is, pixels of the image within the same neighbourhood tend to have similar values. The value of previous pixels which have already been encoded in the same scan line or earlier section of the image can be used to make a prediction for the current pixel. For example, Figure 2.6 shows a pixel p_m, within a portion of an

image and its neighbouring pixels which have preceded it labeled as A, B, C and D. These neighbouring pixels can be used to form a prediction for p_m.

Differential Pulse Code Modulation (*DPCM*) is the most common form of predictive coding [25] [48], and is used by JPEG. The original design of DPCM systems was developed in 1952 by Culter [19]. In DPCM, the image is encoded one pixel at a time, from left to right top to bottom across the scan lines of an image (Figure 2.1). When encoding pixel p_m in Figure 2.6, advantage is taken of the fact that previously coded pixels may contain some information about it [48]. Accordingly, a prediction ε_m on the value of p_m can be made. A linear prediction is defined by

$$\varepsilon_m = \sum_{i=0}^{m-1} a_i \, p_i \tag{2.5}$$

where ε_m is the prediction, the p_i are the m previous pixel values, and a_i represents a weighting factor for the corresponding p_i [25]. The number of pixels used in the prediction is called the order of the predictor. There is usually not much to be gained in coding with more than a third-order predictor [34]. An optimal set of predictor coefficients can be computed for a given image. For example, for an image based on a first order *Markov* process with correlation coefficient ρ, the optimal third-order predictor is given by

$$\varepsilon_m = \rho A + \rho^2 B + \rho C. \tag{2.6}$$

However in practice, simpler predictors are the predictions,

$$\textit{(first order)} \quad \varepsilon_m = A \tag{2.7}$$

$$\textit{(second order)} \quad \varepsilon_m = \tfrac{1}{2}A + \tfrac{1}{2}C \tag{2.8}$$

$$\textit{(third order)} \quad \varepsilon_m = A - B + C. \tag{2.9}$$

When the prediction of a pixel's value is based on previous inputs from the same scan line, the coder is said to a 1-dimensional DPCM coder. When predictions are based

on previous inputs from more than one scan line, the DPCM coder is said to 2-dimensional.

After the prediction has been made, the difference between the actual value of the pixel and the prediction is then calculated. This quantifies the 'new information' of the pixel and eliminates the redundancy. The predicted values ε_m are significantly less correlated than the original pixel values p_m. In lossless DPCM, the predictions are coded directly using an entropy encoder such as Huffman coding.

Lossy DPCM is achieved with quantization of the predictions prior to encoding. Quantization is discussed in detail in the next section. The DPCM block diagram for lossy coding is shown in Figure 2.7.

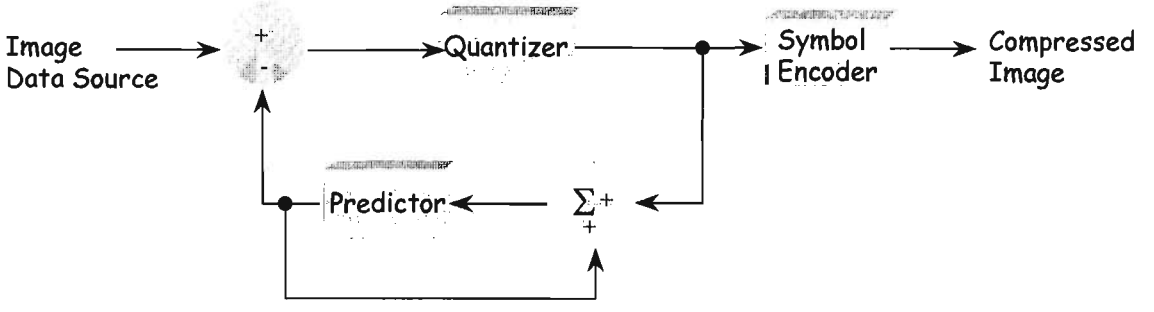


Figure 2.7 DPCM lossy encoder block diagram

The data compression depends on the ability to predict p_m accurately, and therefore the inter-pixel correlation. In a highly correlated data source, the predictions are expected to be good on average. Predictive lossy algorithms are capable of reproducing images almost indistinguishable from the originals at compression rates around 2.5:1 [29] [55]. However, severe distortions are noticeable when coding at bit rates less than 3-bits per pixel from original 8-bit source data [28], which represents compression rates of about 4:1. Lossless predictive algorithms generally produce compression rates between 1.5:1 and 2:1 depending on the image [55].

Predictive Coding is easy to implement and gives good reproduction at lower compression ratios. However, since predictive coding works by removing redundancy between pixels in the spatial domain, it is very sensitive to changes in the

input data. For example, it does not perform well in images where there are a lot of edges and sharp contrasts.

2.5.2 Transform Coding

Transform coding is more complex to implement. The main steps in transform coding are shown in Figure 2.8. The technique gets its name from the first step, which is the *transform* stage.

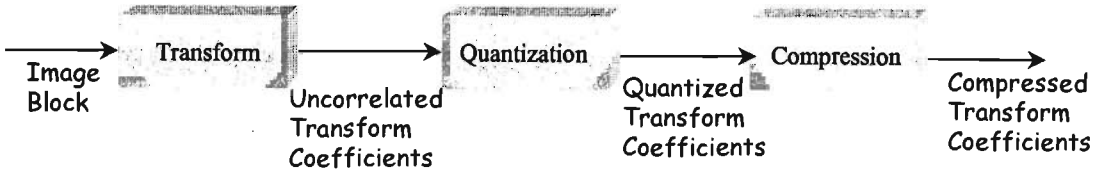


Figure 2.8 General steps in transform coding

At its simplest, the transform is a process that maps data from one domain into another [90]. The transform coding of 2-dimensional images was introduced by Andrews and Pratt [6] [48]. The image is divided into blocks of pixels, a typical block being 8×8 or 16×16 pixels. The transform is then applied to each block, which effects a spectral decomposition of the input data, into a set of transform coefficients in the frequency domain. Some of the better known transforms used in image coding are discussed in following sections.

The transform itself does not provide any compression. It maps the image into the frequency domain where compression can be achieved more effectively. The general form of a 2-dimensional transform that maps the pixels x_{ij} into transform coefficients y_{kl} [28] is given by

$$y_{kl} = \sum_{i=1}^{n} \sum_{j=1}^{n} x_{i,j}\, a_{ijkl} \qquad (2.10)$$

where the symbol a_{ijkl} is the forward transformation kernel. The coefficients of the transform $\{y_{k,l}\}$ are approximately uncorrelated, which means that most of the redundancy in the input data has been removed. A large proportion of the block's information is packed into a relatively few transform coefficients. The efficiency of the transform will dictate how many of the block's transform coefficients are significant in representing each block. In general, the more efficient the transform is at packing the blocks information into fewer coefficients, the longer it takes.

Quantization

The quantization stage reduces the accuracy with which the transform coefficients are represented by giving fewer values to the coder. This is an important step as it can make many of the transform coefficients zero. Its role is to further prepare the coefficients for coding, so the maximum compression can be gained. However, if the quantization scheme is not carefully chosen, it can cause severe image degradation.

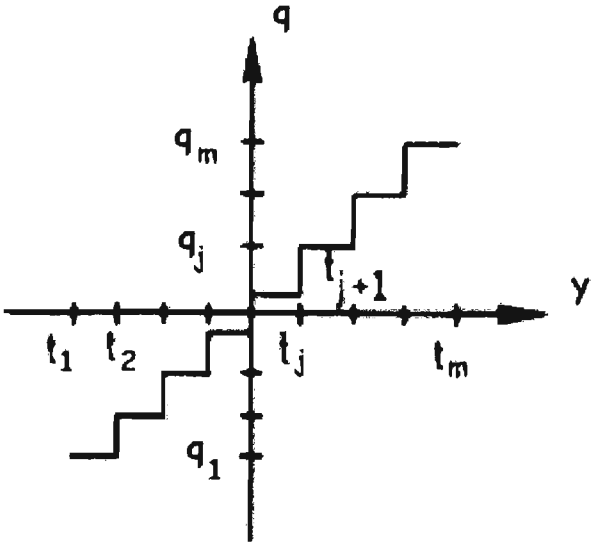


Figure 2.9 Typical I/O characteristics of a quantizer [48]

Most digital image compression algorithms require a quantizer [48]. In the case of transform coding, the quantizer maps the discrete decorrelated coefficients $\{y_{kl}\}$ from (2.10) into a smaller set $\{q_1, q_2, \ldots, q_m\}$ of quantization levels. This mapping is usually referred to as a staircase function, and is depicted in Figure 2.9. This works as

follows. The set $\{t_1, t_2, ..., t_m\}$ defines a set of increasing transition levels (or decision) levels. If transform coefficient y_{kl} lies in the interval $(t_j, t_{j+1}]$ then y_{kl} is mapped to q_j where $j \in [1, ..., m]$. The quantity q_j, called the reconstruction level, is the quantized value of y_{kl} and lies in the interval $(t_j, t_{j+1}]$. During decoding, the quantized value q_j is assigned a representative value from the interval $(t_j, t_{j+1}]$, which is usually a number mid-way in this interval. The quantizer design problem is to determine the optimum transition and reconstruction levels given the probability density and optimization criterion [48].

There are several quantizer designs available that offer various tradeoffs between simplicity of implementation and performance. A well-known quantizer design is based on the Lloyd-Max algorithm [62,64], that can iteratively develop the quantizer based on minimizing a specific distortion (quantization noise) for a particular distribution [70]. Quantization of image samples for compression is called Pulse Code Modulation, (PCM) [48], however it should be noted that when converted from an analog format a digital image is already PCM coded.

Lossy Transform Coding

Transform coding achieves relatively larger compression ratios than predictive coding techniques through quantization, but is essentially a lossy process. Loss is introduced into transform coding during the quantization stage, where the transform coefficients are mapped into a smaller output range of values. When these quantized values are used as coefficients on the inverse transform during transform decoding, error is introduced as information has been lost from the original image block. During the compression or coding stage, an entropy encoder such as Huffman or arithmetic coding is usually used, and these are not a source of loss in themselves. However, the coding stage can introduce further loss if not all the quantized coefficients are coded.

An efficient transform packs much of a block's energy into a few transform coefficients. To achieve higher compression ratios many of the least significant transform coefficients can be simply dropped off during the encoding. While this is a

source of further loss, the least significant transform coefficients represent very high frequency information (fine image detail), whose loss from an image block may not be visually perceived. For example, in an 8×8 transformed block of coefficients, over half of the 64 transform coefficients can be omitted from the encoding without much perceptible loss of visual detail. This, however, is very subjective and depends on the application.

Predictive vs. Transform Coding

As discussed in section 2.5.1, predictive coding is very sensitive to changes in the statistics of the data, for example, at edges where there is usually little correlation between elements in the image. Transform coding however, is largely unaffected by this and will distribute the energy of each transformed block over the transform coefficients. Normally only adaptive predictive techniques achieve the compression rates of transform coding. However from an implementation viewpoint, predictive coding techniques are less complex.

Details of some transforms used for image transform coding are provided in the following sections.

2.5.2.1 Fourier Transform

The *Fourier Transform* has many applications in science and engineering. It was only natural that the Fourier Transform be one of the first applied to image processing, and over the years has been successfully used in many aspects of this field. For image coding, the Fourier Transform's usefulness lies in its ability to convert a signal from the spatial domain to its frequency domain.

The Fourier Transform is simply a change of coordinates [15]. The original coordinate system is called the spatial domain for image functions, and the Fourier Transform space is called the frequency domain. Because a digital image is a two-dimensional

set of discrete samples, the *Discrete Fourier Transform* (DFT) is used in digital image processing. Its definition [28] is

$$F(u,v) = \frac{1}{N}\sum_{x=0}^{N-1}\sum_{y=0}^{N-1} f(x,y)\, e^{-j2\pi(ux+vy)/N} \tag{2.11}$$

Equation (2.11) is the 2-dimensional DFT, and the value N is the size of the square sample block. This is appropriate since an image is a two dimensional object.

One of the properties of the DFT that makes it particularly useful for image coding is its separability. This means that the 2-dimensional transform in (2.11) can be calculated by successive application of a 1-dimensional transform. Given an 8×8 image block, the 2-dimensional DFT can be calculated by application of the 1-dimensional transform (shown in 2.12) across the rows of the block, then down the columns of the block. The result is an 8×8 array of Fourier transform coefficients defined by

$$F(u) = \frac{1}{N}\sum_{x=0}^{N-1} f(x)\, e^{-j2\pi ux/N} \tag{2.12}$$

where N is the number of samples. One of major uses of the separability of the transform is in the development of a *Fast Fourier Transform*, which is used in place of a DFT to reduce the processing time involved. This can only be done by reducing the number of arithmetic computations. The number of complex multiplications and additions of the DFT in (2.11) is $O(N^2)$, where N is the size of the image block [28]. In 1965, Cooley and Tukey [18], developed the *Fast Fourier Transform* (FFT) which is a DFT algorithm. The FFT reduces the number of computations to $O(N \log_2 N)$. The FFT algorithm is simplified when N is a power of two, but it is not a requirement.

2.5.2.2 Discrete Cosine Transform

The *Discrete Cosine Transform* (DCT), was first applied to image compression in 1974 by Ahmed, Natarajan and Rao [2]. It was demonstrated that the DCT performs close to the optimal *Karhunen-Loeve* Transform (section 2.5.2.3), in producing uncorrelated coefficients. Uncorrelated coefficients are important in image coding, because it means that during the coding stage, the coefficients can be treated independently without any compression degradation. Another feature of the DCT is the ability to quantize the coefficients using visually weighted quantization values [69]. The 2-dimensional DCT [70] is defined by:

$$F(u,v) = \frac{1}{4} C(u)C(v)\sum_{x=0}^{7}\sum_{y=0}^{7} f(x,y)\cos\frac{(2x+1)u\pi}{16}\cos\frac{(2y+1)v\pi}{16} \qquad (2.13)$$

where $C(u),C(v) = \frac{1}{\sqrt{2}}$ *for* $u,v = 0$; $C(u),C(v) = 1$ *otherwise*.

As for the DFT, the theoretical equation in (2.13) involves many real multiplications and additions. The development of efficient algorithms for computation of the DCT began soon after Ahmed et al. reported their work in [2], [70]. Many algorithms have been reported, but the algorithm developed by Chen, Smith and Fralick [14] is regularly used in implementations of the DCT [69].

Chen et al. define a method for computation of the DCT by use of sparse matrix factorizations [70]. This results in a series of alternating sine/cosine butterfly loops in the calculations and results in a significant reduction in real multiplications and additions. According to [2] [70], this algorithm requires $\frac{3N}{2}(\log_2 N - 1) + 2$ real additions, and $N \log_2 N - \frac{3N}{2} + 4$ real multiplications, where N is the sample size. This FDCT algorithm is discussed and the signal flow diagram is provided in Chapter 3, section 3.2.3.2. This is a 1-dimensional FDCT. However, since the DCT is a separable transform, this can be used as a 2-dimensional FDCT by taking 1-dimensional transforms across the rows of an image block, and then down the columns. This transform is implemented in the research algorithms.

In 1985, Haque [35] reported a 2-dimensional recursive FDCT, that operated by rearrangement of the input block into a block matrix form [70]. Each block is then put through a half-size 2-dimensional FDCT. This method reported the number of real multiplications of an N^2 image block as $\frac{3}{8}N^2 \log_2 N^2$.

The energy packing efficiency and performance of the DCT is superior to that of the DFT of the previous section, and approaches that of the optimal *Karhunen-Loeve* transform discussed in the next section. For these reasons the DCT has become important to image coding, and is integral to the international standard for continuous tone still picture image compression [52].

2.5.2.3 Karhunen-Loeve Transform

The *Karhunen-Loeve Transform* (KLT), was first discussed by Karhunen [54], and then later by Loeve [63], [70]. This transform is a series representation of a given random function. The KLT is an optimal transform because it displays the following properties, [70]:

- The transformed block's coefficients are completely decorrelated in the transform domain.
- The Mean Square Error (MSE) is minimized in data compression.
- It packs the most of an image block's energy in the fewest number of transform coefficients.
- It minimizes the total representation entropy of the image block.

Despite the theoretical superiority of this transform over other transforms, it has practical limitations which prohibit its use for applications. First, it is necessary to estimate the source block covariance matrix. Next, if the transform is to be optimal, this needs to be done for both row and column processing in a 2-dimensional coding scheme. Then the eigenvector determination has to be performed to produce the basis matrix. All this has to be done before any image coding takes place, and must be

done for each block. The basis vectors need to be transmitted along with each coded block.

The KL Transform is the best linear transform in the sense that it leads to uncorrelated coefficients (see section 2.5.2.5). It is not often used in practice because of its computational load. It does however provide an upper bound of what other transforms which are more computationally efficient, should attempt to reach [55]. Fast KL Transforms are reported by Jain [48], which decompose the original random process into two mutually orthogonal processes with fast KL Transforms. The efficiency of this method approaches the original KL Transform efficiency. However, the computational effort is still in excess of that needed for other efficient transforms.

2.5.2.4 Walsh-Hadamard Transform

In the literature, the term *Walsh-Hadamard Transform* (WHT) is often used to denote either the Walsh Transform, or the Hadamard Transform. When N is a power of two, the *Discrete Walsh-Hadamard Transform's* (DWHT) kernel, forms a symmetric matrix whose rows and columns are orthogonal. These properties lead to an inverse kernel that is identical to the forward kernel except for a constant multiplication factor of *1/N*. Unlike the Fourier Transform, which is based on trigonometric terms, the DWHT consists of a series expansion of basis functions whose values are either +1 or -1 [28]. The 2-dimensional DWHT is

$$H(u,v) = \frac{1}{N}\sum_{x=0}^{N-1}\sum_{y=0}^{N-1} f(x,y)(-1)^{\sum_{i=0}^{n-1}[b_i(x)b_i(u)+b_i(y)b_i(v)]} \tag{2.14}$$

where $b_k(z)$ represents the i^{th} bit in the binary representation of z [28].

The WHT kernels are also separable, so the two dimensional transform in (2.14) can be computed by successive applications of the 1-dimensional transform. The procedure followed is the same as that for the DFT and the DCT. Algorithms used to

compute the FFT can easily be modified to compute a Fast DWHT by setting all the trigonometric terms to one.

The WHT is real, and as such, computer algorithms to compute the DWHT generally needs less storage space than the Fourier Transform algorithms which are generally complex. The WHT is also fast compared to the other separable transforms since its computations involve no multiplications [70]. The WH Transform is often used in the fast computation of other transforms such as the DCT and the DFT.

2.5.2.5 Summary

This section briefly summarizes the transforms described in the previous sections.

The KL Transform is optimal on a mean square error basis, but is difficult to implement. The calculations involved make it prohibitively time consuming for implementation as a real time algorithm. This transform is most useful as a benchmark.

The DFT involves complex variables and its use is recommended only if the frequency domain is mandatory such as in visual coding where the source data has to pass through the Fourier domain. The DCT performs well for highly correlated data (i.e. correlation coefficient ≥ 0.5), such as in digital images. The WHT is useful for small block sizes ($\approx 4 \times 4$). The implementation of this transform is much simpler than either the DFT or the DCT.

The following graphs (Clarke [15]), compare some transforms according to a variety of criteria. For digital image coding, the most useful criteria are the energy packing ability of the transform, and its ability to decorrelate the transform coefficients.

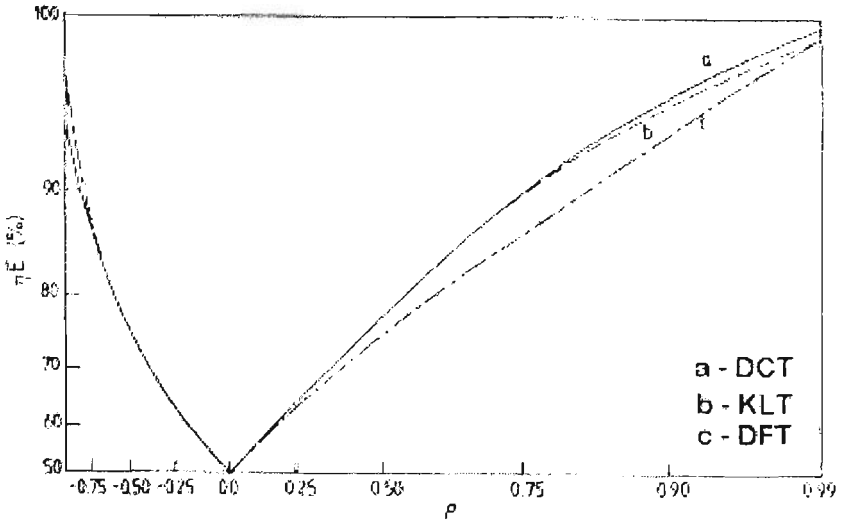


Figure 2.10 Energy packing efficiency as a function of correlation coefficient [15]

Figure 2.10 shows the energy packing efficiency η_E as a function of the correlation coefficient ρ, with the number of samples N being 8 with 4 coefficients retained. The DCT has a slight edge particularly as the inter-element sample correlation increases. The DFT has the poorest performance.

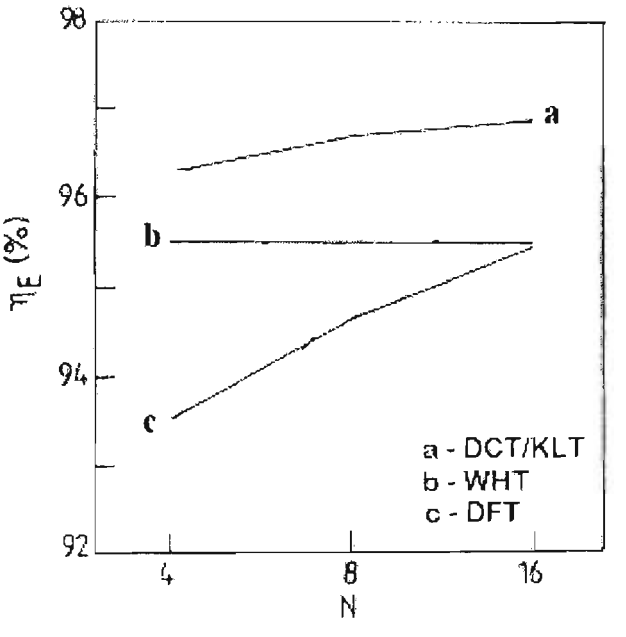


Figure 2.11 Energy packing efficiency as a function of transform block size [15]

Figure 2.11 shows the energy packing efficiency, η_E, as a function of the input block size. Again, the DCT performs at the optimal level of the KLT.

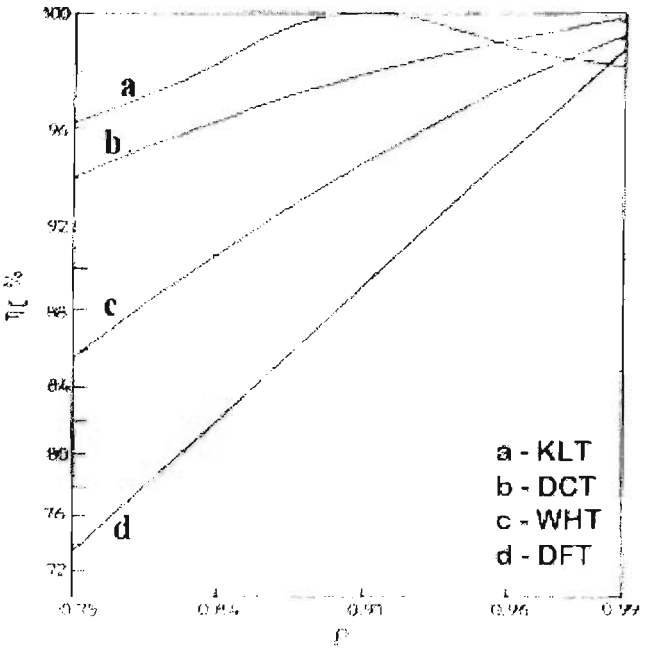


Figure 2.12 Decorrelation efficiency as a function of the correlation coefficient [15]

Figure 2.12 shows the decorrelation efficiency η_c as a function of the correlation coefficient ρ. The KLT displays the optimal decorrelation of transform coefficients, but the DCT performs closer to this than the other transforms.

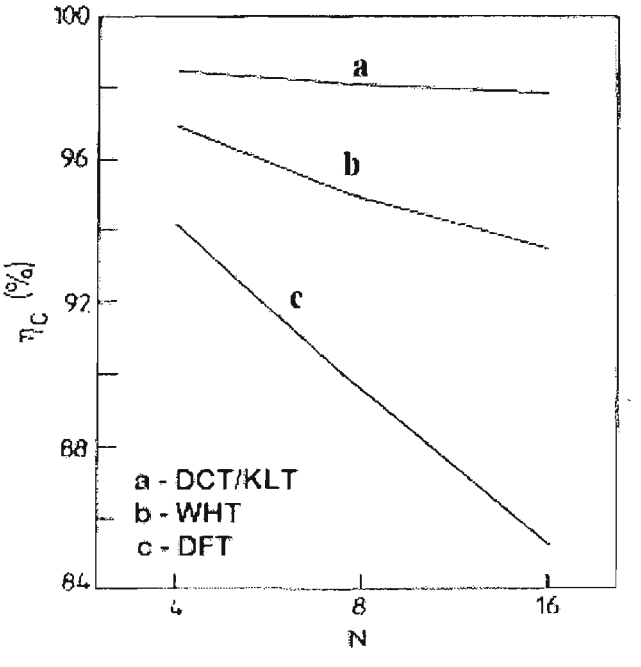


Figure 2.13 Decorrelation efficiency as a function of transform block size [15]

Finally, Figure 2.13 shows the decorrelation efficiency as a function of the transform block size. The DCT is performing at near optimum performance compared to the other transforms.

The graphs in the four figures from this section are derived from [15], and in many cases the scales of the ranges have been expanded to show some detail. This was necessary since the area of interest was around 100 percent.

The conclusion to be drawn is that for practical purposes the DCT of all the transforms is best (in that it performs close to the KLT), provided the correlation coefficient between data samples is high [15]. This is usually the case in image processing.

2.5.3 Vector Quantization

Vector Quantization (VQ) has raised some considerable interest since, in principle, it can nearly achieve optimal rate-distortion performance [3]. VQ is the joint quantization of a block of signal values. In VQ, an n-dimensional input vector $[x_1, x_2, ..., x_n]$ denoted by X, whose values represent discrete samples of a signal are mapped, or quantized into one of N possible reconstruction vectors Y_i, where $i = 1, ..., N$ [32]. The distortion in approximating the discrete sample X with Y_i is denoted by $d(X, Y)$. The most common distortion measure is *MSE*, which is calculated as the square of the *Euclidean* distance between the two vectors, as:

$$d_{MSE}(X,Y) = \frac{1}{n}\sum_{i=1}^{n}(x_i - y_i)^2 \qquad (2.15)$$

The set of quantization vectors Y is often called the reconstruction *codebook*. The quantization process is straightforward, and is based on a minimum distortion rule. An input vector of discrete samples is compared to all the quantization vectors in the codebook, and is quantized to the codebook entry that results in the minimum distortion. That is, the codebook entry Y_k is chosen such that

$$d(X,Y_k) \leq d(X,Y_j) \quad for\ j=1\ldots N. \tag{2.16}$$

The only thing that needs to be kept is the index number of the codebook entry used. With N possible codebook entries, the output of the vector quantizer can be specified with $log_2 N$ bits, and the resulting bit rate per vector component is $(log_2 N)/n$ bits [25]. A fundamental result of Shannon's rate-distortion theory [79] is that better performance (compression) can be achieved by coding vectors instead of scalars, and as $n \rightarrow \infty$ the quantizer distortion rate gets arbitrarily close to the rate distortion bound, and for codebook of size N, the output entropy approaches $log_2 N$ [32].

Large values for the input vector size n and large values of N can make searching a codebook prohibitive. To help minimize the mean distortion, the temptation is to increase the number of quantization vectors, N. However, as N increases, the time to search the codebook grows geometrically. Care must be taken to construct the initial codebook carefully. The codebook vectors are normally designed in the spatial domain by a cluster algorithm named *LGB* developed by Linde, Buzzo & Gray [61].

Between 1980 and 1982 four separate groups developed successful applications of VQ techniques to image coding [32]. The only difference between these applications and the techniques discussed thus far is that these applications used two-dimensional vectors, or blocks from the input image. In [32] example images are coded using 6-bit codebook vectors and input block size of 4×3 pixels, at a compression ratio 16:1. At this rate, the blocky effect of the distortion was severely noticeable.

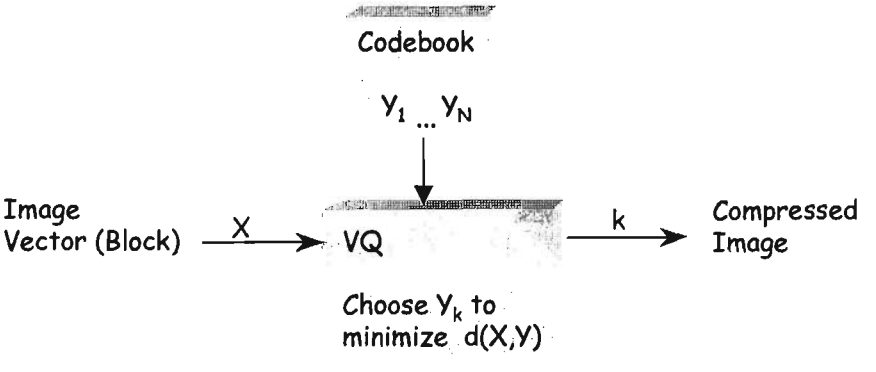


Figure 2.14 Vector quantization block diagram

It should also be noted that the algorithm for generation of codebook entries [61], is image specific. A VQ technique shows considerable degradation of quality for images not included in the codebook design [3]. A block diagram showing the generalized process of Vector Quantization is shown in Figure 2.14.

2.6 Selection of International Standard

Towards the end of 1986, a group of experts formed the JPEG. Their task was to select a high performance universal compression technique for development as an international standard [41]. The JPEG conducted subjective image quality evaluations on two occasions with a number of image compression techniques in order to select a candidate for the international standard.

The first took place in June 1987 at KTAS[3]. Twelve image compression techniques were tested according to a number of criteria, and pre-determined compression rates. From these twelve, three finalists were selected for further subjective testing at a later meeting of the JPEG [41] [58].

Four pictures were tested at compression rates of 0.25 bits/pixel, 0.75 bits/pixel and 4.0 bits/pixel, the later compression rates providing images "indistinguishable" from the original. From this testing three techniques were chosen for further analysis, the Adaptive Discrete Cosine Transform (ADCT), the Adaptive Binary Arithmetic Coder (ABAC), and the Block Separated Progressive Coding (BSPC).

The second meeting of the JPEG also held at KTAS, took place in January 1988. The purpose of this meeting was to select from the above three techniques, one that would be chosen for refinement with the goal of producing an ISO[4] standard. The algorithms were required to produce progressive stage images coded at 0.08, 0.25, 0.75 and 2.25 bits/pixel.

[3] The Copenhagen Telephone Company Research Labs

[4] International Standards Organization

The ADCT technique excelled in producing images at all of the above bit-rates. The ADCT was selected as the technique for final refinement with the aim of issuing the draft standard to ISO bodies during 1989, for final voting in 1990. This technique became the International Standard for continuous tone still picture compression in 1992.

2.7 Recent Compression Techniques

Since the acceptance of the JPEG standard as the international standard for continuous tone still picture compression, other techniques for compression have gained in popularity. Two of these techniques are *Wavelet compression* and *Fractal compression*. There are claims that both of these methods offer superior compression and quality of compressed images to JPEG, with some skepticism in the international community over the case for Fractal compression.

In the following two sections both of these recent compression techniques are discussed, followed by a comparison of the three techniques, JPEG, Wavelet and Fractal compression.

2.7.1 Wavelet Compression

Wavelet compression involves the use of the Wavelet Transform. The Wavelet Transform was not discovered by any one individual but grew from a number of similar ideas. The first mention of wavelets appeared in an appendix to the thesis of A. Haar in 1909, but they were not given a strong mathematical foundation [23] until relatively recently [90]. The Wavelet Transform has a close association to the Fourier Transform. Where the Fourier Transform breaks a signal into a series of sine waves of different frequency, the Wavelet Transform breaks the signal into wavelets which are scaled and shifted versions of the "mother wavelet" [5].

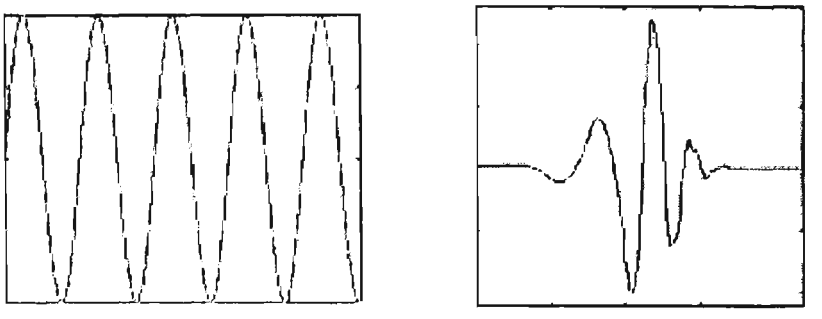


Figure 2.15 Comparison of sine wave and Daubechies wavelet

Figure 2.15 shows the sine wave in comparison to the Daubechies-5 wavelet. In comparison to the sine wave that is periodic and infinite, the Daubechies wavelet is of irregular shape and compactly supported. These properties of wavelets make them ideal for analysing signals. The irregular shape lends itself to analysing signals with discontinuities or sharp changes, which correspond to object edges and contours in images.

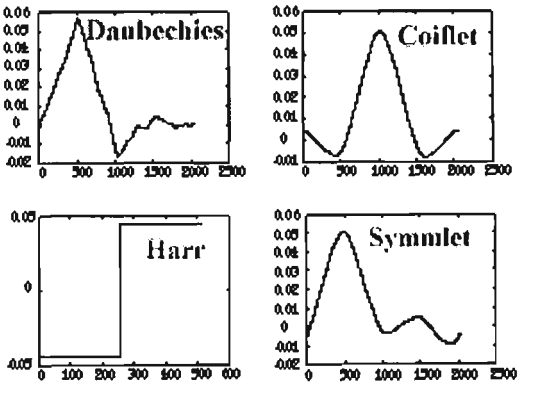


Figure 2.16 Some well known wavelet families

There are an infinite set of Wavelet Transforms, but only a few useful ones. Some of these can be seen in Figure 2.16. Different sets of wavelet families make different trade-offs between how compactly the basis functions are localized in space and how smooth they are. The wavelet signal transform procedure is to adopt a wavelet prototype function called a "mother wavelet". Temporal and frequency analysis is performed using high and low frequency versions of the same wavelet. The original signal can be represented in terms of wavelet expansion, using coefficients in a linear combination of the wavelet basis functions. Compression can be achieved using these coefficients as input to an entropy coder.

Where this is different from the DCT is that a set of wavelet basis functions can be obtained by simply scaling a single wavelet on the time axis. Each wavelet contains the same number of cycles such that, as the frequency reduces, the wavelet gets longer. Thus, the frequency discrimination of the Wavelet Transform is a constant fraction of the signal frequency. This is depicted in Figure 2.17.

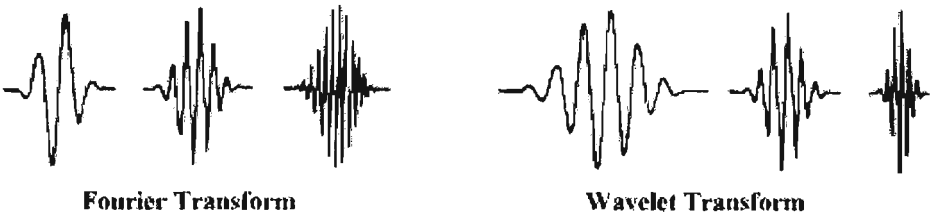


Figure 2.17 Discrete fourier basis functions vs. scaled wavelet basis functions

For the discrete Wavelet Transform, dilations and translations of the mother function $\Phi(x)$ define the wavelet basis [31], as

$$\Phi_{(sl)}(x) = 2^{\frac{-s}{2}} \Phi(2^{-s}x - l) \tag{2.17}$$

where variables s and l are integers that scale and dilate the mother function $\Phi(x)$ to generate wavelets. The scale index s indicates the wavelets width, and the location index l gives its position. To span the data domain at different resolutions, the mother wavelet is used in the scaling equation as

$$W(x) = \sum_{k=-1}^{N-2} (-1)^k c_{k+1} \Phi(2x + k) \tag{2.18}$$

where $W(x)$ is the scaling function for $\Phi(x)$, and $\{c_k\}$ are the wavelet coefficients.

The fundamental idea of wavelets is to analyze according to scale. Wavelet algorithms process data at different scales or resolutions. If we look at a portion of an image with a large continuous area, we notice coarse features, while if we look at a

portion containing smaller areas we notice fine detail. Wavelets analyze images in this way, and these areas are known as the *trends* and *outliers* in wavelet compression.

Trends span a wide range and have high spatial correlation. They correspond to the slowly varying large parts of an image (e.g. the light grey colour of a person's face). Outliers are bursty and have low spatial correlation, and these correspond to the very concentrated, and rapidly changing areas of an image (e.g. edges). The wavelet transform encodes the trends at low resolution, and the outliers at high resolution [73]. That is, high frequencies in an image corresponding to outliers are transformed with short basis functions, whereas low frequencies corresponding to trends are transformed with long basis functions [90].

Figure 2.18 Outliers and trends coefficients

Therefore, the low resolution areas contain many significant coefficients, but have small size, while high resolution areas are made mainly of zeros, with few coefficients representing the outliers, as shown in Figure 2.18. The trick of wavelet compression is to quantize the transformed image so that many "unnecessary" high resolution coefficients are mapped to zero and, to find an efficient way to encode the position of the zeros. Run length encoding or Huffman coding is usually used.

At low bit rate compression, the JPEG DCT introduces a blocky effect into the reconstituted image due to the division of the original image into sub-blocks. This blocky effect is also responsible for artifacts. Similar low bit rate coding in wavelets tends to produce a smearing around edges in the reconstituted images, and the artifacts are more difficult to characterize. JPEG artifacts are always in the same

position of an image since they are caused by the division into blocks. In wavelet compression, the errors responsible for artifacts tend to be spread over the entire image.

The work with wavelets for data compression is along a similar theme to earlier work [55] [56], with pyramidal encoding and contour-texture techniques. These techniques were based on studies of the human visual system and used edge detection and contour mapping to code edge areas of the image at a higher rates than other areas. Comparing JPEG and wavelet compression reveals a higher image quality for a given bit rate with wavelets than for JPEG.

In 1993, the U.S. Federal Bureau of Investigation (FBI) Criminal Justice Information Services Division, developed standards for fingerprint digitization and compression. These standards are based on a wavelet compression technique, and were developed in cooperation with the U.S. National Institute of Standards and Technology [31]. The FBI is using these standards for the coding of their approximately 2,000 terabytes of fingerprint data collected from 1924.

2.7.2 Fractal Compression

The term *fractal* was first used in 1977 by Benoit Mandelbrot to describe an iterative mathematical technique that generates extremely complex yet natural looking images [77]. Fractals have been applied to image compression, and are used to represent the structures in images. Fractals can be loosely defined as shapes that are irregular, and have the interesting property of self-similarity [25]. A self-similar object is one that looks approximately the same regardless of the scale at which it is viewed.

One of the better known examples of images generated using fractals is that of the fern leaf shown in Figure 2.19. The most important property of fractals is the small amount of information which can be used to generate a complex image. The fern leaf example image can be generated from 12 bytes of data. This is the property that makes fractals an ideal tool for image compression.

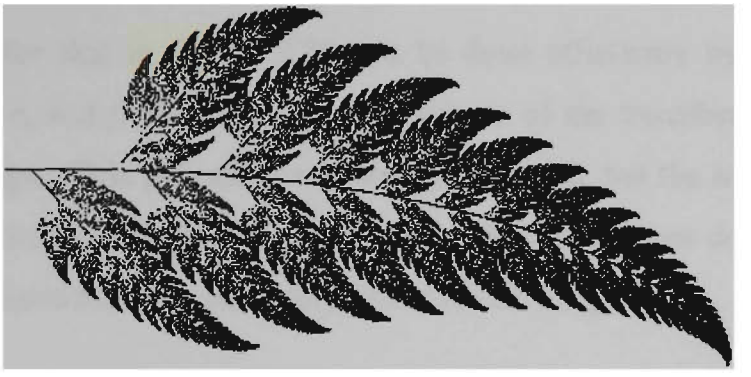


Figure 2.19 Fractal generated fern leaf

Another example of a fractal is the *Sierpinsky Triangle*, whose construction can be seen in Figure 2.20. The original image is shrunk by half, then pasted to the top, left and right corners. The process proceeds in an iterative fashion. Fractals can be constructed in this manner [74]. The two properties of importance here are, "the further the process goes the more detail is added", and "a different initial image can be used to construct the same fractal". It does not make a difference which initial image is used, only the process is important.

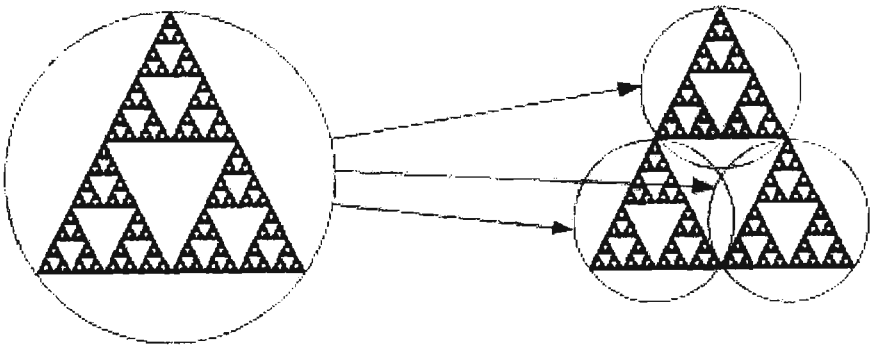


Figure 2.20 Sierpinsky's triangle construction

The fractal can be determined by its transformation process, and the initial image is the *attractor* of the process [74]. Different transformations lead to different fractals. In most cases only a limited type of transformation is used, and these do simple things such as scaling, moving, rotating or flipping an image. The transformations are affine and are written as

$$w(x,y) = (ax+by+e, cx+dy+f) \tag{2.19}$$

Storing images like that in Figure 2.20 can be done efficiently by just storing the values *a*, *b*, *c*, *d*, *e*, and *f*, which are the coefficients of the transformation needed to construct the image. This process seems straight forward, but the inverse problem is more difficult. That is, "Given a complex, real-world image how do we arrive at the set of transformations that can accurately generate that image".

A lot of initial interest in fractal based compression is due to the work of Barnsley and Sloan [9], [7], [80]. In 1988, Dr Michael Barnsley claimed to have discovered the key to the inverse problem. He invented the *fractal transform*, and registered the term as a trademark, forming a company called Iterated Systems[5]. Barnsley and Sloan have made claims of extremely high compression ratios while supplying only limited technical details of the process, supposedly for patent issues [25]. This has raised doubts as to the validity of fractal based compression. One criticism is that many of the example images supplied were from a very "restricted" class of images.

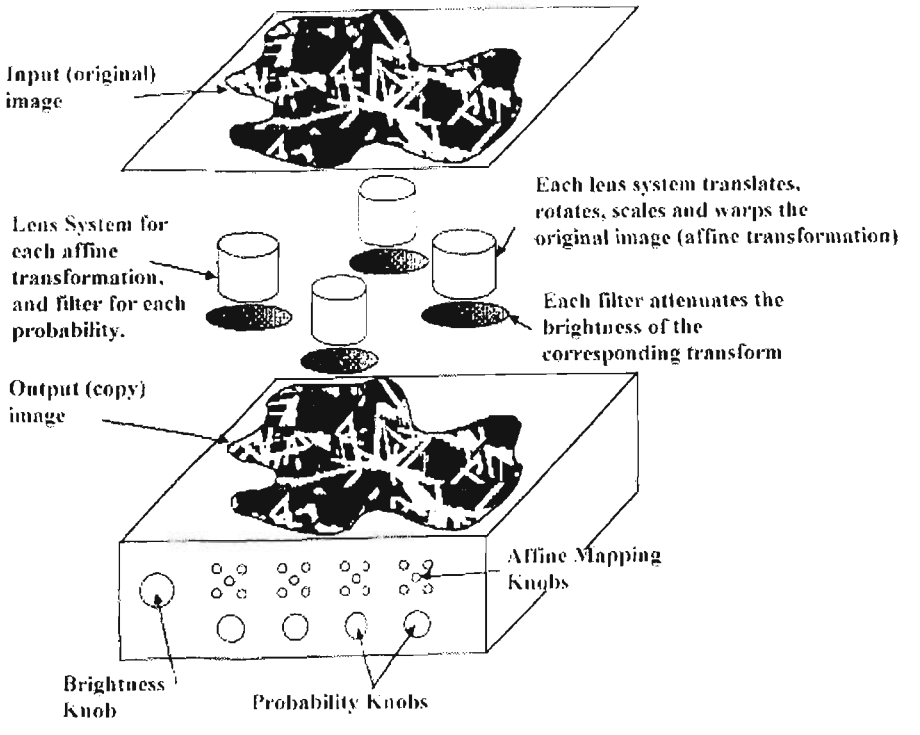


Figure 2.21 Barnsley's photocopy machine

[5] Iterated Systems, Inc., Atlanta, Georgia, U.S.A.

According to Barnsley et.al., there must be a set of basis functions or rules for the self-affine transformations to generate an image, and they are in principle discoverable [8]. Figure 2.21 shows Barnsley's photocopy machine [72] depicting this. This is known as his *college theorem*, and states that there is a set of mappings and probabilities that produce an output identical to the input to any desired degree of accuracy. These transforms and probabilities constitute his image compression algorithm (U.S. patent 5065447).

Jacquin, a former student of Barnsley, was the first to quantitively describe how fractals can automatically encode images [47]. This process begins by partitioning the original image into a set of N non-overlapping range blocks $\{R_j\}_{j=1}^{N}$, where the size of each block is 8×8 [47]. A domain pool $\{D_i\}_{i=1}^{M}$ is constructed by partitioning the image into a second, coarser set of M possibly overlapping blocks [16] [36]. Next, a class of contractive block transformations called the transformation pool, T, is defined. Each transformation consists of a value component called a *massic* [47], which alters the values of pixels in the block, and a geometery component which shuffles the positions of the pixels in the block. A transformation is contractive if and only if it brings every two pixels in a block both nearer spatially and closer in value.

For each image block, R_j, the encoding process searches for the pair (D_i, T_k) such that $T_k(D_i)$ best matches R_j. The quality of the match is determined by the distortion measure L_2 shown in (2.20). This measure sums the square of the difference between corresponding pixels of the two blocks X and Y with dimensions, $w \times h$ [16] [17].

$$L_2(X,Y) = \sum_{x=0}^{w}\sum_{y=0}^{h}(X(x,y)-Y(x,y))^2 \qquad (2.20)$$

One of the main disadvantages of fractal compression is that it is slow. Compression and decompression times are asymmetric. That is, it takes much longer to compress an image than it does to de-compress one. Figures from [77] based on a 386 PC give times of a few seconds for decompression, but up to 5 minutes for the actual compression process. It should be noted however, these times are calculated using software alone, and are based on outdated processors.

2.7.3 Comparison with JPEG

It is often difficult to compare the claims of different compression techniques, as so much depends on the actual implementation. Compression ratios and times can often differ greatly even between different implementations of same technique. Both wavelet and fractal compression are lossy methods, and for comparison should both be compared to the JPEG baseline sequential method.

Some compression performance results of the JPEG algorithm are given in [69]. For the lossy baseline sequential mode, nine images [69 : Chapter 15] with an average of 16 bits per pixel, were compressed with the average compression ratio at 20:1. These are the same images used in the original selection of the JPEG standard. This reported average compression ratio is typical of that reported by JPEG algorithms for very good decompressed picture quality.

The FBI target compression bit rate for fingerprint images discussed earlier, was 0.75 from original 8 bit data, corresponding to a ratio of 10.6:1. According to [11], extremely good quality was being achieved at a ratio of approximately 18:1, and good quality for the same images was reported at ratios of 26:1 [31]. The JPEG standard coding of these images was unacceptable due to artifacts at a ratio of 12.0:1 [11].

For fractal image compression Cochran [16] reported varying results of compression ratios depending on range block size. These results ranged from 10.99:1 to 729:1 for some medical imaging. The extremely high data compression results in most detail being obscured and only the edges being reproduced. Compression times greatly increased with compression ratios.

Randolf Shultz from the Computer Graphics Institute, at Rostock University, Denmark [76], provides comparative compression data on the same set of images for all three methods, JPEG, wavelets and fractal compression. The algorithms tested for comparison were, the JPEG implementation of the IJG (Independent JPEG Group) V5, the EPIC wavelet coder by Eero P. Simoncelli, and a fractal compression

algorithm called Coder, by Yval Fisher. Details of images at various compression rates can be seen in [76], and the results summary is presented here.

Table 2.4 presents the results from the comparisons of the three compression methodologies, and Table 2.5 presents the observed behaviour of the image artifacts which appear at higher compression ratios. The performance of compression times in Table 2.4 are measured as relative times with JPEG set at 1.

Table 2.4 Comparison of compression methodologies

Property	Criteria	JPEG	Wavelet	Fractal
max compression ratio	good image quality	1:30	1:30	1:25
	mediocre image quality	1:70	1:90	1:50
	poor image quality	1:160	1:350	1:375
Performance	compression time (rel)	1	5	20
	decompression time (rel)	1	3	5
	image dependence	medium	strong	very strong
	symmetric	yes	no	no

Table 2.5 Comparison of artifact behaviour of methodologies

Methodology	Appearance (ratio)	Effects
JPEG	1:119	Blocking artifacts Strong reduction in colours Edges destroyed by blocking
Wavelet	1:124	Overall image blurring Edges destroyed by excessive blurring
Fractal	1:38	Blocking artifacts Non-uniform overlapping blocks Small details disappear at low compression

It should be noted that the above comparison data is presented as is from [76]. However, the general trends are echoed further in the literature as outlined above. General results from wavelet compression reports that for the same quality image

higher compression than JPEG is available. The blocky features of higher compression JPEG images are spread more evenly in a wavelet compressed image, which may account for better subjective testing results. For general real-world images, the fractal compression seems to perform worse than its JPEG counterpart, despite some promising results in the literature. There is a direct dependence on fractal compression times with the compression ratio.

2.8 Summary

This chapter has discussed the basics of image compression. An overview of developments and methodologies leading up to the selection of the JPEG standard for continuous tone still picture image compression as the International Standard in 1992 has been included. There has been much work in the area of image compression since the selection of the JPEG standard, with wavelets and fractals being a part of this. These two techniques represent the better known recent developments and have been covered in detail.

The next chapter discusses the mechanics of one aspect of the JPEG standard. The JPEG sequential baseline mode of operation is then used in the implementation of a parallel algorithm in subsequent chapters.

CHAPTER 3

THE JPEG STANDARD AND PARALLEL SYSTEMS

3.1 Introduction

The JPEG standard is now the International Standard for the coding of continuous-tone still images, accepted by the ISO and CCITT. This research therefore, has adopted this standard for the implementation of the parallel image compression algorithms developed and discussed in Chapters 4 and 5.

This chapter gives a brief overview of the JPEG standard, and covers those aspects of the standard which are used in later chapters. Since a parallel implementation is developed, this chapter also provides an introduction to parallel systems and paradigms. An overview of parallel architectures and programming paradigms is provided only in enough detail to set the context of the research. Section 3.2 discusses the JPEG standard while Section 3.3 discusses parallel systems and paradigms.

3.2 JPEG Standard

The JPEG International Standard for image compression is defined in [52], and covers all aspects of still picture compression. Not all the standard applies to this research. Indeed the standard provides much latitude in some areas, and acts as a guideline to successful implementation. Many of the annexes provided in [52] are not an integral part of the recommendations but are provided as examples. Thus, there can be a wide difference between implementations of JPEG in applications said to be conforming to the standard. However, they must all satisfy the compliance testing outlined in [53].

The following sections discuss those aspects of the JPEG standard chosen for implementation in the development of a parallel algorithm. There were certain aspects which needed implementation as directed, but where there is latitude the choices made are clearly stated. Only those recommendations required by the research were implemented. Those that are required for successful compliance testing but not implemented in the research algorithms, are also identified in the following sections.

3.2.1 Lossless Vs. Lossy Compression

The JPEG standard specifies two classes of coding and decoding processes, lossless and lossy. The lossy class is based on the Discrete Cosine Transform (DCT) and permits considerable compression. The lossless class is based on Differential Pulse Code Modulation (DPCM), and is specified to satisfy the needs of those applications that may require the original image to be reconstructed exactly.

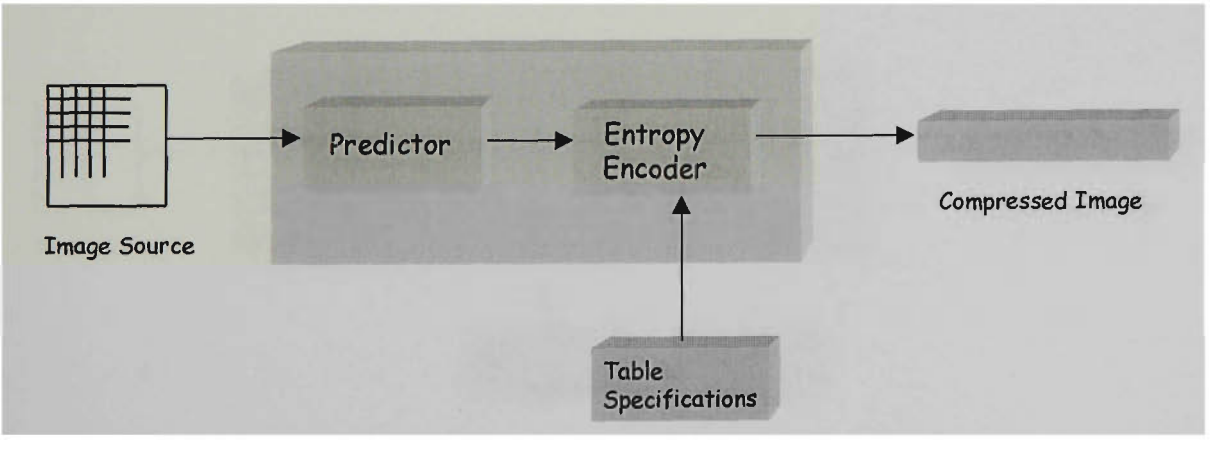


Figure 3.1 Structure of lossless encoder [52]

The structure of the JPEG standard lossless encoder is shown in Figure 3.1. The lossless coding process does not use the DCT, but is based upon a prediction using up to three neighbouring values *a*, *b*, and *c* to estimate a value for the current sample, *x*, see Figure 3.2 below. The difference between the prediction and the actual value for sample *x* is then coded using an entropy encoder. The lossless compression achieved

is not as great as the DCT based processes, and is around 2.5:1 [55] for DPCM techniques in general. A compression rate of approximately 2:1 [69] was reported for the JPEG version using arithmetic coding.

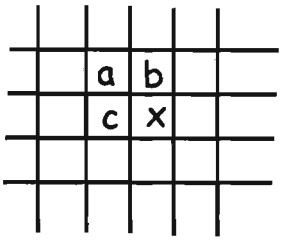


Figure 3.2 Prediction of pixel x from 3 neighbours

The structure of all lossy encoding processes (which are based on the DCT), is shown in Figure 3.3. This figure shows all the main procedures used during the lossy encoding process. The predictor in Figure 3.1 is replaced by a Forward Discrete Cosine Transform (FDCT) and followed by a quantization process. The quantization is responsible for most of the loss that occurs, though more may be introduced by discarding some of the DCT coefficients during the encoding stage. This is discussed later in section 3.2.2.

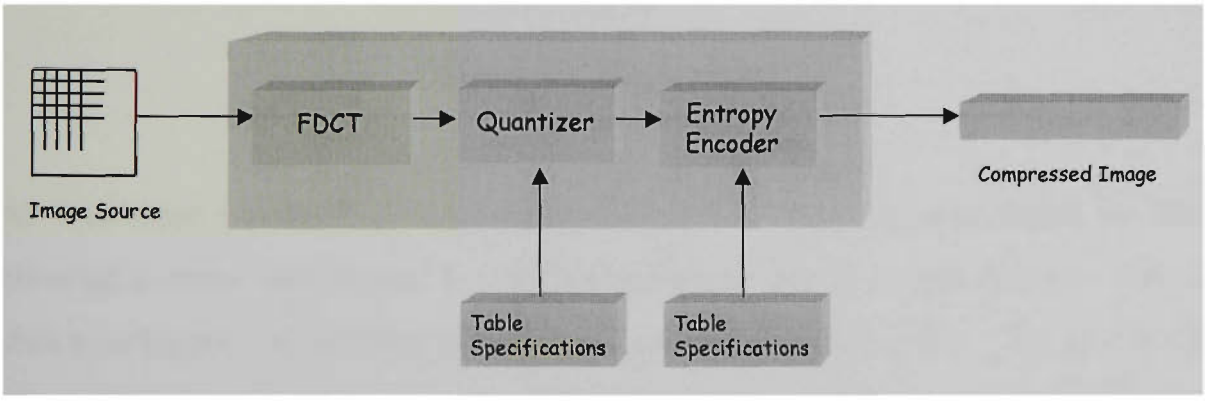


Figure 3.3 General structure of DCT based encoder

Most applications use lossy encoding as this provides far more compression, and some degree of control over the compression rate is possible. Also, within the lossy class of procedures, there are three distinct modes of operation, these are *sequential* DCT-*based*, *progressive* DCT-*based*, and *hierarchical*. The progressive DCT-based mode allows for the quick display of partially decoded coefficients, with progressive displays sharpening the image and providing more detail at each iteration. The

hierarchical mode also offers a progressive presentation, but is more useful in environments that have multi-resolution requirements [52]. The simplest of these is the first and is referred to as the *baseline sequential* process. This mode of operation is adequate for most applications.

This research uses the lossy baseline sequential encoding model of Figure 3.3. The specification, of which class of process and which mode within that class was used to encode the source image, is supposed to be stored as a *marker* code within the compressed image data. This item is omitted in the parallel implementation, as only the baseline sequential mode of operation is implemented.

3.2.2 DCT-based Compression Overview

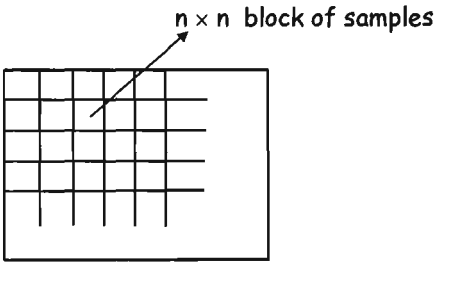


Figure 3.4 Image blocks

As with other transform compression techniques, an image is compressed by first breaking it down into smaller $n \times n$ pixel blocks (where n is typically 8 or 16), as shown in Figure 3.4, and then applying the transform to the blocks. The size of the image block specified in [52] is 8×8 and the research algorithms implemented use this size. All image blocks are now assumed to be 8×8 pixels in size unless otherwise stated. Each block is compressed separately. It is compressed by first converting the image samples, which are the intensities, into their frequency domain by application of the DCT. The result is an 8×8 block of transform coefficients.

The DCT maps the image block from its spatial domain to its underlying spatial frequency domain (the transform domain). It thus represents the image by using a set

of cosine basis functions, each with a particular spatial frequency. The basis functions are periodic with progressively higher frequencies. Each transform coefficient in the transformed block represents a scaling factor for its particular basis function. The complete set of basis functions are then scaled by their respective transform coefficients and then summed to give the original image block.

The number of basis functions (and hence the number of coefficients) depends on the dimension of the original block. When the image is divided into 8×8 blocks, the DCT can be computed independently for each block, since the cosine basis functions do not change. Only the transform coefficients change from one block to another.

Applying the one-dimensional DCT of (3.1) to a set of n spatial domain samples gives n transform coefficients as in Figure 3.5. Of these, one coefficient t_1 has special meaning. It is termed the *DC* coefficient, and represents the average frequency of the n point sample set. The remaining coefficients are termed the *AC* coefficients. They are coefficients for the basis functions of increasing frequencies. The descriptions DC and AC go back to when the DCT was used to analyze electrical currents with both Direct and Alternating components.

$$s_1, s_2, s_3, \ldots, s_n \xrightarrow[\text{forward DCT}]{\text{one dimensional}} t_1, t_2, t_3, \ldots, t_n$$

Figure 3.5 Application of one-dimensional DCT

Of all the transform coefficients, t_1 contains the most information used by the Inverse Discrete Cosine Transform (IDCT) shown in (3.2), in the reconstruction of the n point sample set. Coefficient t_2 contains the next largest amount of information and so on, down to t_n, which has the least amount of information of all the coefficients.

Dividing a digital image into 8×8 blocks gives us two-dimensional data samples. Applying a one-dimensional DCT to this two-dimensional data would yield coefficients that still display a high degree of correlation. For example, in an 8×8

block, applying the one-dimensional DCT across each of the eight rows of the block, results in an 8×8 block of transform coefficients, where the coefficients are uncorrelated horizontally across the rows of the block, but not vertically. In transforming to the frequency domain, the energy of the block is shifted to the coefficients in the first column, with no attention given to the vertical relationships between the original samples.

The two-dimensional DCT in (3.3), when applied to an 8×8 image block, produces an 8×8 block of transform coefficients uncorrelated both horizontally and vertically, as shown in Figure 3.6. A two-dimensional DCT is performed by applying a one-dimensional DCT across the rows of the sample block, and then down the columns of the partially transformed block. This removes the correlation in both dimensions.

$$\begin{matrix} s_{11} & s_{12} & s_{13} & \cdots\cdots & s_{18} \\ s_{21} & s_{22} & s_{23} & \cdots\cdots & s_{28} \\ s_{31} & s_{32} & s_{33} & \cdots\cdots & s_{38} \\ & & \vdots & & \\ s_{81} & s_{82} & s_{83} & \cdots\cdots & s_{88} \end{matrix} \xrightarrow[\text{forward DCT}]{\text{two dimensional}} \begin{matrix} t_{11} & t_{12} & t_{13} & \cdots\cdots & t_{18} \\ t_{21} & t_{22} & t_{23} & \cdots\cdots & t_{28} \\ t_{31} & t_{32} & t_{33} & \cdots\cdots & t_{38} \\ & & \vdots & & \\ t_{81} & t_{82} & t_{83} & \cdots\cdots & t_{88} \end{matrix}$$

Figure 3.6 Application of a two-dimensional DCT

In Figure 3.6, transform coefficient t_{11} is the DC coefficient, all the other coefficients are the AC coefficients. This is shown in Figure 3.7. As in the one-dimensional case, the DC coefficient represents the average energy for the block, and the AC coefficients represent coefficients for the basis functions of increasing frequency. In addition, the DC coefficient contains the most information of all the coefficients when used in the two-dimensional IDCT in (3.4), for reconstruction of the original image block. The AC coefficients, then contain decreasing amounts of information used by the IDCT following a zig-zag pattern, until coefficient t_{88} which contains the least.

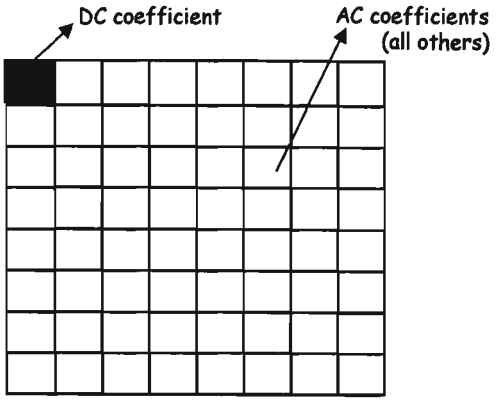


Figure 3.7 Transformed block coefficients

The two-dimensional DCT is the most efficient transform, apart from the optimal KLT, in energy packing ability. It moves as much of a block's energy into the upper left, transform coefficients as possible. The DCT is also very efficient at packing most of this energy into the DC coefficient. After transformation, the high frequency coefficients correspond to the very fine detail in the image block. They can either be coded with the rest, or discarded with little perceptible loss of detail. Quantization rounds many of these coefficients to zero. The AC coefficients towards the lower right of the compressed block in Figure 3.7 correspond to these higher frequencies. The basis functions of the two-dimensional DCT can be seen in Figure 3.8. The finer detail, and hence the higher frequencies of the transform can be seen increasing from the upper left of the block to the lower right.

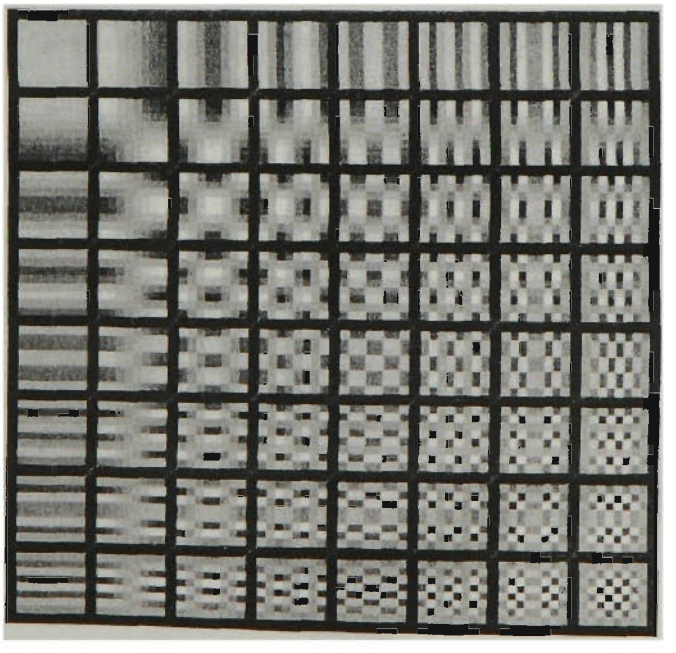


Figure 3.8 Two-dimensional 8×8 DCT basis functions [70]

The transform coefficients can be ordered in a linear fashion. The standard ordering used with the DCT is shown in Figure 3.9 and is called the *zig-zag sequence*. The order of importance is defined by the amount of information contained in each coefficient, and thus the contribution each makes to the IDCT. The sequence of Figure 3.9 arranges the coefficients in decreasing order of importance, thus the first coefficient (DC) makes the largest contribution to the IDCT. The zig-zag ordering is fundamental to the image compression of the JPEG standard and the pattern in Figure 3.9 approximately orders the two-dimensional cosine basis functions of Figure 3.8, from low to high spatial frequencies.

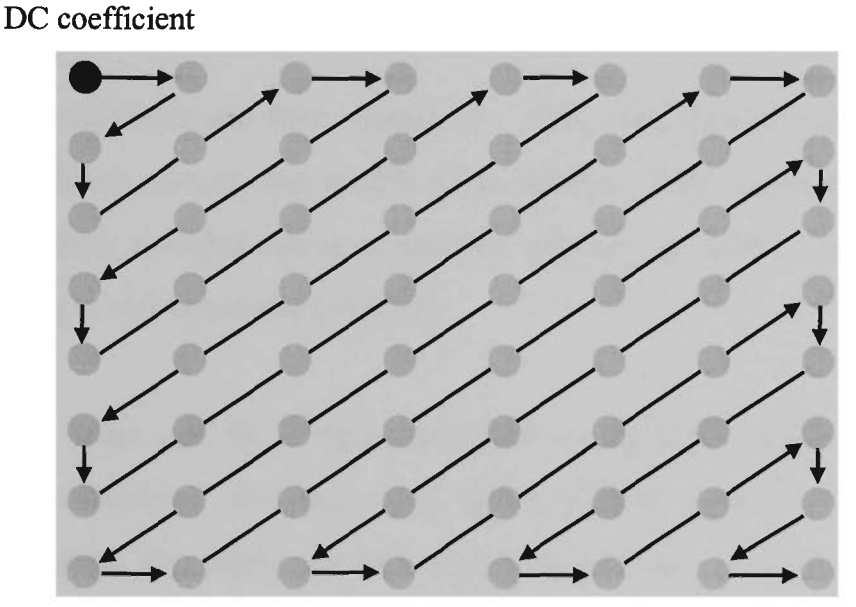


Figure 3.9 Zig-zag sequence for 8×8 block of transform coefficients

This ordering of the coefficients according to the zig-zag sequence affects the statistics of the symbols in the statistical model used by JPEG to encode the coefficients. The probability of the coefficients being zero in this ordering is an approximately monotonic increasing function of their index within the ordering [69]. This can be seen in Figure 3.10. Because of the high probability of coefficients being zero towards the end of the sequence, the encoding process can also use run-length encoding on these zero-coefficients. See Section 3.2.3.4.

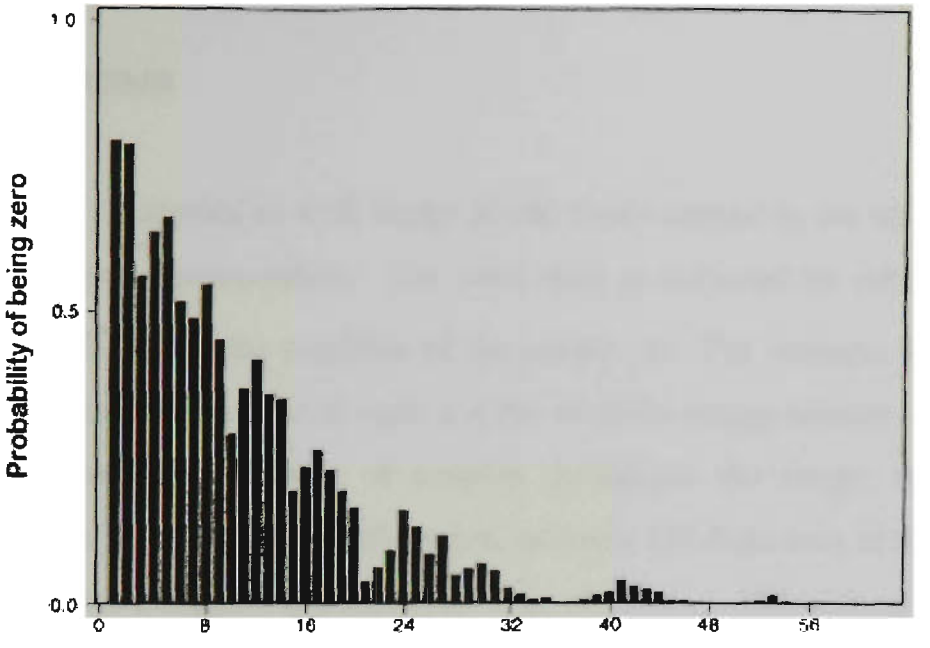


Figure 3.10 AC coefficient probability in zig-zag index [69]

The transform coefficients are then quantized. This step divides each frequency coefficient by a fixed amount and rounds to an integer. The quantized frequency coefficients are then encoded using a suitable entropy encoder. Arithmetic or Huffman encoding is recommended by JPEG.

The compression ratio can be further controlled by omitting some of the later coefficients in the sequence. In many cases, their omission results in no perceptible loss of image quality, depending on how many are omitted. The research in the following chapters does not omit any of these coefficients. All coefficients are encoded, with the only loss of data coming from the quantization process.

3.2.3 JPEG Steps

Once an image is decomposed into its corresponding 8×8 pixel blocks, four major steps in the JPEG algorithm are applied to each block. These are *level shift*, the *DCT*, *quantization*, and *entropy encoding*. Each of these steps are described in the next four subsections. Particular detail is given to the DCT step as a fast version of the DCT is substituted for the ideal functions presented in Section 3.2.3.2.

3.2.3.1 Level Shift

Before the DCT is applied to each image block, every sample in the block is level shifted to a signed representation. The level shift is achieved by subtracting the quantity 2^{P-1} where P is the precision of the sample set. For example, in eight bit grey scale images, the precision is eight and the intensity ranges between 0 and 255. Assuming a random distribution of samples throughout the image, the average intensity value is 128. The level shift process subtracts 128 from each of the intensity values. This gives a signed representation range of -128 ... 127 with an average of zero.

The level shift reduces the internal precision requirements in the DCT calculations. In principle, this level shift affects only the DC coefficient, shifting a neutral grey intensity to zero [69]. During image decompression, the reverse process takes place. After the Inverse DCT is performed, the value 2^{P-1} is added to each decoded value of the IDCT.

3.2.3.2 DCT

Next, each block has the two-dimensional DCT applied to it. The mathematical definitions for the DCT and IDCT below contain terms that cannot be represented with perfect accuracy by any real implementation [52]. The accuracy requirements for the DCT and the IDCT are specified in [53]. The function definitions for the 1-dimensional DCT and 1-dimensional Inverse DCT (IDCT) are

$$F(u) = \frac{1}{2} C(u) \sum_{x=0}^{7} f(x) \cos \frac{(2x+1)u\pi}{16} \tag{3.1}$$

$$f(x) = \frac{1}{2} \sum_{u=0}^{7} C(u) F(u) \cos \frac{(2x+1)u\pi}{16} \tag{3.2}$$

where $C(u) = \frac{1}{\sqrt{2}}$, *for* $u = 0$, $C(u) = 1$ *for* $u > 0$.

The function definition for the 2-dimensional DCT and the 2-dimensional IDCT are

$$F(u,v) = \frac{1}{4} C(u)C(v) \sum_{x=0}^{7} \sum_{y=0}^{7} f(x,y) \cos\frac{(2x+1)u\pi}{16} \cos\frac{(2y+1)v\pi}{16} \quad (3.3)$$

$$f(x,y) = \frac{1}{4} \sum_{u=0}^{7} \sum_{v=0}^{7} C(u)C(v)F(u,v) \cos\frac{(2x+1)u\pi}{16} \cos\frac{(2y+1)v\pi}{16} \quad (3.4)$$

where $C(u),C(v) = \frac{1}{\sqrt{2}}$ *for* $u,v = 0$; $C(u),C(v) = 1$ *otherwise*.

It should be remembered that in [52], (3.3) and (3.4) are the *ideal* function definitions. These functions do not have to be strictly applied, as long as the functions used in the resulting algorithm conform to the compliance testing for accuracy in [53]. A fast algorithm implementing the DCT is normally substituted for (3.3) and (3.4) in order to improve speed of computation. The ideal functions are computationally intensive. For example, the 2-dimensional DCT (3.3) applied to an 8 × 8 block (64 samples), requires 1024 real multiplications and 896 real additions [69].

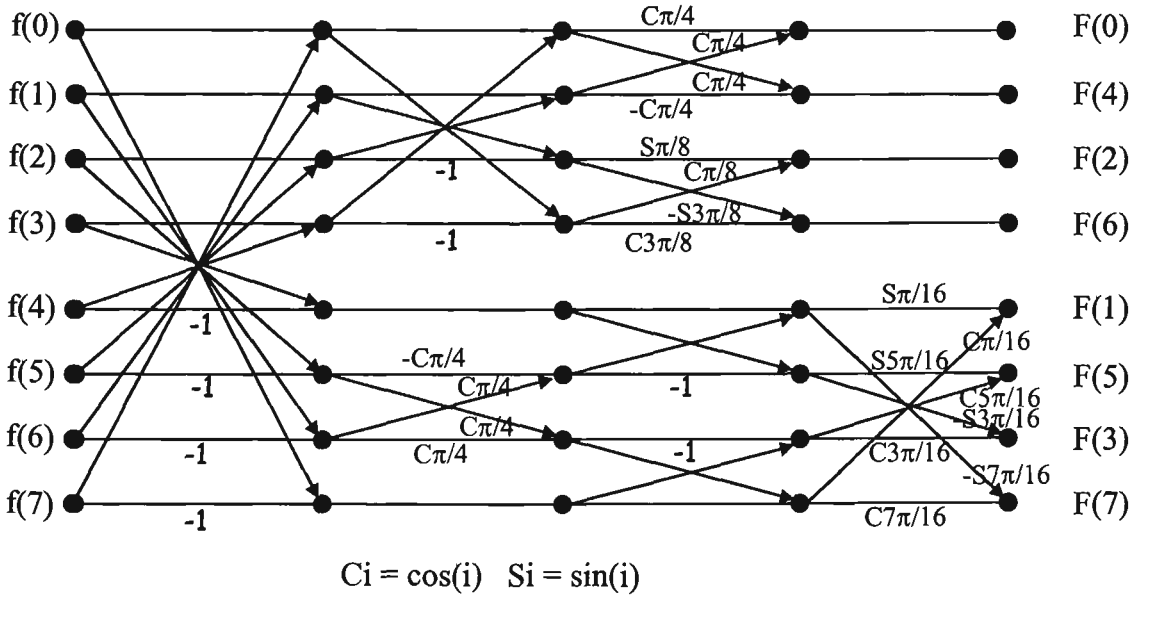


Figure 3.11 Signal flow graph for fast DCT [14], [70]

The research algorithms developed in Chapter 4 implement a fast version of the DCT developed by Chen et.al. in [14]. The signal flow graph for this algorithm can be seen in Figure 3.11 above. The algorithm implementation of this is listed in Section B3, Appendix B. This fast version of the DCT has less than 1/6 of the computational steps of the ideal DCT function [14]. While there are faster DCT versions, this algorithm is adequate for most implementations, with the only disadvantage being the complex index mapping [70].

3.2.3.3 Quantization

Next, the 64 coefficients are quantized. That is, they are divided and then rounded by using a corresponding value from an 8×8 quantization table as in

$$Sq_{uv} = round\left(\frac{S_{uv}}{Q_{uv}}\right).$$

Sq_{uv} is the quantized coefficient, and rounding is to the nearest integer. The quantization step allows the transform coefficients to be represented with much less accuracy, and hence use less storage space. In the JPEG standard, the values in the quantization table, and the resulting quantized transform coefficients are always integers. This integer representation is chosen so that with appropriate quantization values, the quantized coefficients are represented using less bits of precision than the corresponding input samples. This is an important step, as it tends to reduce many of the coefficients to zero, especially those representing the high spatial frequencies.

The JPEG standard does not specify any default values for the quantization table. This is left to the particular application in order to customize the table to the characteristics of its particular set of images. However, Annex K of [52], "Examples and Guidelines", does provide two example quantization tables for luminance and chrominance. These tables were arrived at empirically by measuring the visibility of

the 8×8 DCT basis functions on an image with luminance resolution 720×576 and a chrominance resolution of 360×576 at a viewing distance of six times the screen width. Threshold quantization table values were then derived and presented in the example tables of Annex K [52]. The quantization table for the luminance values are presented in Table 3.1.

Table 3.1 Luminance quantization table

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

The research algorithms use the quantization values in Table 3.1 for this step in the JPEG standard. The images used during testing were 8-bit precision grey scale (shown in Section A.2 Appendix A). Hence the use of the luminance quantization table was appropriate. While presented as examples only, the quantization tables in Annex K [52] can be used with satisfactory results. However some artifacts are noticeable, and it is reported [52], [69], that dividing the values given in Table 3.1 by two, will result in reconstructed images indistinguishable from the original. In Chapter 4, the quantization table values in Table 3.1 were used as presented, as can be observed in Section B.8 Appendix B.

3.2.3.4 Baseline Coder

The research algorithms implement the baseline sequential DCT mode of compression. The details of the encoder model for this mode of operation is shown in

Figure 3.12 [69]. This model shows the separation of the JPEG entropy encoder from the other processes, which includes a simple Differential Pulse Code Modulation (DPCM) encoder for the quantized values of the DC coefficients.

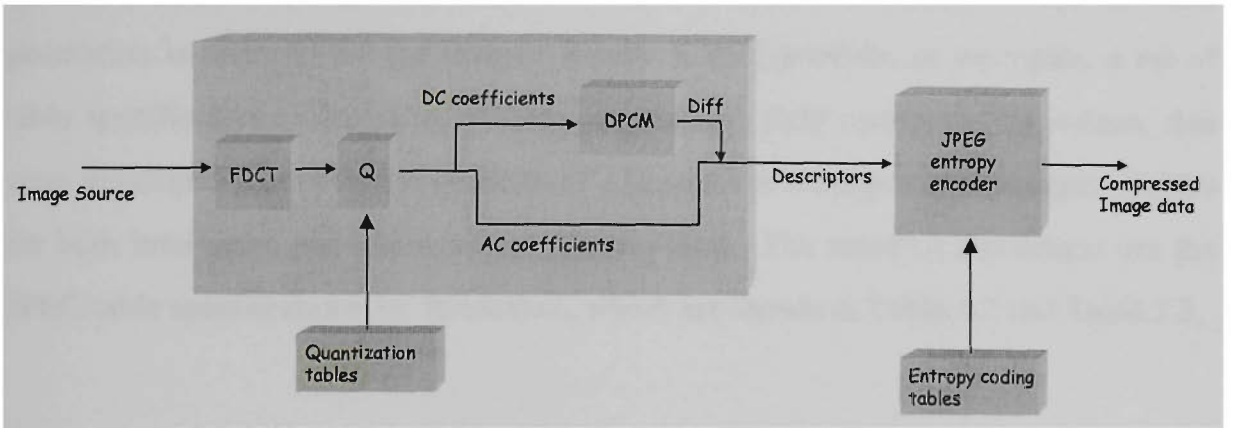


Figure 3.12 Baseline sequential-DCT encoder model

The DPCM encoder, whose model is shown in Figure 3.13, uses a one-dimensional predictor that is the DC coefficient of the previous image block. The previous image block is the block before the current one in the left-to right, top-to-bottom ordering of the 8×8 blocks from the source image. The DPCM encoder then returns the difference between these two blocks to be encoded by the JPEG entropy encoder.

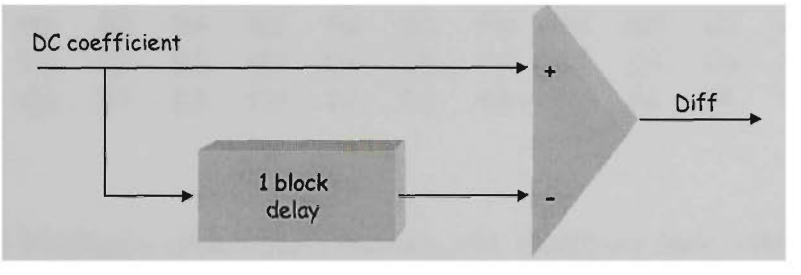


Figure 3.13 DPCM model for DC coefficient encoding

Entropy Coder

The JPEG standard allows one of two entropy encoding procedures to be used, either *Huffman encoding* or *arithmetic encoding*. The research algorithms developed in Chapter 4 use Huffman encoding. In the JPEG standard, when Huffman encoding is used, Huffman table specifications are passed to the encoding routine. The tables

contain a list of code lengths and their corresponding values, which are gathered from the statistics of the image to be encoded. With these tables, Huffman tables of codes and code lengths can be constructed from which the image can then be entropy encoded. These table specifications are only constructed if a custom Huffman table generation is required for the image. Annex K [52] provides as examples, a set of table specifications, consisting of code lengths and their corresponding values, that were developed from average statistics of a large set of 8-bit precision images. Tables for both luminance and chrominance are provided. The research algorithms use the JPEG table specifications for luminance, which are shown in Table 3.2 and Table 3.3.

Table 3.2 List of luminance code lengths for Huffman tables

00	02	01	03	03	02	04	03	05	05	04	04	00	00	01	7D

Table 3.3 Set of luminance code values for corresponding code lengths in Table 3.2

01	02	03	00	04	11	05	12	21	31	41	06	13	51	61	07
22	71	14	32	81	91	A1	08	23	42	B1	C1	15	52	D1	F0
24	33	62	72	82	09	0A	16	17	18	19	1A	25	26	27	28
29	2A	34	35	36	37	38	39	3A	43	44	45	46	47	48	49
4A	53	54	55	56	57	58	59	5A	63	64	65	66	67	68	69
6A	73	74	75	76	77	78	79	7A	83	84	85	86	87	88	89
8A	92	93	94	95	96	97	98	99	9A	A2	A3	A4	A5	A6	A7
A8	A9	AA	B2	B3	B4	B5	B6	B7	B8	B9	BA	C2	C3	C4	C5
C6	C7	C8	C9	CA	D2	D3	D4	D5	D6	D7	D8	D9	DA	E1	E2
E3	E4	E5	E6	E7	E8	E9	EA	F1	F2	F3	F4	F5	F6	F7	F8
F9	FA														

With the above Huffman table specifications, the Huffman code tables can then be constructed. The code tables are constructed using the procedure diagrams in Annex C [52]. These procedure diagrams are included in Section A.4 Appendix A. Once the Huffman code tables have been constructed, each of the 64 transform coefficients from the 8×8 transformed block are then arranged according to the zig-zag pattern in section 3.2.2 and encoded.

Each non-zero coefficient is described by an 8-bit composite value *RS* of the form

$$RS = 'RRRRSSSS' \tag{3.5}$$

The four least significant bits '*SSSS*' represent the category for the amplitude of the next non-zero coefficient. The categories of amplitudes are shown in Table 3.4. The value '*RRRR*' represents the number of zero-coefficients preceding this coefficient. In this way, the Huffman entropy encoder in the JPEG standard incorporates run-length encoding for runs of zero-coefficients. Since '*RRRR*' is only four bits, runs of zero-coefficients are grouped into runs of 15, with 15 zero-coefficients encoded with zero amplitude for the '*SSSS*' component.

Table 3.4 Amplitude categories for coefficients

SSSS	AC Coefficient
1	-1 , 1
2	-3 , -2, 2, 3
3	-7, ...-4, 4, ...7
4	-15, ...8, 8, ...15
5	-31, ...-16, 16, ...31
6	-63, ...-32, 32, ...63
7	-127, ...-64, 64, ...127
8	-255, ...-128, 128, ...255
9	-511, ...256, 256, ...511
10	-1023, ...512, 512, ...1023

The code table is represented by a pair of tables, one containing the code bits and another containing the length of each code in bits. Both of these tables are indexed by the composite value of *RS* in (3.5). Using these tables and the *RS* coefficient representation, the AC coefficients are then Huffman encoded. The encoding procedure diagrams for the Huffman encoding are specified in Annex F [52] and included in Section A.4 Appendix A.

There is also a similar set of tables and procedures specified in the appropriate Annexes of [52] for the separate Huffman encoding of the DC coefficients. The research algorithms have combined the encoding of the DC and AC coefficients into one set of procedures by slight extension of the procedures of the AC coefficient encoding. This just involves extension of the index on the program loops to include the DC coefficient. These changes can be seen in the Huffman encoding functions in Appendix B, section B.3.

The initial reasoning being, that with parallel execution of JPEG components on different processors, the Huffman encoding on one processor may not be aware of the value of the DC coefficient of the previous block. This block may have been processed on a different processor. Thus, the DC coefficient would not be able to be DPCM encoded. Thus for research purposes, the DC coefficient was encoded using the AC coefficient procedures. The only effect of this alteration to the encoding procedures is to slightly increase the size of the compressed image, as the DC coefficient would not be encoded as efficiently as possible. This is discussed further in Chapter 4 where a method for overcoming it is presented.

3.2.4 Compressed Data Formats

Annex B [52] describes the complete compressed image file data formats, which must be provided for successful compliance testing in [53]. This includes compressed image header and trailer information, frame headers and scan headers for different operational modes. These compressed file format headers and markers are required for complete implementation and interchange between different products implementing the JPEG standard. The research algorithms do not implement these compressed data formats as described in the Annex. They are required for operational products implementing JPEG, and were not of significance for the research.

3.3 Parallel Systems and Processing

Demands for faster processing speed and processor capacity have been occurring over the last three decades as the range of computer applications increases. The complexity of problems tackled has also increased as well as the sophistication of our problem solving techniques. While the increase in power and capability of computer systems has had definite gains over the last few decades, there are physical laws which will eventually become obstacles. For instance, no signal can travel faster than the speed

of light, which indicates that there is a definite upper limit to the speed of processing that we can expect from a single processor in the future.

The continual reduction in size of computer components has helped to shorten processor communication path lengths and thus reduce signal transmission times, but again limits are being encountered with respect to size. A recent example of this was the later than expected introduction of the Pentium[1] processor due to efforts to dissipate excess heat. While this problem was successfully overcome, physical size limitations will eventually slow development efforts. One strategy to overcome the speed limitations of serial processing is parallel processing.

Parallel processing is the utilization of multiple processors, simultaneously working on one problem [39]. The construction of parallel computers began over 30 years ago with the Burroughs[2] D825 computer using four processors. Parallel systems architecture and programming techniques have been evolving ever since. The initial hope was that if a single processor could generate say *X* floating point operations per second (FLOPS), then 10 processors would generate *10X* FLOPS, and so on. While this is theoretically possible, the reality is that practical considerations such as arrangements of processors, interconnection, synchronization and speed of connections all play a part in reducing this theoretical maximum speed increase. While there are speed gains, it is not a linear gain when mapped against the number of processors.

The range of applications for parallel processing has proven to be much broader than expected, and the following general categories of applications for parallel processing are listed [39], [59], [40]:

- General
- Numeric
- Signal Processing

[1] Pentium is a trademark of the Intel Corporation

[2] Burroughs Corporation

- Image Processing
- Graphics
- Database Processing
- Artificial Intelligence
- Simulation
- Optimization

The results of a survey [59] showing the distribution of parallel processors among various applications listed above are shown in Figure 3.14. The areas of image processing, graphics and signal processing, which are closely related, constitute a large slice of these applications.

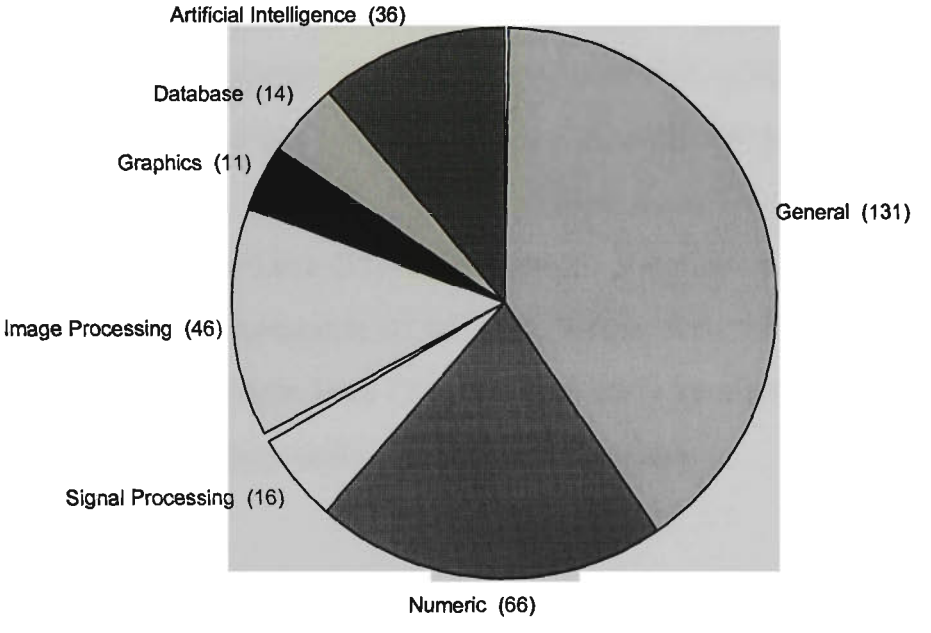


Figure 3.14 Distribution of parallel systems among applications

The use of parallel processing techniques in image processing, for a reduction in execution time, is not new. However, current literature reveals little on the application of parallel paradigms to the JPEG standard. One of the most recent papers with direct relevance to this research is [85]. However this is not concerned with the JPEG standard, but rather a particular programming paradigm, called the processor-farm, that is discussed in a later section.

The following sections explore the architecture of parallel systems and their specific applications, and the programming techniques and paradigms used to program these applications. These sections provide background for the research and algorithms developed in Chapters 4 and 5.

3.3.1 Supporting Architecture

Despite many attempts to classify the architectures of parallel computers, the classification scheme of parallel systems used by Flynn [26] is best known and still widely used. This scheme identifies two characteristics, instruction flow and data flow, and divides parallel systems into those that possess single flow or multiple flows of these characteristics. The classification scheme is shown in Table 3.5. Any system which has at least one multiple stream is classed as a parallel processor. However, according to Lerman and Rudolph [59], no system designed to date matches the Multiple Instruction Single Data (MISD) definition. As a consequence, most papers and books use the broad categories of Multiple Instruction Multiple Data (MIMD), and Single Instruction Multiple Data (SIMD), to classify parallel processing systems. These classifications are discussed in the following sections.

Table 3.5 Flynn's parallel systems classification scheme

		Data	
		Single Stream	Multiple Streams
Instructions	Single Stream	SISD	SIMD
	Multiple Streams	MISD	MIMD

3.3.1.1 MIMD vs. SIMD

In MIMD parallel systems, there are several processors, each processing their own instructions and working autonomously. When used to run parallel programs, the processors in these machines work simultaneously on different parts of the problem.

Thus the term, multiple instruction multiple data. This is regarded as a decentralized, control driven architecture.

In SIMD parallel systems, there are many processors all performing the same instruction on different data sets. Hence, single instruction multiple data. The parallel programs that run on these machines are programs for applications where large sets of data must all have the same processing performed on them. Many image processing applications fall into this class. Parallel processors with this architecture generally have more centralized control.

A mixed mode architecture concept, PASM (Partitionable-SIMD/MIMD) architecture, is being developed at Purdue[3] University [92]. This is a design for a large scale dynamically reconfigurable parallel processing system based on commodity microprocessors. Each processor can independently perform mixed-mode parallelism.

The decentralized control of the MIMD architecture offers far greater flexibility. It has the ability to run different streams of control, as opposed to only one in the SIMD architecture. However, the flexibility gained is diminished by other problems. In MIMD systems where the flow of control and process synchronization are specified by the program, scalability of the system can be a problem. Also when there are sections of program code which require each processor to be programmed individually, then the maximum number of processors which can be used is limited. This limitation is usually caused by a programmer's inability to effectively track the many operations being performed concurrently on many processors.

By contrast, the more centralized control of SIMD systems means that process communication and synchronization is usually done automatically. This helps provide for better scalability. However, the relative rigidity of these systems limits their uses.

[3] Purdue University - West Lafayette, Indiana, USA

3.3.1.2 Memory Organization

Another major distinction between the MIMD and SIMD architectures is whether they employ *shared* or *distributed memory*. When using shared memory, all processors in the system must have access to all parts of memory. When using distributed memory, each processor has its own local memory. Typically, MIMD parallel processors share memory, while SIMD parallel processors employ distributed memory [40]. The type of memory, (i.e. distributed or shared), also dictates the data exchange mechanism between processors.

Parallel processors using shared memory communicate by manipulating data values in memory accessible to all processors. Synchronization and locking mechanisms must be employed to ensure fair and consistent access for all processors to the shared memory. When distributed memory is used, processors must communicate by transmitting and receiving data over communication channels connecting the processors. This mechanism is called *message passing*.

Message passing is a simpler concept to handle from an engineering viewpoint, and offers an advantage in terms of data rates achieved since overhead can be amortized by the use of longer messages. However, shared memory offers greater flexibility as message passing can also be implemented in the software, by the use of message buffers and access routines. The drawback is the cost of communications, as access to all shared data must be regulated by use of locking and semaphores. These systems are more complex from an engineering viewpoint, as all processors must have access to all shared memory components which increases the communication network involved as more processors are added.

The type of memory access also plays a role in determining the connectivity of the parallel system. For example, Figure 3.15 shows some examples of distributed memory architectures, and Figure 3.16 shows some examples of shared memory architectures [39]. As can be seen in Figure 3.16, in all shared memory architectures,

all processors must somehow be linked to all memory elements, whereas with message passing, a wider variety of connection architectures can be employed. When two processors that communicate using message passing are not directly linked, other processors need to route messages.

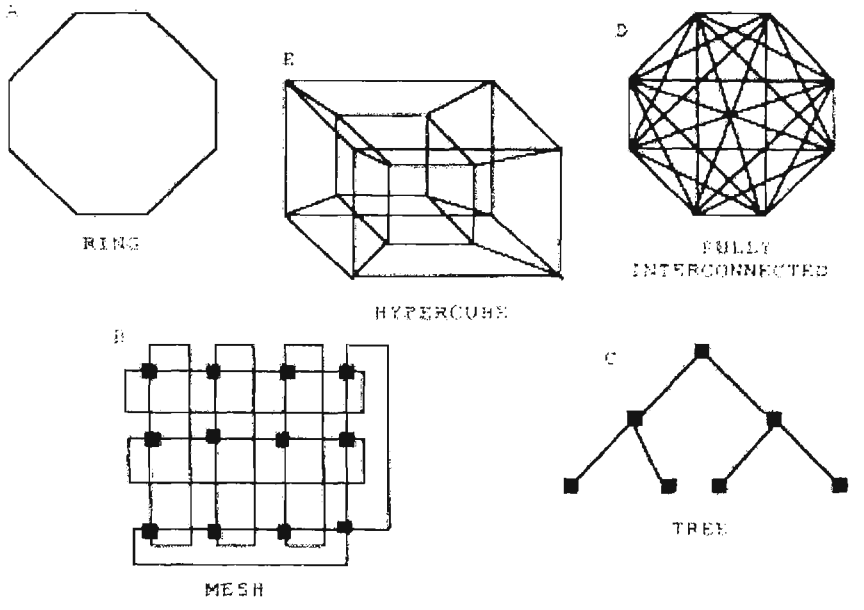


Figure 3.15 Some distributed memory architectures

According to studies by Lerman and Rudolph [59], message passing architectures have become more dominant with time. In the pre-1975 period, message passing was only associated with SIMD parallel processors, and these systems only comprised a small percentage of the parallel systems. Since 1985, message passing has become the more popular mechanism for both architectures.

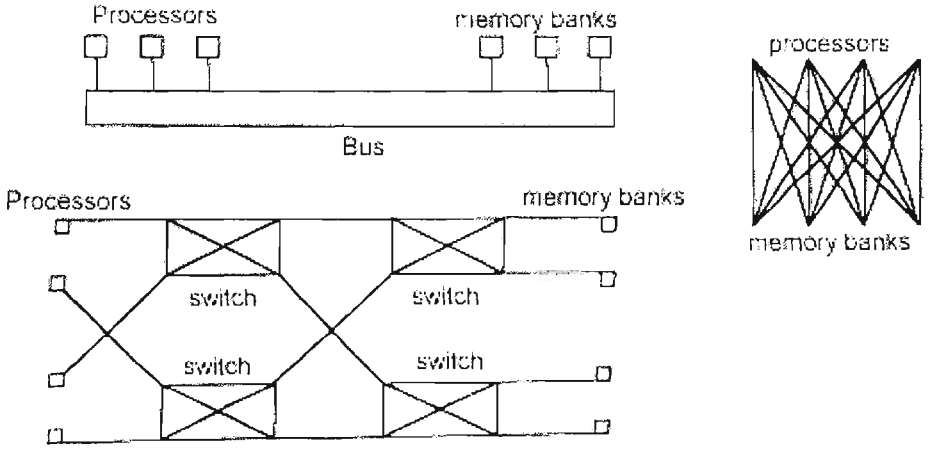


Figure 3.16 Some shared memory architectures

There are a number of factors for the emerging dominance of message passing, even among the MIMD architecture systems. Many of the original problems of message passing related more to software and were not addressed until after the work of Hoare in his article "Communicating Sequential Processes" [38]. This was a landmark on the treatment of the problems particularly facing MIMD type architectures at that time. Another reason for message passing gaining in popularity is related to scalability. As more processors are added to a system, more complex communication networks must be used to maintain connectivity. Message passing architectures are more amenable to scaling since this model contains almost no inherent contention for shared resources.

During the 70s the construction of message passing computers lagged behind that of the shared memory parallel systems. The situation has now been reversed somewhat, mainly due to the introduction of the Transputer in the early 1980s [59].

3.3.1.3 The Transputer

In the early 1980s the Transputer was introduced by Inmos[4] as a new concept in VLSI architecture. It is a single circuit containing a processor, local RAM and four input/output ports [84]. This circuit was a computer by definition, containing the processor, some memory to store code and data and several I/O channels for exchanging data. However, these circuits were designed so that they could be connected together with the same simplicity that transistors could be. In fact, the name *transputer* is combination of the words *transistor* and *computer.* A diagram showing the main features of the transputer can be seen in Figure 3.17.

[4] Inmos is a trademark of the INMOS group of companies

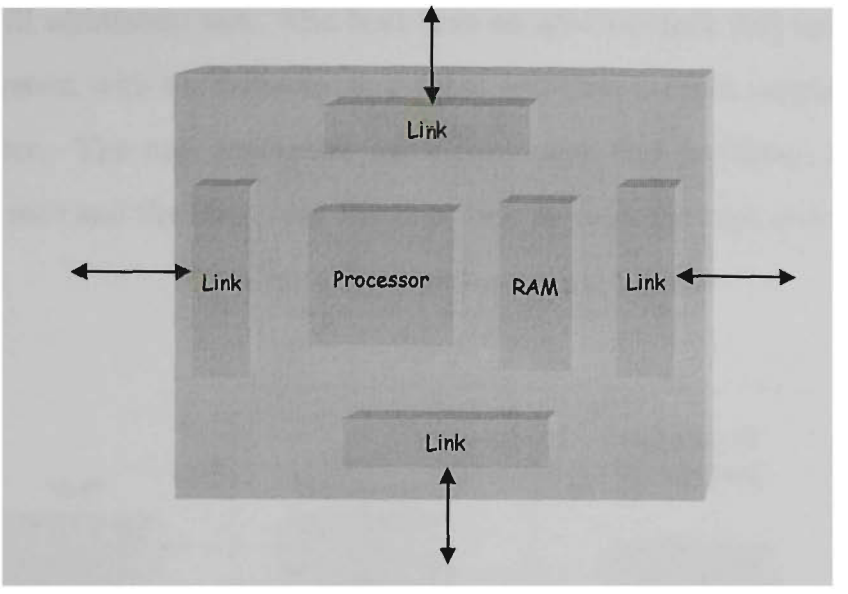


Figure 3.17 Generalized diagram of a transputer

The transputer was an important innovation. Although combining these elements onto a single chip did not represent a technological breakthrough, it was the novelty of the combination and its implications that aroused interest. A number of factors combined to make this innovation a milestone for parallel systems development. The introduction of the occam[5] programming language, which enabled an algorithm to be described as a collection of concurrent processes which communicate through channels, was one factor. One of the design objectives of occam was to use the same concurrent programming techniques for a single processor system, as for a multi-processor system. The second factor was the ease with which transputers could be connected together. The four I/O ports were designed to interface directly to other transputers, so they could be stacked, or placed closely together, enabling several transputers to fit together on a small footprint. Boards were constructed to hold many transputers, which could be added into a PC as an add-in board. Thus, the transputer became a parallel system, building block.

The general form of a transputer system connected to a PC via an add-in board is shown in Figure 3.18. The add-in board contains a network of inter-connected transputer modules of which one is called the *root* transputer. The root transputer is connected to the PC host, which in turn controls all the peripheral devices that the

[5] occam is a trademark of the INMOS group of companies

transputer will ultimately use. The host runs an *afserver* task that initially loads the transputer system with the software to be run, and then accepts instructions from the root transputer. The root transputer has a *filter* task that facilitates communication between the root and the host. All I/O is passed through the root and then ultimately to the host.

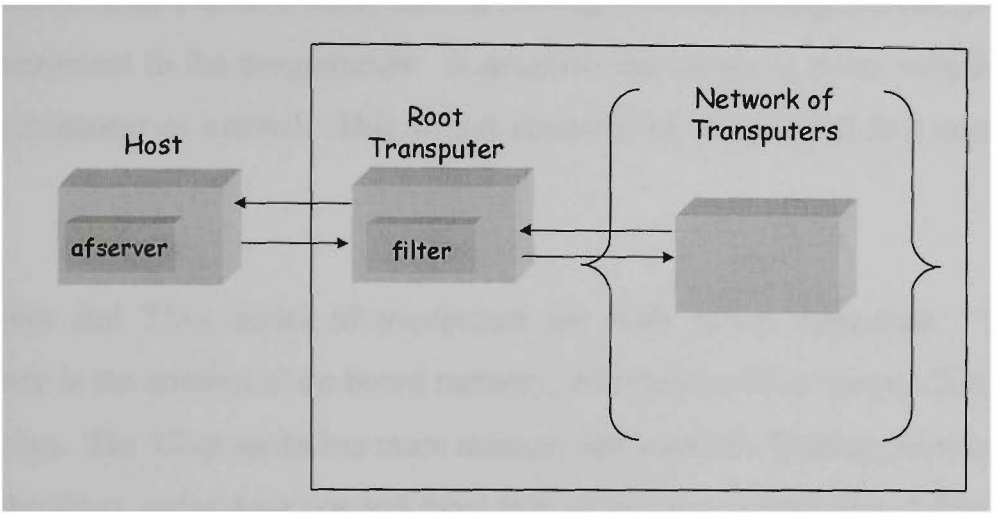


Figure 3.18 General form of transputer system connected to PC host

Transputers communicate via the use of high speed *serial links*. As can be seen in Figure 3.17, each transputer can have a maximum of four physical links. The small number of wires and the network protocols used makes networking the transputers relatively easy. In most cases, the configuration is user-configurable. Some transputer configurations can be seen in Figure 3.19; often the linear chain (a) is used as it is the simplest to implement.

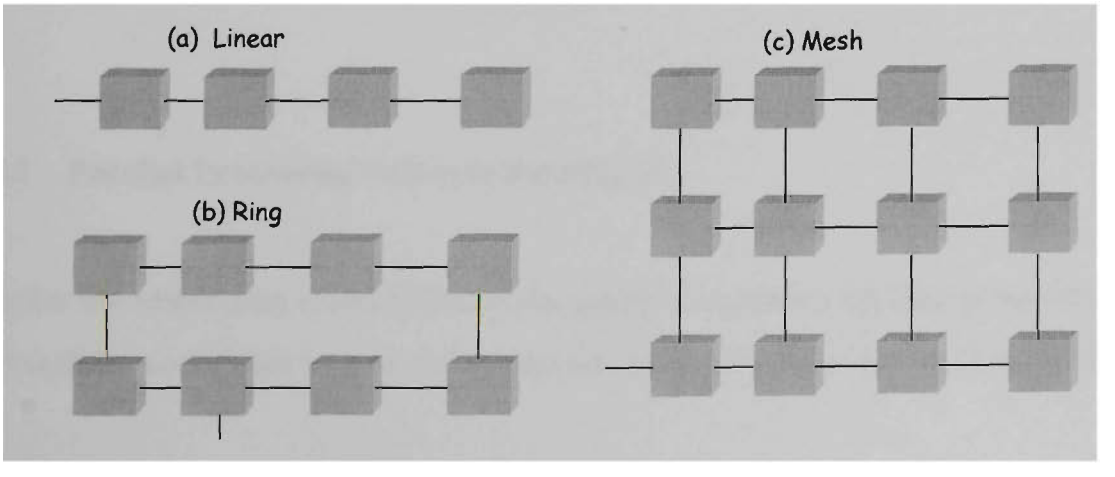


Figure 3.19 Some examples of transputer topologies

To help overcome the limited connectivity imposed by the 4-links per processor, the networking software allows the creation of a virtual channel between two processors. A virtual channel is a logical link between two processors where no physical link exists, but a path can be found using intermediate processors. Therefore, even if they are not connected, two processors can be logically treated by the programmer as if an I/O channel exists between them, and the routing software manages message delivery. It is transparent to the programmer. A program can create as many virtual channels from a processor as needed. This virtual connectivity is specified in a configuration file.

The T4xx and T8xx series of transputers are both 32-bit processors. The main difference is the amount of on-board memory, and whether they contain floating point processors. The T8xx series has more memory and contains floating point processors, while the T4xx series does not and must rely on software emulation of floating point arithmetic. While these series of processors are now quite old, Inmos announced early 1993 the introduction of the T9000 transputer. At the time, the T9000 was the world's fastest single chip computer, and is still quite a sophisticated system. Today the transputer has become a generic term, with many manufacturers providing TRAMS (transputer modules) to standard sizes. For example, Texas Instruments[6] manufactures a TMS320C40 chip, which is like a T9000 transputer but has other features, including 6 serial I/O channels. Alta[7] manufactures CTRAMS, which follow the mechanical and electrical standards for size-1, size-2 and size-5 TRAMS and are compatible with Inmos and other transputers.

3.3.2 Parallel Processing Software Paradigms

Despite the underlying architecture of the parallel computer, parallel programs must successfully coordinate two or more program tasks to ensure correctness and higher

[6] Texas Instruments Corporation, http://www.ti.com/

[7] ALTA Technology Corporation, http://www.altatech.com/

speed. Exactly how parallelism is controlled is largely determined by the paradigm used by the programmer. Many paradigms can be found in papers and articles, and there tends to be general agreement [10] [60] [13] [65], that these fall into two broad categories, *homogeneous* and *heterogeneous* parallelization, terms taken from [10]. The corresponding terms generally found in the literature are *data-parallel* and *control-parallel*. The former terminology is the author's preference.

The philosophy behind these two categories is discussed in the following sections.

3.3.2.1 Homogeneous Parallelization

Homogeneous parallelization is possible when the work to be done by the program can be partitioned into identical (homogeneous) sub-tasks to be performed on different parts of the original problem. These can then be run on different processors in the system. For instance, if each element in a large set of data has to have the same operations performed, then different elements of the input data set can be processed by identical sub-tasks on different processors. This situation is depicted in Figure 3.20.

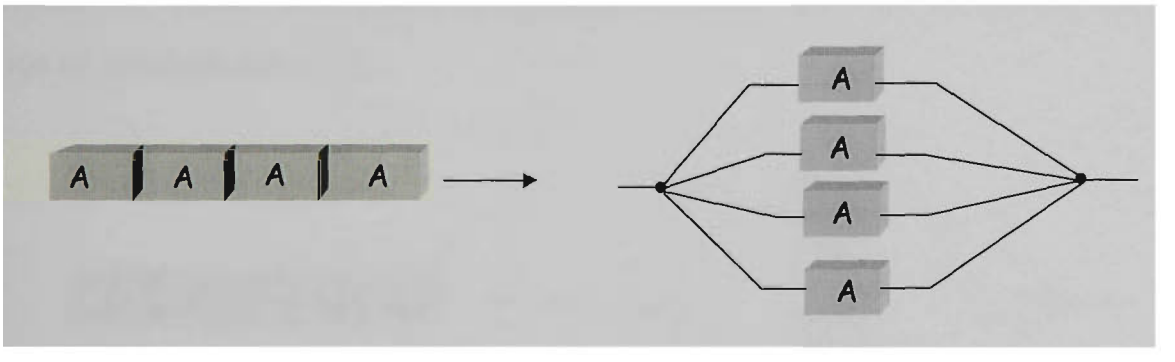


Figure 3.20 Homogeneous parallelization [10]

This is also known as *data-parallel*, as the inherent parallelism is determined by the data. The data source is broken into smaller subsets and exactly the same processing performed on each subset. This is the same philosophy underlying the architecture of the SIMD parallel systems. However, this parallel paradigm can also be applied to a general purpose message passing MIMD system, such as a transputer. A well known

example of this type of paradigm is the *processor-farm* paradigm, which is discussed in a following section.

Many image processing tasks are well suited to this parallel paradigm, since quite often the same processing is applied to different parts of the image. The JPEG compression standard is a prime example, as the image is decomposed into 8×8 blocks, and each block is transformed and encoded independently of the other blocks. As part of the research, a parallel algorithm is developed for the JPEG standard using this paradigm.

3.3.2.2 Heterogeneous Parallelization

Heterogeneous parallelization is possible when the work to be performed can be separated into different subtasks. These different subtasks can then be placed on different processors and run in parallel. This type of processing is depicted in Figure 3.21. The only requirement placed on code within each block is that it must be data-independent of the others. A loop, prevented from homogeneous parallelization because of internal dependence relationships between data, may be suitable for this type of parallelization [10].

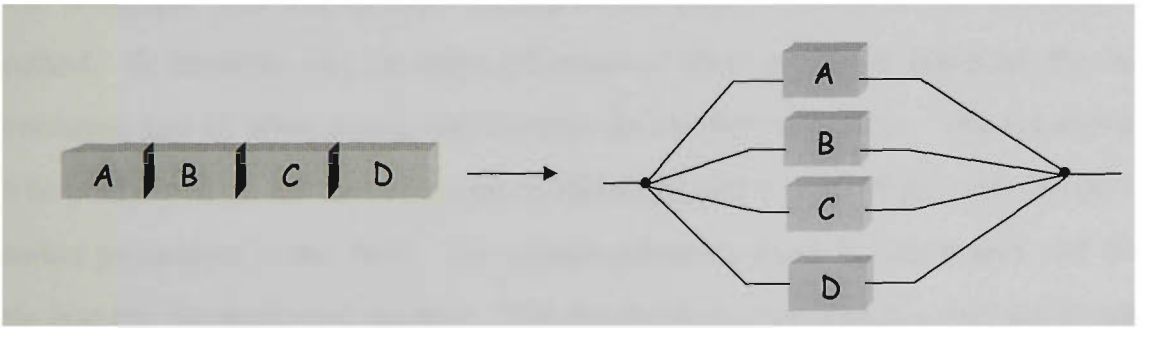


Figure 3.21 Heterogeneous parallelization [10]

Some variations on this general paradigm are referred to as the *result*, *specialist* and *agenda* paradigms in [13], and these seem to correspond to some paradigms proposed by other authors. A well known example of the specialist type of parallelization is that of the pipeline. A pipeline can be employed where a task is to be performed

repeatedly on a sequence of data elements, and this task can be broken into subtasks. Parallelization can then be introduced by placing the subtasks on separate processors, and passing the results of one processor on to the next as in an assembly line.

Because of the nature of the JPEG baseline algorithm, a pipeline approach can also be used to introduce parallelization into any algorithm implementing the standard. However, because of the type of processing involved this approach would not be optimal. The reasons for this are explained in section 4.7.1 of Chapter 4.

3.3.2.3 Processor Farm Paradigm

The distinction between homogeneous and heterogeneous parallelization is clear. Homogeneous parallelization describes the distribution of repetitive work over a number of processors. Heterogeneous parallelization describes the distribution of independent components of an algorithm over multiple processors [10]. Usually the case for homogeneous parallelization is easily recognizable by the presence of distinct repetitive code blocks.

One technique that implements homogeneous parallelization is the processor-farm method. It involves two or more processors. One processor becomes the *master* processor, and all other processors become the *worker* processors. The repetitive task to be performed on all data elements is identified and a copy of this task is sent to all worker processors in the farm. The master processor reads the input data and divides this into the discrete work packets. The master processor then has two major tasks to perform: scattering the work packets to the farm, then gathering the results of the worker processors for collation and storage. The job of the worker processors is to receive a work packet, perform the repetitive task, send the processed work packet back to the master processor and wait for more work. This is depicted in Figure 3.22.

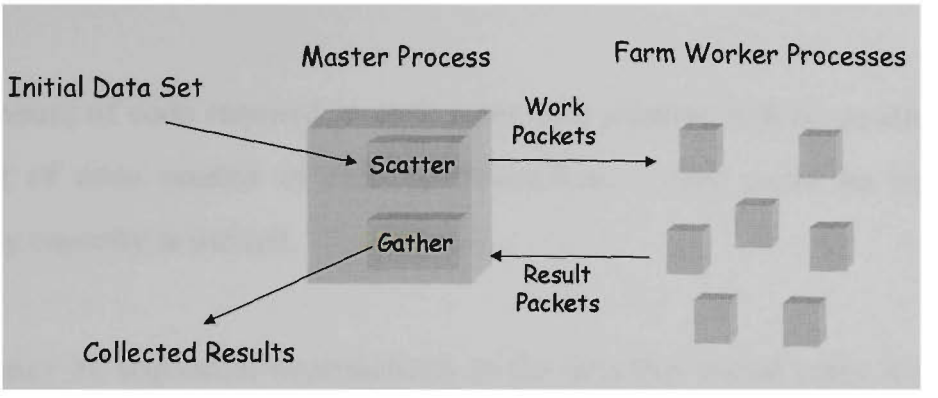


Figure 3.22 Processor farm methodology

The networking software that comes with the transputer allows for the easy specification of a processor farm configuration. When implemented correctly, the processor farm is very good at achieving a good load balance on the processors in the farm. This is important for the reduction of overall execution time of the algorithm.

Quite often when a processor-farm type algorithm can be employed, a pipeline can also be considered. It was mentioned above that either a processor-farm or a pipeline methodology could be used to gain parallelization in the JPEG standard. The following is a brief comparison of the two techniques [44]:

- The throughput of a pipeline is limited by the throughput of the slowest part. That is, in an *n-stage* pipeline

$$pipe\ processing\ time = n \times max(\delta t(i)) \quad (i = 1, \ldots n),$$

where $\delta t(i)$ is the time taken for stage i of the pipeline. The processing time of the equivalent sequential implementation is

$$sequential\ processing\ time = \sum_{i=1}^{n} \delta t(i).$$

There will be some extra processing time in passing messages between processors in a pipeline. Equivalent equations for the processor-farm method are developed in Chapter 4.

- The amount of code required in each stage of a pipeline will be smaller than the amount of code needed in each processor-farm. This could be important if memory capacity is limited.

- There may be sequential dependencies in the data that would make it difficult to use the processor-farm methodology.

The research algorithms implement a parallel JPEG version using the processor-farm paradigm on a transputer.

3.3.3 Performance

As always, the question of performance is an important one in parallel processing. The reason behind parallel programming is to gain speed advantages over the corresponding sequential algorithm. Performance really depends on both the underlying architecture and the software paradigm used. However, there are some models of performance that can be generally discussed.

Performance is generally measured in terms of *speedup*, which is a comparison of the execution time of the sequential algorithm running on one processor to the parallel version running on n processors, [60] [84]. Thus

$$speedup = \frac{\delta t_1}{\delta t_n}. \tag{3.6}$$

Efficiency (E) is a measure of how well the parallel processors are utilized and can be measured by

$$E = \frac{speedup}{n} \tag{3.7}$$

The ultimate goal is to increase the number of processors n, keeping the efficiency at or near one. However, this is not a practical expectation, as the speedup is limited by *Amdahl's law* (3.8) [60]. This law states that the scaling performance of a parallel algorithm is limited by the fraction of inherently sequential code in that algorithm [92]. Therefore, if an algorithm contains a fraction f of work that must be performed sequentially, then the speedup possible on a machine with n processors is:

$$speedup = \frac{1}{[f+(1-f)/n]} \tag{3.8}$$

While Amdahl's law might accurately model algorithms based on the heterogeneous paradigm, it does not properly model algorithms based on a homogeneous paradigm. This is because it ignores the parallelism that might be exploited in the data to be processed. It overlooks the possibilities of SIMD type computation where potential parallelism increases with the number of available processors. Amdahl's law is pessimistic for this type of parallelism. For homogeneous parallelization, (3.8) can be modified according to [60], to arrive at the *Gustafson-Barsis law*:

$$speedup = n-(n-1)f \tag{3.9}$$

Homogeneous paradigms or data-parallel methods are more efficient. They maximally utilize all processors and can achieve a speedup proportional to the size of the data. That is according to the above laws. Figure 3.23 shows a comparison of Amdahl's law and the Gustafson-Barsis law as a function of the sequential fraction f of a program. Both of these laws omit one important thing, and that is communication overhead, which increases as the number of processors in the parallel system increases.

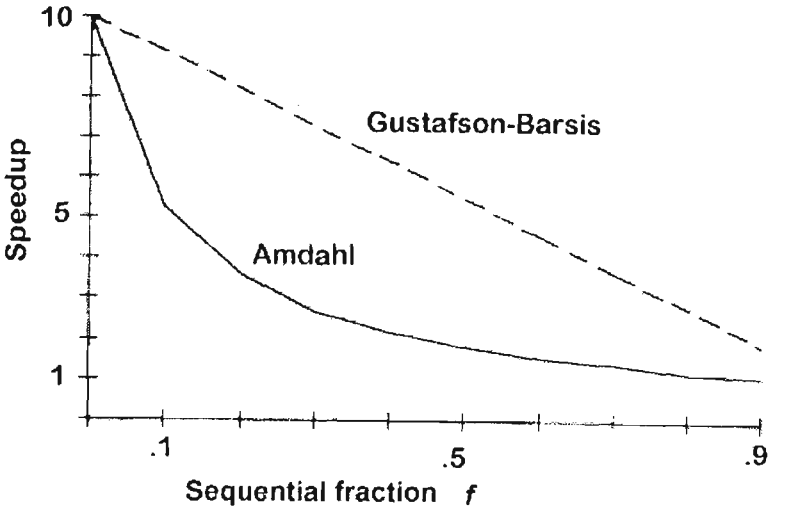


Figure 3.23 Amdahl's law vs. Gustafson-Barsis law

As an example of these limitations, [84] provides data showing the execution times for a program which displays the *Mandelbrot set* on a VGA screen at resolution *640* × *480* pixels. The details of the algorithm can be referred to in [84]. The data in Table 3.6 shows the execution times of a parallel algorithm used to display this Mandelbrot set using from 1 to 6 transputers.

Table 3.6 Execution times for displaying Mandelbrot set

Number of Transputers	Execution time (seconds)
1	99.230769
2	60.274725
3	49.285714
4	43.681319
5	40.714286
6	41.978022

The line graph displayed in Figure 3.24 shows the speedup curve for the parallel algorithm whose execution times are given in Table 3.6. The top straight line shows the upper bound for speedup assuming the fraction of sequential code is zero. The bottom curve represents the actual speedup observed by using the value of 99.230769 seconds for δt_1 in (3.6).

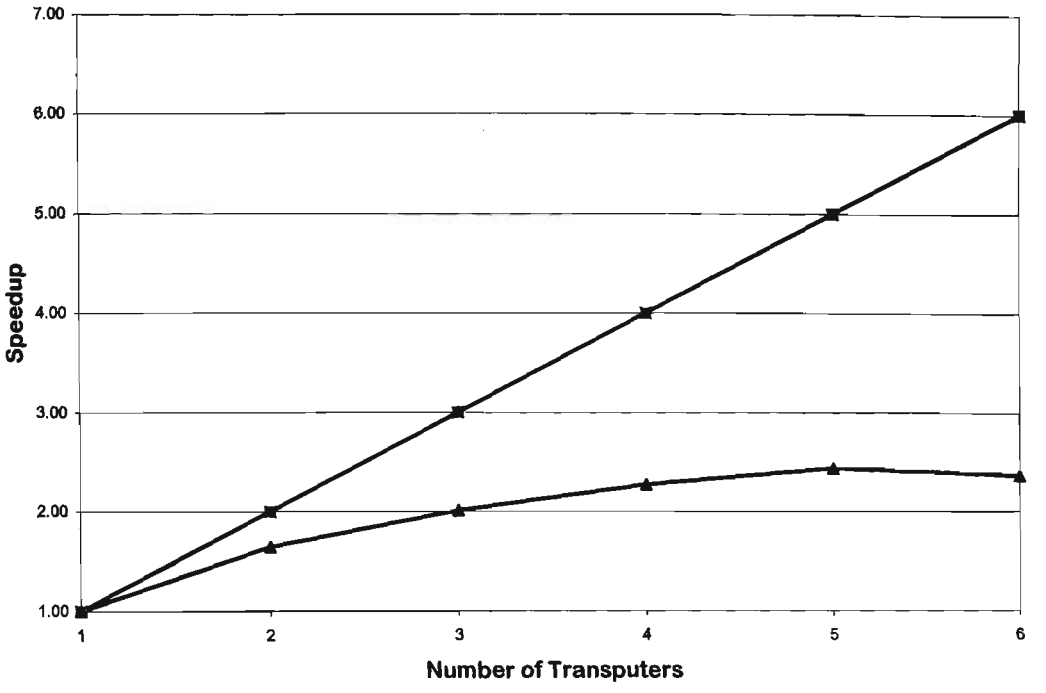


Figure 3.24 Speedup curve for Mandelbrot data from Table 3.6

As can be seen from Figure 3.24, the speedup peaks at five transputers, and actually decreases when another transputer is used. The underlying topology for this example was a linear chain [84] (see Figure 3.19). Thus a limit is reached in the number of useful processors, and once this limit is passed, communication overhead increases caused by excessive message passing from the extra processors at the end of the chain. This results in a drop in the speedup.

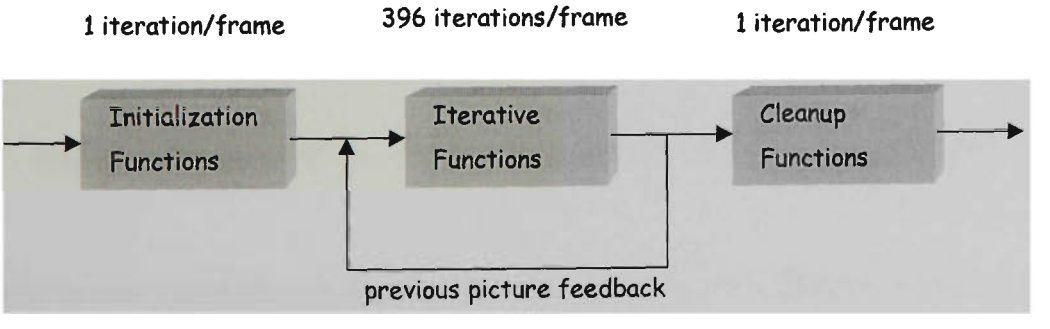


Figure 3.25 Simplified representation of H.261 encoding execution

Another example from [92] uses a processor farm paradigm to implement a parallel algorithm for the CCITT H.261 coding algorithm. A simplified representation of the encoder execution is shown in Figure 3.25, details are available in [92]. A group of functions which were executed 396 times per frame were chosen as the worker task to

be farmed out to the processor farm, and the initialization and cleanup functions were placed on the master processor.

The performance of the H.261 parallel algorithm can be seen in Figure 3.26, shown against an increasing number of processors. Again, as in the previous example (Figure 3.24), the performance increases with the number of processors to a plateau, then degrades as more processors are added. This is due to the increasing overhead associated with an increasing number of processors. The plateau in this case is reached at approximately nine processors.

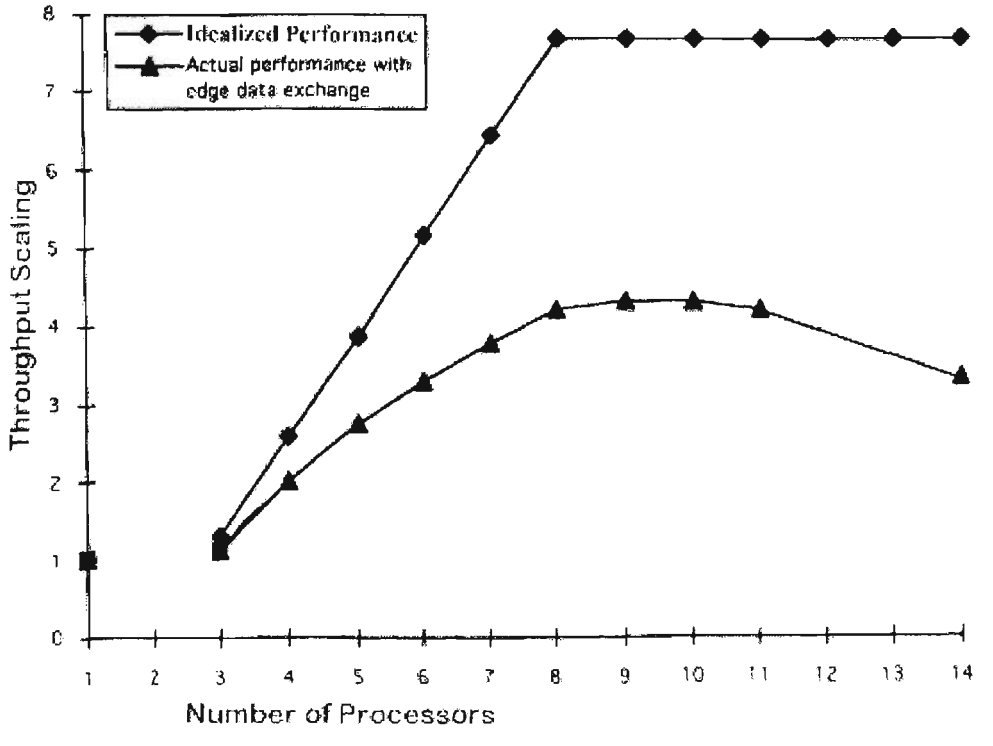


Figure 3.26 Idealized vs. actual performance for the H.261 encoder [92]

The work in this example supports the research conducted in Chapter 4 and 5.

Recent research [85] has produced performance models for the processor farm paradigm run on a number of underlying architectures. The model depends on many factors, such as underlying topology, local speed of processors, number of tasks, execution time related to receiving and forwarding messages, and task execution time. It is suggested [85] that a breadth first spanning tree topology is the most efficient

topology for the processors. However, the CPU overhead for a processor in forwarding, receiving and re-routing messages (βf), limits the performance of the model and ultimately the number of effective processors which can be utilized [85]. This is further investigated in Chapter 4 and 5. It is also suggested that for problems which fit the processor-farm paradigm, there is possibly no need to consider more elaborate parallelization techniques as this paradigm gives good performance.

3.4 Summary

The JPEG standard, which is now the international standard for still picture compression, has been discussed in detail in section 3.2. Section 3.3 discussed parallelization and parallel paradigms in detail. In Chapter 4 and Chapter 5 a parallel implementation of the JPEG is shown, and its behaviour using a number of processors is explored. As noted in 3.3.2.3 the JPEG algorithm is a perfect candidate for the processor-farm paradigm, and this is used to implement the parallel versions.

The parallel JPEG algorithm was then executed on a transputer. Details on the transputer used for testing are given in Chapter 4. The idea of a limit on the number of effective processors, which can be used in a parallelization of the JPEG standard, is also explored in the following chapters.

CHAPTER 4

PARALLEL JPEG IMPLEMENTATION

4.1 Introduction

In the previous chapters, the concepts of image compression and parallel computing were discussed, and the development of the JPEG standard for still picture image compression was outlined. With the development of the JPEG standard, much effort has been concentrated on the application of this standard. JPEG compression is often implemented in hardware where high speed compression is required, and this can be seen from the many "JPEG chips", implementing the JPEG codecs that are available, for example the L64702 from LsiLogic[1] and the ZR36060 from Zoran[2]. Lossy compression in JPEG is based on the DCT transform, which is computationally intensive and presents the biggest obstacle to fast performance. The research in this chapter develops a software based parallel JPEG algorithm and investigates the optimal paradigm, the distribution of processes for best performance and the limit on the number of useable processors.

A parallel implementation of the JPEG standard using two processors is developed and implemented in software to be run on a transputer. The performance of this algorithm is measured against the single processor version and the results presented. From the measurements of speed taken on a general n-processor system, an algorithm is designed using the processor farm paradigm discussed in Chapter 3, and an optimal configuration developed. As the number of processors available for testing is limited, results are extrapolated from those obtained on available processors.

[1] LSI Logic Corporation

[2] Zoran Corporation

The work performed in this chapter forms the basis for the simulations developed in the next, where simulated parallel runs allow the behaviour of the parallel algorithm to be investigated without being limited by the availability of physical processors. The aim of these simulated tests is to test the hypothesis from the results obtained in this chapter using actual multiple processors. From the general algorithm developed, it is shown there is a limit to the number of processors that such an algorithm can effectively use, called its *saturation point*. This concept is investigated in depth. The effects of the saturation point and its impact on the structure of the parallel algorithm are explored and the results presented.

4.2 Methodology

The methodology used to conduct this research is as follows.

Firstly, a sequential program, SV1, designed to run on a single processor was developed. This program implements the baseline sequential DCT-based algorithm, as specified in Annex F of the JPEG standard [52]. The implementation of this algorithm was tested and timed for overall performance. Performance here refers to speed of operation. The major components of this algorithm were then identified, so each could be separately timed.

The timing of the algorithm's components was performed on an image containing 4096 8×8 pixel blocks; then the timing for each component calculated on a block-by-block basis. All times used in calculations are averages computed from five timing runs of each measurement. It is expected that variations in measured times will be minor, and may only be significant when a measured component performs disk I/O, which is affected by operating system disk caching and background updates.

A parallel algorithm was then designed to run on multiple processors. Concurrency was introduced by off-loading some of SV1's components to run in parallel on

different processors. The programs were then timed and the results compared with those of the sequential algorithm SV1.

All timing data is obtained by function calls, which reference a hardware clock. All the timing function calls are also made from threads running at the same priority (low priority). As the clock has a resolution of 64μs, all timing data is presented in the form of clock ticks, where there are 15,625 clock ticks (cts) per second. Increasing all thread priorities to high would give a finer resolution of the clock (approximately 1 million cts per second), but the low priority values are sufficiently accurate for this research.

4.3 Parallel Processing Environment

The parallel processing hardware on which this research was conducted consists of a transputer system connected to an IBM PC compatible host. There was a choice of two parallel programming languages, *Occam 2* and a 3L[3] *parallel C* compiler. All programs implementing the algorithms developed in this chapter and running on the transputer system were programmed in parallel *C*. This version of parallel C followed the ANSI[4] standard for C very closely, thus its code is relatively portable.

The host to which the transputer system was connected was an IBM PC compatible with Intel 486 DX2 CPU, running at 66 MHz. This PC was equipped with an Inmos[5] IMS B008 add-in board. The B008 board has slots for up to 10 TRAMs. TRAM stands for Transputer Module and three IMS T800 TRAMs were available. The T800 transputer module incorporates a 32 bit processor, four serial communication links, 4Kbytes of RAM (expandable to 4 gigabytes) and a floating point unit (FPU) on a single chip. The FPU is a coprocessor integrated on the same chip as the 32 bit processor, operates concurrently with the processor and performs 32 and 64 bit

[3] 3L® is the registered trademark of 3L Limited, Scotland

[4] American National Standards Institute

[5] Inmos is a Trademark of the INMOS Group of companies

floating point arithmetic, to the IEEE[6] 754 standard. The configuration of TRAMs on the B008 board used is shown in Figure 4.1.

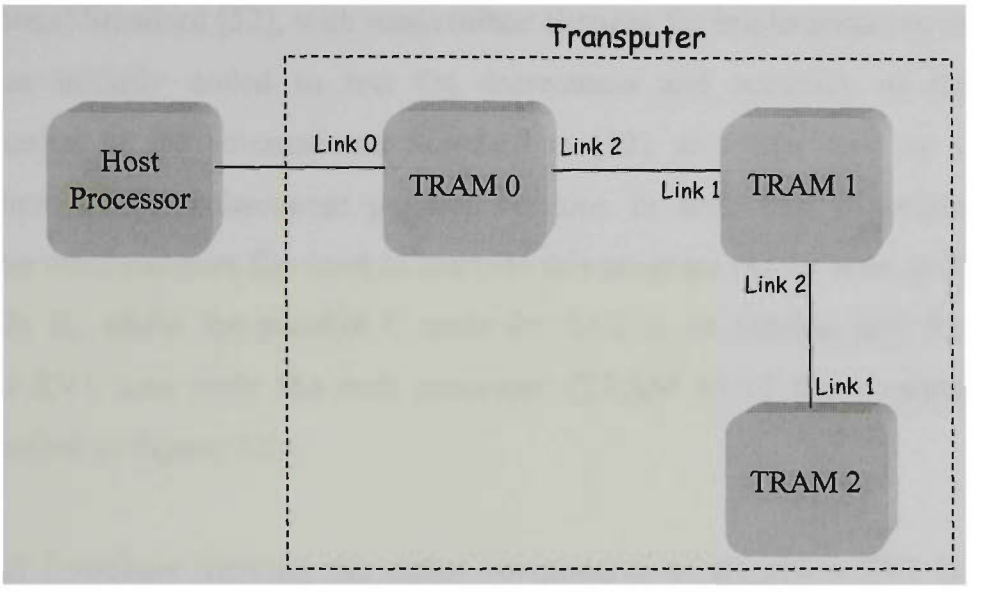


Figure 4.1 Physical transputer configuration with three processors

Although each transputer module can have up to four serial links, the transputer modules in the actual configuration were connected to just one other transputer module in serial fashion as shown in Figure 4.1. Each one of these serial links between two TRAMs consists of two uni-directional signal lines capable of operating at speeds up to 20 Mbits per second [43], [46], [45]. Logical links can be defined between two TRAMs using a configuration file, regardless of the underlying physical configuration.

TRAM 0, in the actual configuration in Figure 4.1, which is connected directly to the *host* processor, is referred to as the *root* transputer. This TRAM is regarded as special, since it runs the *afserver* software and thus all transputer I/O must be routed through this processor. This situation is described in Chapter 3 Section 3.3.1.3 and depicted in Figure 3.19.

[6] Institute of Electrical and Electronic Engineers

4.4 Sequential Algorithm

The SV1 algorithm implements the JPEG baseline DCT algorithm as specified in the International Standard [52], with some minor changes for implementation expediency. SV1 was initially coded to test the correctness and accuracy of the author's interpretation of the International Standard in [52], and then used as a basis for comparison of the subsequent parallel versions in time trial experiments. The transputer configuration file used to compile this program can be seen in Section B.2 Appendix B, while the parallel C code for SV1 is in Section B.3 Appendix B. Program SV1 uses only the root processor (TRAM 0) of the transputer system configuration in Figure 4.1.

A partial flowchart showing the major components of algorithm SV1 is shown in Figure 4.2. This compression algorithm consists of a number of basic steps. After the initialization routines and the initial building of the Huffman code tables, there are six repetitive steps, which are performed for each block in the digital image. They are:

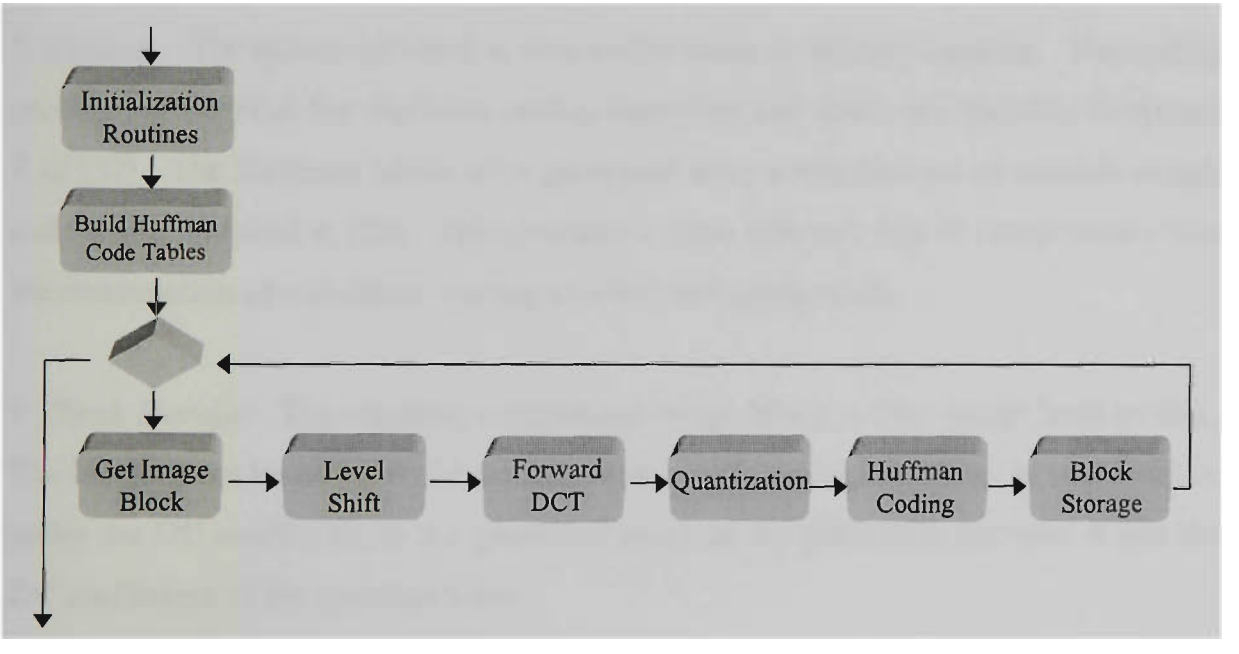


Figure 4.2 Sequential algorithm SV1 main components

1. *Get Image Block* : This component decomposes the digital image into 8×8 pixel blocks (i.e. uses $n = 8$, where n is the block size). It could be easily extended to other values of n, though the value 8 is used in the JPEG standard [52].

2. *Level Shift* : A level shift is then performed on each sample in the block by subtracting 2^{P-1} where P is the precision of the sample. In this case, 8-bit precision is used (that is, samples are in the range from 0 to 255); therefore the value 128 is subtracted to shift the samples to a signed representation with value range from -128 to 127. This effectively centres the samples around the value zero. This is specified in Annex A of [52].

3. *Forward DCT* : A Forward Discrete Cosine Transform (FDCT) algorithm is then applied to the level shifted image block. This is the fast algorithm version of the DCT proposed by Chen [14], and the signal flow diagram for this algorithm can be found in Rao [70] and Chapter 3 Section 3.2.

4. *Quantization* : The transformed block is then quantized by application of a uniform quantization table and algorithm as specified in Annex K of [52]. This is the Luminance Quantization Table K.1 used for 8-bit grey scale digital images. Here, loss of information is introduced into the compression process.

5. *Coding* : The quantized block is then coded using an entropy encoder. The coding process used applies the Huffman coding algorithm and tables are specified in Annex F of [52]. The Huffman tables were generated from a standard set of variable length code words supplied in [52]. This provides a more efficient way of compression than the construction of a Huffman coding tree for each application.

.

6. *Block Storage* : The resulting compressed image block is then stored back to disk. The blocks must be stored in the correct order, as the coding algorithm in step 5 above codes the DC coefficient in the quantized block as the difference between it and the DC coefficient of the previous block.

For testing, the algorithm was run using a number of images, which are shown in Appendix A Section A.2. Performance over all the images was similar, so the image "*George*" was used for obtaining most of the test runs' data. It is the most intricate because of the nature of the source, [Appendix A]. All images were resampled at *512* × *512* pixels so they contain exactly 4096 image blocks.

The overall timing results for this algorithm implementation are shown in Table 4.1. These times are measured in clock ticks (cts).

Table 4.1 Overall timing for algorithm SV1

Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	
1,474,320	1,474,242	1,474,307	1,474,231	1,474,314	cts
Average 1,474,283					cts

The algorithm *SV1* was then modified to obtain timing data for the six major block processing components in the iterative loop in Figure 4.2. Only the components to be parallelised, which are those inside the iterative loop, are timed. The timing data will be used to decide on the structure of the parallel algorithm and the allocation of processes. Timing data for each of these six components is presented in Table A.1 Section A.3 of Appendix A. This shows timing data for the six components over five time trials with average clock ticks over all image blocks, and per image block.

Table 4.2 Time trial averages for SV1 components

Component	Overall Average (cts)	Block Average (cts)
Get Block	41,655.4	10.1698
Level Shift	24,794.0	6.0532
FDCT	1,286,885.4	314.1810
Quantization	75,587.8	18.4541
Huffman Coding	26,170.6	6.3893
Block Storage	17,635.2	4.3055
Totals	1,472,728.4	359.5528

Table 4.2 is a synthesis of the data from Table A.1 showing the averages of clock ticks for each component, over all five time-trials. The average processing times for each of the six components per image block are also represented pictorially in Figure 4.3.

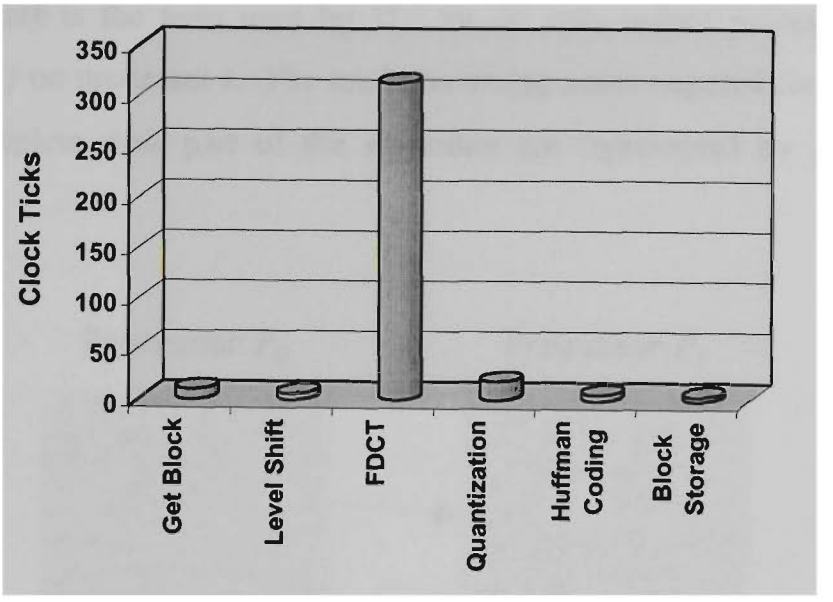


Figure 4.3 Average processing times per image block

If we take the average of the overall timing for SV1, that is 1,474,282.8 clock ticks from Table 4.1, and compare this to the sum of the averages of the six major algorithm components, 1,472,728.4 clock ticks, from Table 4.2, it can be seen that the sum of the six major components represents 99.9% (rounded to one decimal place) of the entire overall processing time for SV1. Thus the sum of the times for the six major components in Table 4.2 is a good indication for the overall timing of SV1.

4.5 Parallel Algorithm

Simple parallelism was next introduced by adding one extra processor upon which to place one or more of the algorithm's components. By observing the behaviour using one extra processor, experience can be gained with multi-processor behaviour before proceeding to the general case of n processors. In designing the parallel algorithm, PV1, the first question was what would be the optimal distribution of the SV1 components on two processors in order to minimize total processing time.

Figure 4.4 shows a representation of two communicating processors, P_0 and P_1 both involved in the processing of an algorithm, with multiple tasks running on each

processor. A *task* is the term used by 3L[7] for an independent process. Task $t_i(j)$ represents task j on processor i. The total processing times required for processor P_0 and P_1 to complete their part of the algorithm are represented by ΔP_0 and ΔP_1 respectively.

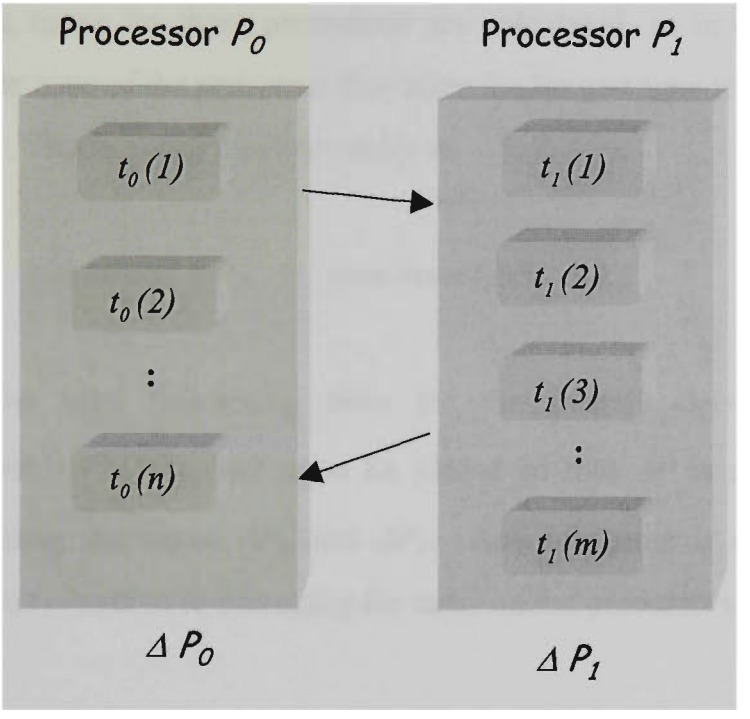


Figure 4.4 Total processing times per processor

The times ΔP_0 and ΔP_1 are equal to the sums of the total processing time of the tasks on each of the processors P_0 and P_1 respectively. That is

$$\Delta P_0 = \sum_{i=1}^{n} \delta t_0(i) \quad and \quad \Delta P_1 = \sum_{j=1}^{m} \delta t_1(j) \ , \tag{4.1}$$

where the quantity $\delta t_i(j)$ in (4.1) is the total processing time of task $t_i(j)$. Similarly, if $t_i(j)$ itself contains a number of components, then $\delta t_i(j)$ is the sum of the processing times of those components.

When two independent processors, each containing a number of tasks are both working on parts of the same problem and transmitting their results, as in Figure 4.4,

[7] 3L Ltd. Edinburgh software house.

sequencing and placements of tasks is crucial. The times ΔP_0 and ΔP_1 in (4.1), include any idle time when a task (or sequence of tasks) is idle as one processor waits for communication from the other. Any such idle time will affect the overall processing time of the algorithm. The term ΔP is used to represent the total processing time of an algorithm, such as that in Figure 4.4 using processors P_0 and P_1. If the processing times for these processors are calculated, as in (4.1), then ΔP is constrained by the time of the processor that takes the longest time to complete its part of the algorithm. This is stated more formally as

$$minimum(\Delta P) \geq maximum(\Delta P_0, \Delta P_1) \tag{4.2}$$

To minimize the total processing time for the overall algorithm, the tasks $t_0(1) \cdots t_0(n)$ and $t_1(1) \cdots t_1(m)$ must be placed so that ΔP is minimized. This involves minimizing the times ΔP_0 and ΔP_1. Any idle time is eliminated where possible, which may involve re-arranging the tasks on the processors.

From Figure 4.3, it can be seen that the component which takes most processing time per block in SV1 is the FDCT (Forward DCT) component. This component takes approximately 87.38% of each block's total processing time, and consequently that of the entire processing time per image. The entire processing time can be no faster than the processing time of the FDCT. The FDCT task itself could be split and placed on different processors. However, because of the cohesion of the elements in this task, any such implementation becomes less portable. Thus, this research considers the FDCT as a single task.

If we are looking for an optimal configuration of the image processing tasks from SV1 on two processors, then the FDCT processing component must be placed in a task on a processor by itself. That is, if the FDCT task is placed on processor P_1 and all other components are placed in a task on processor P_0, then using Table 4.2 it can be seen that

$$\Delta P_0 \approx 185{,}841 \text{ clock ticks}$$

and $$\Delta P_1 \approx 1{,}286{,}885 \text{ clock ticks.}$$

Thus, moving any other of the image processing tasks of SV1 to processor P_1 would only increase ΔP_1.

Table 4.3 Processing task symbols

Component	Symbol
Get Block	T_{gblock}
Level Shift	T_{lshift}
FDCT	T_{dct}
Quantization	T_{quant}
Coding	T_{code}
Block Storage	T_{store}

To help state this more formally, the six major components from algorithm SV1 identified in Figure 4.2 are represented with the symbols shown in Table 4.3. Then from Table 4.2 it can be seen that

$$\delta T_{dct} > \delta T_{gblock} + \delta T_{lshift} + \delta T_{quant} + \delta T_{code} + \delta T_{store} \tag{4.3}$$

where $\delta T_{component}$ represents the time it takes for component $T_{component}$ to process per image block. These times are shown in Table 4.2. Therefore an optimal placement of components on processors P_0 and P_1, which minimizes ΔP in (4.2), is

$$T_{dct} \rightarrow P_1 \quad \text{and} \quad T_{gblock}, T_{lshift}, T_{quant}, T_{code}, T_{store} \rightarrow P_0 \tag{4.4}$$

where the symbol "$\rightarrow$", is used to mean, "is allocated to". Thus,

$$\Delta P_0 \approx B(\delta T_{gblock} + \delta T_{lshift} + \delta T_{quant} + \delta T_{code} + \delta T_{store}) = 185{,}841 \text{ cts} \tag{4.5}$$
$$\Delta P_1 \approx B(\delta T_{dct}) = 1{,}286{,}885 \text{ cts}$$

where B is the number of image blocks, and this was 4096. The above calculations ignore for the moment the communication times between processors P_0 and P_1.

If the components from SV1 are placed as in (4.4), then apart from the first block, while each subsequent block is being processed by T_{dct} on P_1, then the previous block can have the T_{quant}, T_{code}, T_{store} performed in parallel, followed by the T_{gblock} and T_{lshift} of the next block on P_0. When processor P_1 passes the transformed block back to P_0, P_0 can then pass the next block to be transformed to P_1, and then continue processing the block just received. This would give the maximum degree of parallelism with these tasks on a two processor system.

Parallel algorithm PV1 was developed from SV1 and was designed to the configuration just described. It runs with the T_{dct} block processing component on one processor P_1 and all other components on the processor P_0. The structure of algorithm PV1 is shown Figure 4.5 . The implementation of PV1 uses TRAM 0 and TRAM 1 of the transputer in Figure 4.1. Processor P_0 is the representation of TRAM 0 (root transputer), and processor P_1 is the representation of TRAM 1. The configuration file for the program is in section B.4, while program code can be seen in section B.5.

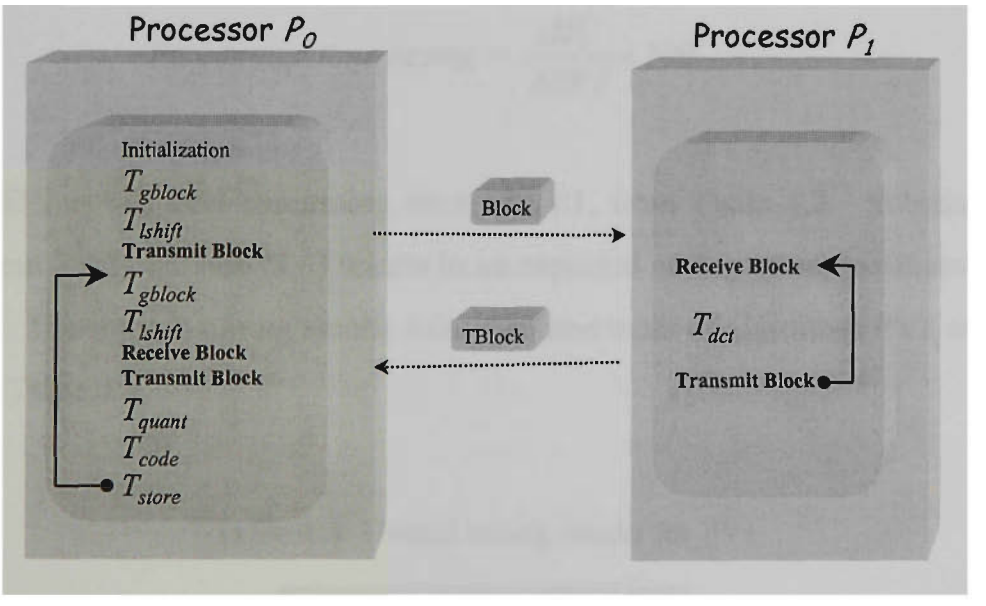


Figure 4.5 Structure of parallel algorithm PV1

The structure of the algorithm PV1 in Figure 4.5 is such that the relationship between the processes on P_0 and P_1 is essentially a master /slave configuration, with only one slave processor. The only process placed on P_1 was the T_{dct}. All other components, T_{gblock}, T_{lshift}, T_{quant}, T_{code} and T_{store} are placed on the processor P_0. As T_{dct} is the most

computationally intensive task (4.3), this algorithm PV1 has introduced only a limited degree of parallelism.

In Figure 4.5, the image blocks (denoted by *Block*) are passed to P_1 and the transformed blocks (denoted by *TBlock*) are passed back to P_0 as fixed length messages. There is no common memory or sharing of common address space on a transputer. The purpose of this algorithm PV1, was to see if the expected gain from simple parallelism could be achieved and hence calculated with some degree of accuracy. From the previous discussion and (4.2), we would expect the overall run time of algorithm PV1 (ΔP) to be approximately that of ΔP_1, that is, $\Delta P \approx \Delta P_1$.

Due to the essentially sequential nature of this algorithm, the only time savings can be that of the processing time for the tasks T_{gblock}, T_{lshift}, T_{quant}, T_{code}, and T_{store}. These tasks are running on processor P_0, in parallel with task T_{dct} which is running on P_1. The expected time saving of PV1 over SV1 is approximately ΔP_0, that is 185,841 clock ticks or, expressed as a percentage,

$$Percentage\ time\ saving = \frac{\Delta P_0}{\Delta SV1} \times 100 \tag{4.6}$$

where $\Delta SV1$ is the total processing time of SV1, from Table 4.2. Substituting the values from Table 4.2 into (4.6) results in an expected saving of approximately 12.6% of $\Delta SV1$. The overall timing results from the time trials of algorithm PV1 are shown below in Table 4.4.

Table 4.4 Overall timing results for PV1

Trial	Times (cts)
1	1,132,467
2	1,125,338
3	1,125,395
4	1,125,486
5	1,125,336
Average	1,126,804

Using the actual figures obtained from Table 4.2 and Table 4.4 a time saving of 345,924 clock ticks or 23.5% over algorithm SV1 was recorded. The results were better than expected. However, the discrepancy is sufficient to warrant further investigation into its cause.

4.6 Component Timing Issues and Communication

Because of the unexpected timing results of the previous section, the behaviour of all the processors in the transputer system of Figure 4.1 is now investigated. This is crucial to the accuracy of any calculations involving the times of the JPEG processes within a parallel algorithm running on the transputer system of Figure 4.1, and any hypothesis made from those results. The timing of message transmission between the processors must also be investigated since any parallel system, which relies on message passing for processor communication, suffers delays while data transmission is in progress. These delays in processor communication must be measured in order to evaluate any impact on the design of a general parallel JPEG algorithm.

4.6.1 Component Timing

It must be realized that an unstated assumption was underlying the calculations in Section 4.5. That was, if task t on processor P_i took x clock ticks to process, then task t on processor P_j would take the same time. That is, the execution time of all components was constant over all processors in the system. The running times of components placed on these processors depend on both the hardware and the software environment.

4.6.1.1 Hardware Issues

The platform described in section 4.3 that was used in this research possessed three processors. The four non I/O components from algorithm SV1, T_{lshift}, T_{dct}, T_{quant}, T_{code},

were individually timed on each of the processors, TRAM 0, TRAM 1, and TRAM 2. These processors will be referred to henceforth as processors P_0, P_1 and P_2 respectively. All three of these processors were eventually used. The timing results are shown in Appendix A Table A.2, which is summarized below in Table 4.5 on a per block basis. The timing results of these components on P_0, have already been shown in Table 4.2, but are now compared to those of the other processors.

Table 4.5 Comparison timing data per block of non I/O processes on all processors

Processor	FDCT (cts)	Level Shift (cts)	Quantization (cts)	Coding (cts)
P_0	314.181	6.053	18.454	6.389
P_1	267.176	5.259	16.031	5.136
P_2	213.327	4.206	12.805	4.074

The values in Table 4.5 are rounded to three decimal places, and represent the average processing times per image block. The corresponding values for P_0 are then taken from the second column of Table 4.2. The results of Table 4.5 are also shown graphically in Figure 4.6. As can be seen, all processing times of these components are different on all three processors.

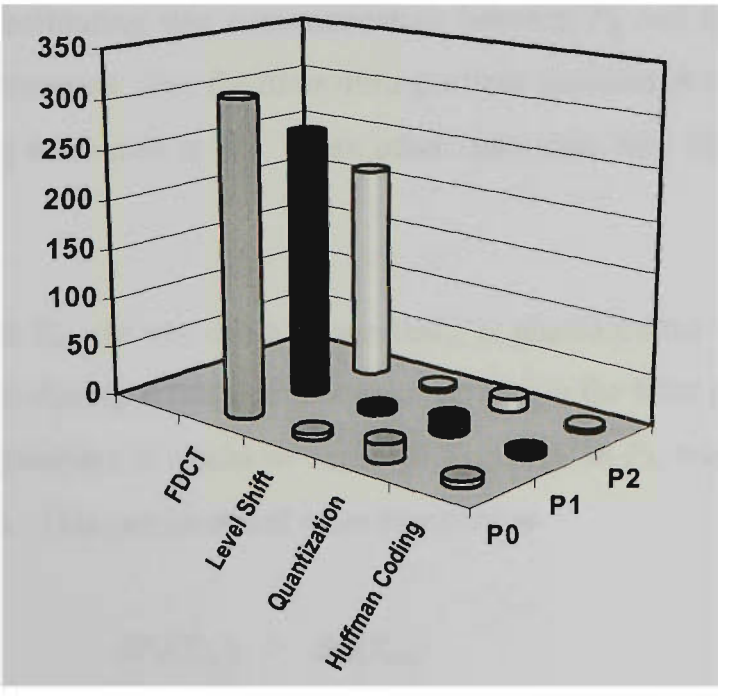


Figure 4.6 Comparative chart representation of Table 4.5

Further investigation into the underlying hardware revealed that processor P_2 (TRAM 2), was purchased some years later. This processor runs at a higher clock speed than P_0 or P_1, therefore a lower value for δT_{dct}, or any of the other components, would be expected on P_2. However, a lower value for all components was also recorded on P_1, which runs at the same clock speed as P_0. This difference in component processing times on processors P_0, P_1, P_2 explains the better than expected performance of algorithm PV1.

4.6.1.2 Software Issues

An explanation of the discrepancy between P_0 and P_1 lies in the architecture of the underlying hardware and the software configuration used to support this multi-processor activity. As can be seen in Section 3.3.1.3 Figure 3.19, the root processor (P_0) on the transputer contains a *filter* task which facilitates communication between tasks on processor P_0 and the *afserver* task on the transputer host processor. Being a separate executable task placed on processor P_0, *filter* is scheduled for computer time along with all other tasks placed on P_0. Processes T_{gblock} and T_{store} that perform all I/O processing are also resident on P_0 and must, by necessity, make use of the filter task indirectly. Thus throughout the lifetime of the algorithm, *filter* uses scheduled processor time facilitating this communication between P_0 and the transputer host. Any scheduled processor time for *filter* during a time quantum δt means less time for other tasks being measured in δt. These other tasks then take longer to run in real time.

When component T_{dct} (or any other component), is placed on the non-root processor P_1, it is no longer sharing scheduled processor time with the filter process. Therefore in a given time quantum, it would be expected that δT_{dct} on P_0, would be greater than δT_{dct} on P_i ($i \neq 0$). This can be stated more formally as

$$\delta P_0(T_{dct}) > \delta P_i(T_{dct}) \qquad (4.7)$$

where $\quad clock\ speed\ P_0 \leq clock\ speed\ P_1.$

This is borne out in the results presented in Table 4.5 above. If the clock speed of P_0 is greater than that of P_1, the inequality in (4.7) may still hold true, but that is not the situation being described here. By examining the ratios of the times for each process in Table 4.5 on processors P_0 and P_1, it can be seen from Table 4.6, that the ratio average of 0.8478 is similar to the ratios of the individual components. This means that the impact of the filter process is consistent across all processes on P_0.

Table 4.6 Ratios of component processing times of P_1 / P_0

Component	Ratio P_1 / P_0
T_{dct}	0.8503
T_{Ishift}	0.8687
T_{quant}	0.8687
T_{code}	0.8038
average	0.8478

4.6.1.3 Expected Time Savings of PV1

Given this, the expected time savings of PV1 in Section 4.5 can now be recalculated to allow for the speed difference of processor P_1. The percentage time saving of PV1 in (4.6) is obtained directly by calculating ΔP_0 as a percentage of the benchmark time $\Delta SV1$. This calculation is correct, from (4.2)-(4.5), if both processors run under identical conditions. Since they effectively do not, because of the filter task, then the percentage can be calculated from the expected running time of the algorithm, ΔP ($\approx \Delta P_1$), as a percentage of $\Delta SV1$. This allows us to use the known processing times on P_1 from Table 4.5. So the measured time saving in (4.6) is more accurately calculated by (4.8), which allows for the time difference of the T_{dct} component on P_1.

$$Percentage\ time\ saving = \left(1 - \frac{\Delta P}{\Delta SV1}\right) \times 100 \tag{4.8}$$

From Figure 4.6 and (4.4), it can be seen that the value of ΔP_1 is still much greater than ΔP_0. By using the more accurate values from Table 4.5, then from (4.1) and

(4.2), we can derive a value for ΔP of 1,094,352.9 clock ticks. It must be remembered that the values in Table 4.5 are per image block, and must be multiplied by 4096 to get total component processing times. Using this figure in (4.8) yields a new expected time saving for PV1 of 25.7%. This is close to the observed time savings of 23.5% calculated in section 4.5 and much better than the 12.6% originally calculated. These calculations do not yet take into account the timing of the message passing between processors, which when accounted for, should bring the expected time savings even closer to that observed.

4.6.2 Communication Timing

While the inter-processor communication times may be small compared to the processing times of the tasks, they contribute to the overall times for the algorithm. The processor communication times are influenced by a number of factors including the length of the message passed, and the topology of the network of processors. The topology of the transputer system used is shown in Figure 4.1. Communication times were found between processors P_0 - P_1, and P_0 - P_2, for varying message lengths.

The communication timing was done by timing the transmission and receipt of 4096 8×8 blocks (or arrays) of elements between processors P_0 - P_1, P_0 - P_2. Four runs were performed with different size block elements. Block communication was done with elements of one byte, four bytes, eight bytes, and finally by transmitting a block of four byte elements but receiving a block of eight byte elements. The final test was performed, as this type of communication is essentially what happens in algorithm PV1. An 8×8 block of 32 bit integers is transmitted to processor P_1 which performs the FDCT, and then transmits back a transformed 8×8 block of 64 bit double precision floating point numbers. The results of the communication timing can be seen in Tables A.3 – A.6 in Appendix A, and are summarized in Table 4.7 below. Note that all times are again in clock ticks and represent the total time taken to transmit and receive all 4096 blocks.

Table 4.7 Communication timing summary between P_0 and P_1 , P_2

Bytes per element	Processor P_1 (cts)	Processor P_2 (cts)
1	5,192.2	10,373.8
4	19,020.2	38,494.4
8	37,441.2	75,961.4
4 / 8	28,236.0	56,922.8

The data in Table 4.7 can also be represented as a comparison bar chart in Figure 4.7 where the communication timing between processors P_0 - P_1, P_0 - P_2 can clearly be seen. From Table 4.7 we can see that the communication times are doubled when communicating between P_2 and P_0. This is due to the configuration of the underlying hardware shown in Figure 4.1. In this configuration, P_0 can only communicate with P_2 via message routing through P_1.

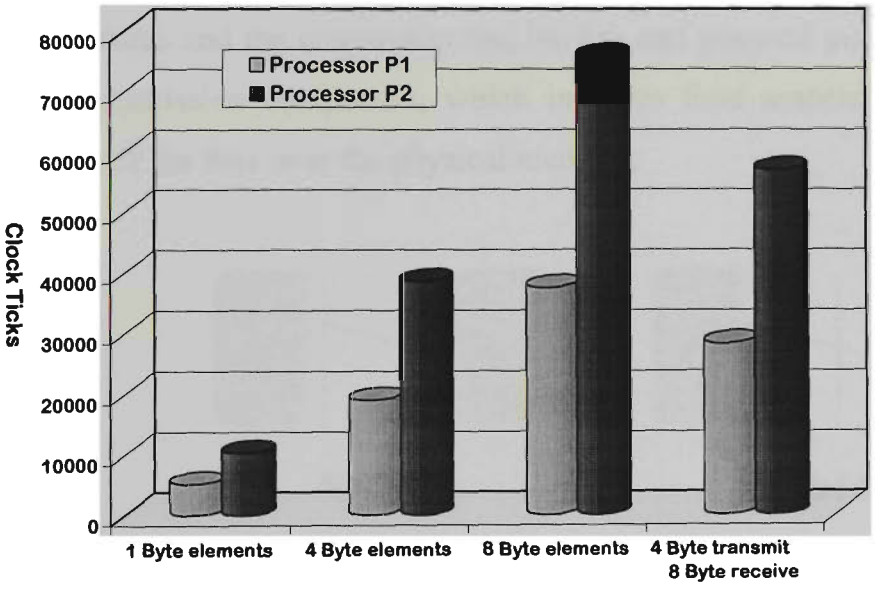


Figure 4.7 Comparison of communication timing between different processors

The net effect is that the communication times for messages between P_0 and P_2 is the sum of the communication times between the two sets of adjacent processors. This implies that communication times are a function of distance between processors. This also suggests that distance be measured not by the actual distance apart, but by the number of receiving processors involved in the message routing. Therefore, in Figure 4.1, the distance between P_0 and P_1 is one, while the distance between P_0 and P_2 is two.

Before discussion of the impact of these communication times, we also need to consider the model by which we are viewing the communication process. Shown in Figure 4.8 are three models for viewing communication between processors. Model A suggests that the communication process is an event whose time duration is shared by each of the communicating processors and ignores the physical medium by which actual communication takes place. Model B is an opposing view, and places the communication event solely in the domain of the physical medium. This assumes the communicating processors take no part in the communication process, and hence are not directly affected by the communication times. In this model each processor can view the communication delay as something apart from itself. Model C combines both A and B. In this model, the communication process has three components. There is a component at each processor end, which involves calling the communication functions, which prepare the data, and copy it to and from supplied program data structures and the communication buffers and physical ports. There is also the actual transmission component, which involves time associated with the actual transmission of the data over the physical medium.

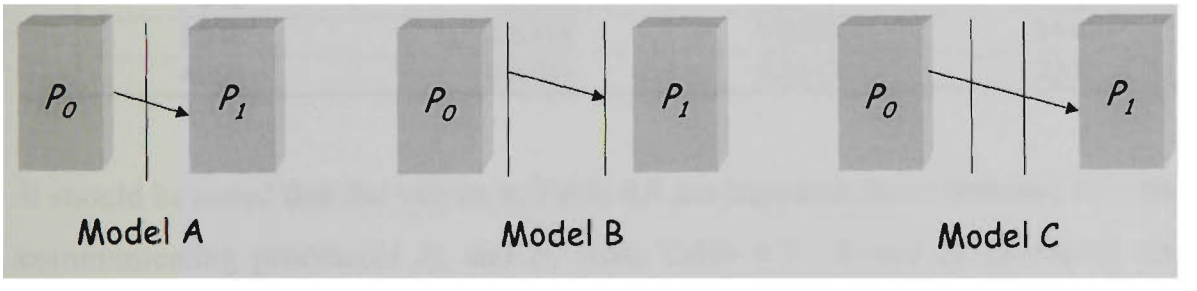


Figure 4.8 Inter-processor communication models

Evidence from Table 4.7 suggests that Model C is the one we should use when considering the effects of communication times. From Table 4.7, 4096 blocks of 64 single byte elements are passed to and from processors P_0 and P_1 in 5,192.2 clock ticks, this equals 0.3323 seconds. Thus it takes approximately 4.0564×10^{-5} seconds to transmit one such block (512 bits), one way. From section 4.3, the speed of the communication link between processors in the transputer system is approximately 20 Mbits/sec, thus requiring 2.4414×10^{-5} seconds to transmit 512 bits. This difference from the measured figure above of 4.0564×10^{-5} seconds must be the time taken by the

processors at each end to initiate and finalize the communication. Thus, approximately 40% of the measured communication time between processors involve the sending and receiving processors P_0 and P_1.

When similar calculations are performed for the larger block communications in Table 4.7, the difference between the measured communication time and the physical time required to transmit via the communication line becomes slightly less. This can be observed in Table 4.8 as the percentage of processor involvement measured against the required time, assuming 20 Mbits/second communication speed. This drop-off in processor involvement is due to the transmission line speed remaining constant, and the efficiency of processors in handling higher volumes of data compared with smaller ones. It is reasonable to assume that the percentage of processor involvement time can be shared equally between the sending and receiving processors.

Table 4.8 Processor involvement in communication

Size of Transmission (Bits)	Measured Time (cts)	Required Time (cts)	Processor Involvement %
512	0.6338	0.3815	39.8
2048	2.3218	1.5259	34.3
4096	4.5705	3.0517	33.2

It should be noted that the values in Table 4.8 are based on those obtained by adjacent communicating processors P_0 and P_1 from Table 4.7. Based on measured results, times for communication between non-adjacent processors can be obtained by using a linear distance function, as discussed previously in this section.

The effects of the communication times can now be included into some of the previously constructed equations based on the results measured in the previous tables. Let δC represent the total communication time involved in the processing of one image block between adjacent processors. Then from Table 4.8 it can be separated into two components

$$\delta C = \delta C_{proc} + \delta C_{tmit}. \tag{4.9}$$

The quantity δC_{proc} represents the communication processing time required by both sending and receiving processors for the processing of one image block. The quantity δC_{tmit} represents the time that data is physically in transit between sending and receiving processors during the processing of one image block. Figure 4.9 is an update of Figure 4.4, with allowance made for communication times.

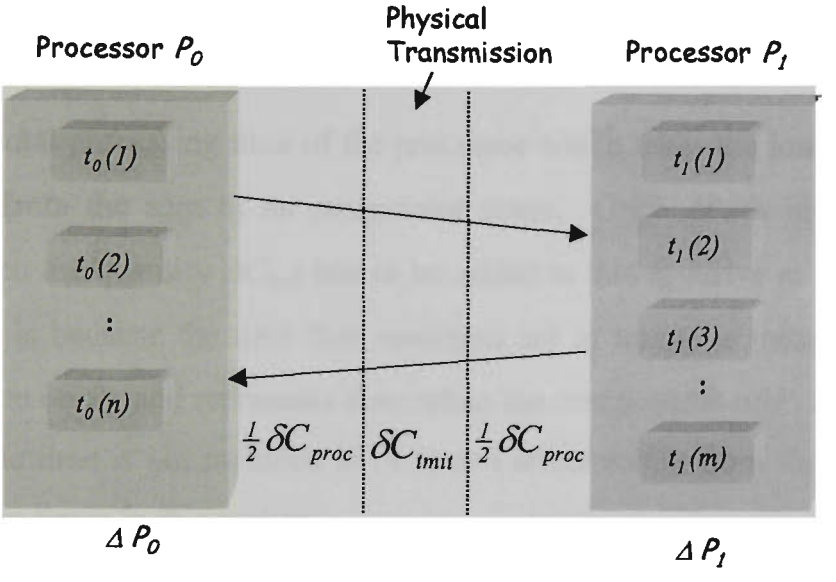


Figure 4.9 Sum of processing times with communication allowance

The equations in (4.1) can be updated to allow for these communication times, giving

$$\Delta P_0 = \sum_{i=1}^{n} \delta t_0(i) + \frac{1}{2} \sum_{blocks} \delta C_{proc} \qquad (4.10)$$

$$\Delta P_1 = \sum_{j=1}^{m} \delta t_1(j) + \frac{1}{2} \sum_{blocks} \delta C_{proc}$$

where ΔP_0, ΔP_1 and $\delta t_i(j)$ are as before. The difference in (4.10) is the addition to the quantity ΔP_i of its share of the processor involvement component of the communication times, measured over all blocks processed. Equation (4.2) can now be rewritten as

$$minimum(\Delta P) \geq maximun(\Delta P_0, \Delta P_1) + \Delta C_{tmit} \qquad (4.11)$$

where $\Delta C_{tmit} = \sum_{i=1}^{n} d_i \delta C_{tmit}$,

d_i is the distance involved in the communication of each block, and n is the number of image blocks. In the simple case of PV1 from section 4.5, the two communicating processors P_0 and P_1 are adjacent thus the quantity d is one. To minimize the total processing time ΔP, we need an optimal placement of processor tasks whose combination also minimizes ΔC_{tmit}. Since there is a special relationship between the two components δC_{proc} and δC_{tmit} shown in (4.9) and Table 4.8, minimizing the δC_{tmit} component of (4.9) will also minimize the δC_{proc} component of (4.10).

In (4.11) the total processing time of the processor which takes the longest time, ΔP_i, is calculated from the sum of its component times. Once ΔP_i is identified as the maximum, then the quantity ΔC_{tmit} has to be added to this to arrive at a lower bound for ΔP. This is because the time that messages are in transit is independent of the processing time on P_i, and represents time when the components of P_i are waiting for data. This idle time is not included in (4.1) and arises solely from the discussion of communication times in this section.

4.6.2.1 Expected Time Savings of PV1 Recalculated

Given the discussion in the previous section, the calculation for the expected time savings of algorithm PV1 from section 4.6.1 can now be recalculated, taking into account the measured communication times. By doing this it is expected that the value calculated should be closer to the observed figure of 23.5%, in section 4.5, than the value of 25.7% calculated from taking into account variations in processor speed in section 4.6.1.

In PV1 from Figure 4.5, the task on processor P_0 transmits to P_1, 8×8 blocks (64 elements) of level shifted integers. Each integer is four bytes in length. The task on P_1 then transmits back 64 element blocks of double precision floating point FDCT transformed numbers. Each double precision floating point number is eight bytes in length. Using the appropriate figures from Table 4.7 and Table 4.8, the following values are calculated

$$\Delta C_{tmit} = 18{,}753.5 \text{ clock ticks}$$

$$\Delta C_{proc} = 9{,}477.2 \text{ clock ticks.}$$

Substituting these values of ΔC_{tmit} and ΔC_{proc} into (4.10) and (4.11) a value of 1,117,845 clock ticks for ΔP can be calculated. Using this in (4.8) the new, expected time savings of algorithm PV1 compared to SV1, taking into account communication times and processor speed is 24.17%. This has taken our expected result in time savings much closer to the measured savings of 23.5%. This is close enough to be confident that hypothesis proposed later can be made with an acceptable error margin.

4.6.2.2 Impact of Communication Times on Optimal Placement

We now need to ascertain whether taking into account the communication timing between processors P_0 and P_1 will produce a different optimal configuration of components allocated to these processors in algorithm PV1. When seeking to minimize the communication time and hence ΔP, the order of the overall tasks to be performed, as shown in Figure 4.2, must be considered. Although multiple processors are being used and a parallel algorithm is being developed, the order of the tasks to be performed for each image block is essentially sequential. Therefore when selecting tasks to move to the second processor, the tasks should be a block of one or more adjacent tasks from Figure 4.2. If two non-adjacent tasks are selected from Figure 4.2 for processor P_1, then the amount of communication would double as processors P_0 and P_1 would then require four bursts of data communication in order to process one block.

If the initial optimal placement of processor tasks shown in (4.4) is considered, then task T_{dct} will be placed by itself on processor P_1. This requires that the output of task T_{lshift} (64-element block, 32 bits/element) on P_0 be transmitted to T_{dct} on P_1, which in turn transmits its output of a transformed image block (64-element block, 64 bits/element) back to task T_{quant} on P_0. We can now update (4.5) using (4.10) and data from Table 4.7 and Table 4.8,

$$\Delta P_0 \approx \sum_{blocks}\left(\delta T_{gblock} + \delta T_{lshift} + \delta T_{quant} + \delta T_{code} + \delta T_{store} + \tfrac{1}{2}\delta C_{proc}\right) = 190{,}682.6 \text{ cts} \quad (4.12)$$

$$\Delta P_1 \approx \sum_{blocks}\left(\delta T_{dct} + \tfrac{1}{2}\delta C_{proc}\right) = 1{,}291{,}726.6 \text{ cts}$$

$$\delta C_{tmit} \approx 18{,}547.6 \text{ cts}$$

$$\Delta C \approx 28{,}230.7 \text{ cts.}$$

From (4.11) $minimum(\Delta P) \approx 1{,}310{,}274.2$.

In order to try to minimize ΔC in (4.12), a different placement of tasks is considered

$$T_{lshift}, T_{dct} \rightarrow P_1 \quad \text{and} \quad T_{gblock}, T_{quant}, T_{code}, T_{store} \rightarrow P_0. \quad (4.13)$$

Because task T_{lshift} is moved to P_1, P_0 now only needs transmit a 64-element block with 8 bits/element to P_1. The transmission from P_1 to P_0 remains the same. Using the same procedure as in (4.12) gives:

$$\Delta P_0 \approx \sum_{blocks}\left(\delta T_{gblock} + \delta T_{quant} + \delta T_{code} + \delta T_{store} + \tfrac{1}{2}\delta C_{proc}\right) = 164{,}774.2 \text{ cts} \quad (4.14)$$

$$\Delta P_1 \approx \sum_{blocks}\left(\delta T_{lshift} + \delta T_{dct} + \tfrac{1}{2}\delta C_{proc}\right) \approx 1{,}315{,}406.2 \text{ cts}$$

$$\delta C_{tmit} \approx 13{,}862.3 \text{ cts}$$

$$\Delta C \approx 21{,}316.7 \text{ cts}$$

Again from (4.11) $minimum(\Delta P) \approx 1{,}329{,}268.5$.

While in (4.14) the quantity ΔC representing the total communication is less than in (4.12), the difference is more than offset by the increase in ΔP_1 due to placing T_{lshift} on P_1. This results in an increase to the overall time ΔP. Task T_{lshift} is the smallest task, therefore moving any other task to processor P_1 would result in a larger increase to ΔP. This would be more than enough to offset any reduction in communication times.

While the communications times do affect the timing of the overall algorithm, when only the two processors P_0 and P_1 are involved the amounts are not enough to change

the optimal placement of tasks discussed in section 4.5. The issue of communications will again be discussed in the next section when the optimal distribution of tasks on more than two processors is considered.

4.7 Development of a General Parallel Algorithm

The algorithm developed in section 4.5 now needs to be extended to allow for processing on a general *N-processor* system. Ideally, we need to construct a parallel algorithm consisting of a number of tasks, and distribute these tasks in such a manner as to make efficient use of the processors, while minimizing communication times. As discussed in section 4.6.2.2, reducing communication times will not necessarily reduce the execution time of the overall algorithm. The problem of distributing tasks in an optimal manner for a general homogeneous N-processor system is NP-hard even for small values such as $N = 3$ [57].

From section 3.3.2, there are two general approaches to be taken when designing a parallel implementation of the JPEG algorithm. *Homogeneous* parallelization where repetitive work is distributed over multiple processors, or *heterogeneous* parallelization where independent algorithm components are distributed over multiple processors. Each of these two categories are fairly broad, and there are many ways to structure an algorithm within each. The structure is dictated to some extent by the problem at hand. The sequential nature of the JPEG algorithm is restrictive to some extent here.

4.7.1 Pipeline Implementation

If we consider heterogeneous parallelization, we can try to distribute the independent components of algorithm SV1 over multiple processors. We have already seen from section 4.5 that if there are two processors, P_0 and P_1 then the optimal distribution of tasks is given in (4.4). The question then arises of how to distribute the tasks under heterogeneous parallelization if there are, for instance, three processors. A common

implementation of heterogeneous parallelization is that of a *pipeline*, section 3.3.2.2. We investigate the use of a pipeline to implement SV1.

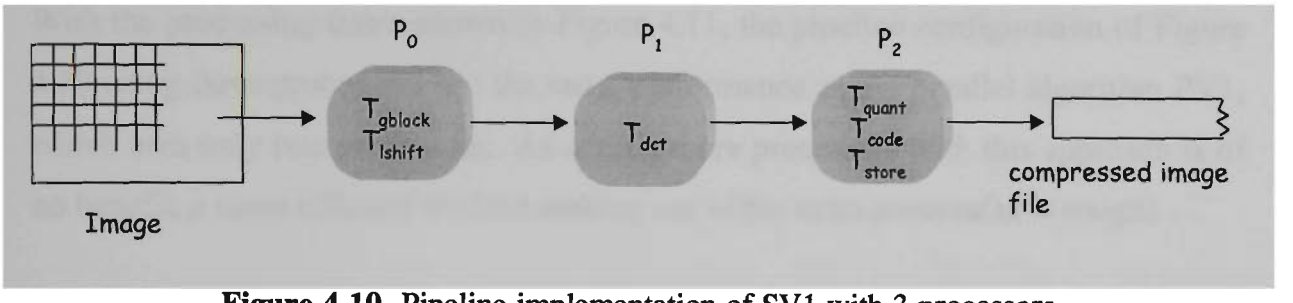


Figure 4.10 Pipeline implementation of SV1 with 3 processors

Figure 4.10 shows a pipeline implementation of SV1 using three processors. This is one possible implementation of a pipeline. With more processors, some of the components sharing a processor could be moved onto processors by themselves. However, the configuration in Figure 4.10 will suffice for illustration purposes. The JPEG algorithm is sequential in nature as all components must be applied in sequence. The measurements in Table 4.2, illustrate an inherent weakness in the pipeline approach to the JPEG algorithm.

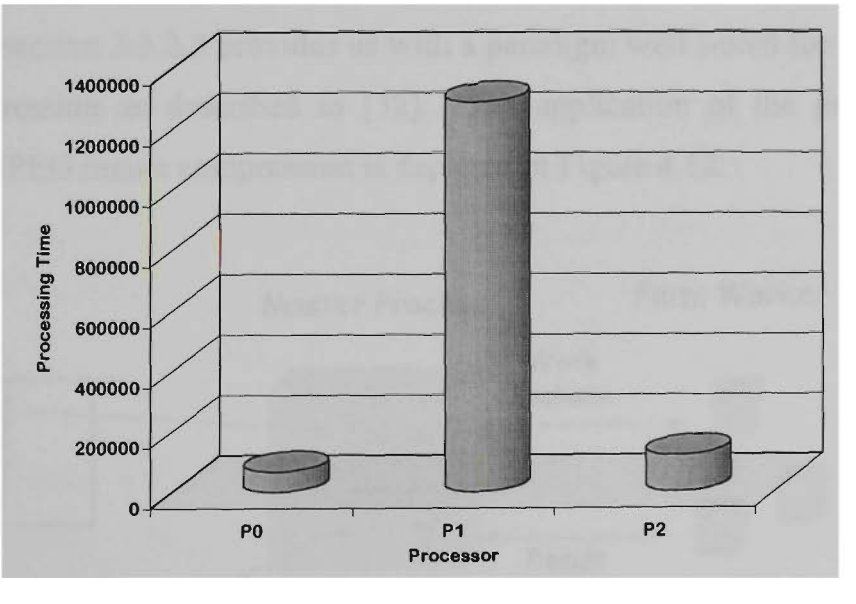


Figure 4.11 Calculated processing times of processors in pipeline

A large process can easily cause a bottleneck slowing the operation of the entire pipeline. Using values in Table 4.2, a bar chart representation of the expected processing times on the three processors in Figure 4.10 is displayed in Figure 4.11. As can be seen from this, processor P_1 containing the T_{dct} component becomes the

bottleneck in the pipeline. Adding more processors to this pipeline to separate the other components would not help; the existing bottleneck still remains.

With the processing times shown in Figure 4.11, the pipeline configuration of Figure 4.10 using three processors has the same performance as the parallel algorithm PV1, which uses only two processors. As adding more processors with this approach is of no benefit, a more efficient method making use of the extra processors is sought.

4.7.2 Processor Farm Implementation

We can construct a parallel algorithm that takes advantage of the parallelism inherent in the geometry of the image. In the lossy compression algorithms of the JPEG standard [52], each block of the source image is compressed independently of other blocks. Thus parallelism can be introduced by distributing the blocks among the processors for either partial or full processing. In fact, if we perform the same processing, on different blocks on different processors simultaneously, then we are in effect performing homogeneous parallelization. The processor farm method described in section 3.3.2.3 provides us with a paradigm well suited for application to image compression as described in [52]. The application of the processor farm approach to JPEG image compression is depicted in Figure 4.12.

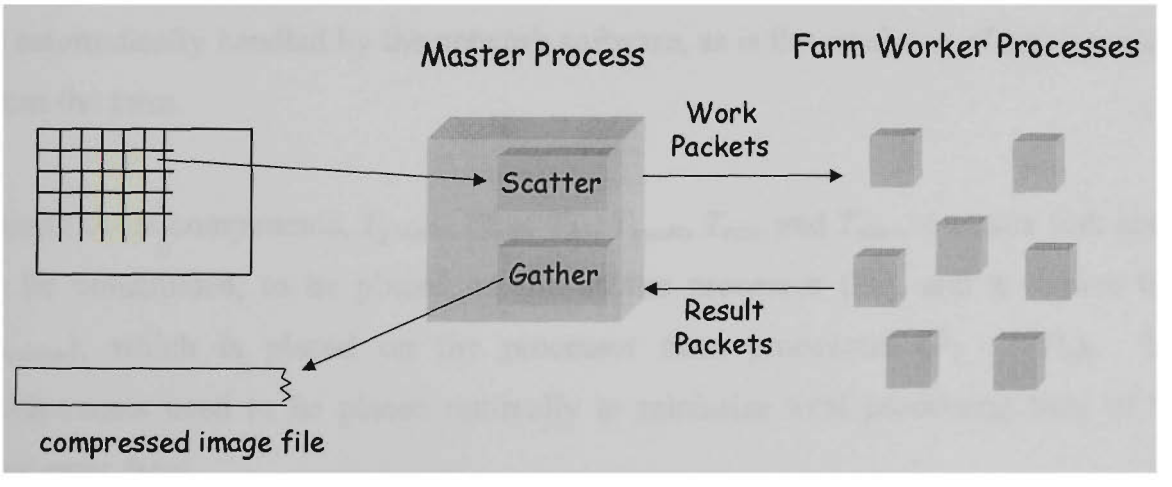


Figure 4.12 Application of processor farm technique to image compression

Algorithm SV1, implementing the JPEG baseline DCT algorithm, fits the processor farm paradigm perfectly. This algorithm breaks an image into blocks and then performs the same processing on each block independently. Algorithm SV1 can easily be reorganised into a *master* and *farm worker* task scenario, with the master task decomposing the image into blocks, and the farm worker tasks processing the blocks of the image. The image blocks become the work packets of Figure 4.12, while the processed blocks being returned become the result packets. In fact, a processor farm type implementation requires only slight modification to algorithm PV1.

4.7.3 Development of Processor Farm Algorithm

The transputer system provides built in support in the configuration software for the construction of processor farm algorithms. The master and worker tasks must be constructed, and then specified as master and worker tasks of a processor farm in a configuration file (see Appendix B.6). The application is then compiled with the configuration file. When run on the transputer, the *afserver* software running on the host CPU automatically places the master task on the root processor (P_0). A copy of the worker task is placed on all remaining processors (P_1 ... P_n), which are configured as the processor farm. Communication is facilitated to and from the farm by the use of *net_send()* and *net_receive()* function calls. The message routing for work packets is automatically handled by the network software, as is the receiving of result packets from the farm.

From the six components, T_{gblock}, T_{lshift}, T_{dct}, T_{quant}, T_{code} and T_{store}, a master task needs to be constructed, to be placed on the master processor (P_0), and a worker task (t_{worker}), which is placed on the processor farm processors (P_1 ... P_n). The components need to be placed optimally to minimize total processing time of the processor farm.

It is only during the image block decomposition (T_{gblock}) and the block storage (T_{store}) tasks of Figure 4.2 that the algorithm must process blocks in a particular order. In many multi-processor systems, the I/O devices are accessible from one processor

only. For these reasons, the tasks T_{gblock} and T_{store} are placed on the master processor P_0. That is

$$T_{gblock}, T_{store} \rightarrow P_0 . \tag{4.15}$$

The worker task, t_{worker}, is constructed from the remaining components, and placed on the processor farm. That is

$$t_{worker} \rightarrow P_1 \ldots P_n . \tag{4.16}$$

Figure 4.3 and the development of PV1 in Section 4.5, shows that the maximum degree of parallelism is gained from placing the component with the largest processing time (T_{dct}) on as many worker processes as possible. What is not clear at this point is, whether with n processors ($n > 2$) unlike in PV1, T_{dct} should be the only component placed in the worker task. This will depend on the number of processors in the farm, as the following discussion shows.

Note : The following discussions assume that given a network of processors $P_0 \ldots P_n$ on which the processor farm will be implemented, the speed of the processors is uniform across the network.

The T_{dct} component of Figure 4.2 is assigned to t_{worker}. That is

$$T_{dct} \rightarrow t_{worker} . \tag{4.17}$$

The remaining non I/O components are assigned to the master processor, with T_{gblock} and T_{store} already assigned in (4.15). That is

$$T_{lshift}, T_{quant}, T_{code} \rightarrow P_0 . \tag{4.18}$$

The five components assigned to P_0 are organized into two concurrent, running threads t_{snd}, t_{rec} (see section 5.2 for explanation of threads), which together make up the master task. The purpose of the two separate threads is so that the master task can

independently send and receive work packets to and from the processor farm, which contains multiple copies of t_{worker} (4.16). The thread t_{snd} is responsible for breaking the source image into blocks and passing them on over the processor farm network to the t_{worker} tasks. Since the t_{worker} task only contains the T_{dct} component (4.17), then we have

$$T_{gblock}\ ,\ T_{lshift}\ \rightarrow t_{snd}. \tag{4.19}$$

The t_{rec} thread receives processed work packets from the processor farm and stores the results in the compressed image file. Thus

$$T_{quant}\ ,\ T_{code}\ ,\ T_{store}\ \rightarrow t_{rec}. \tag{4.20}$$

With the configuration of tasks as described in (4.17, 4.19, 4.20), the t_{snd} component of the master task can contain a loop to prepare image blocks for processing by the farm. Since it is running independently from the rest of the master task, when the farm is saturated with work, (i.e. there are no free worker processors), this thread will become idle and not use any processor time. The t_{rec} thread accepts processed blocks from the processor farm, performs some processing of its own and stores the compressed blocks to disk. When there are no processed blocks to accept from the farm, (i.e. the worker processors are busy), then this thread will become idle and not use any processor time. These two master task threads work in harmony in the processor farm paradigm.

Parallel algorithm PV2 was developed from SV1 and designed for the configuration just described. Using the basic components of SV1, it implements the master task as two threads, a *send* and *receive* thread, and a worker task containing the FDCT component of SV1 only. The structure of PV2 is shown in Figure 4.13. The master task is located on processor P_0 (TRAM 0) of the actual transputer in Figure 4.1, while the worker tasks are allocated to all other processors P_1 ... P_n in the transputer. The transputer configuration file can be seen in section B.6, while the corresponding program code for the two tasks can be seen in section B.7.

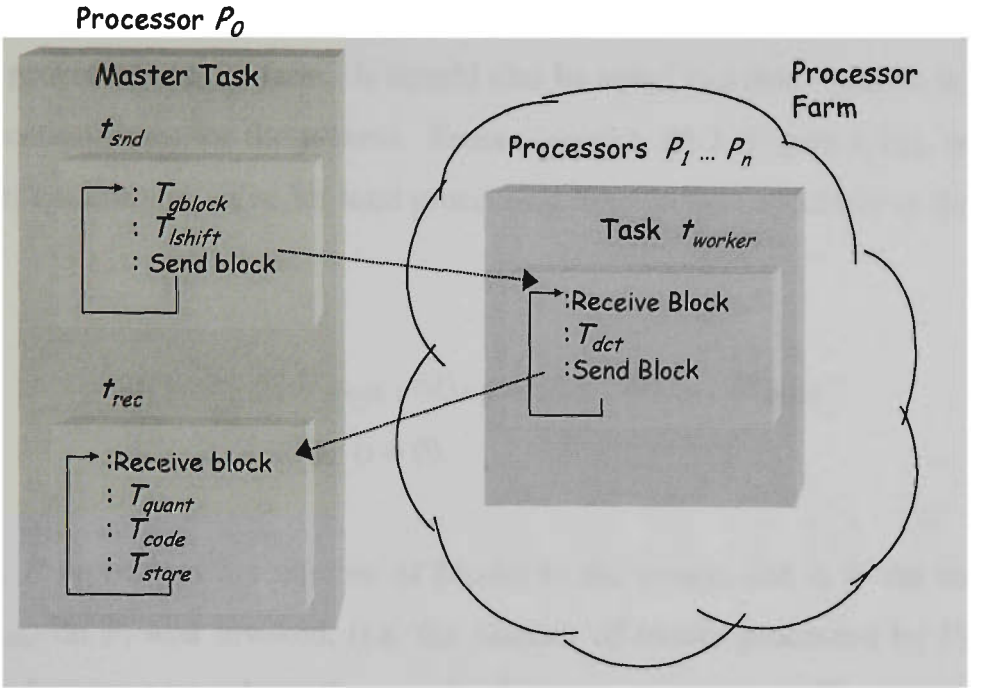


Figure 4.13 Structure of parallel algorithm PV2

We will now show that when implementing the JPEG algorithm using the processor farm paradigm with a specific configuration, as in Figure 4.13, there are a maximum number of processors which can be effectively used in the farm. This number of processors is called the *saturation point*, and its value is denoted with the symbol *sp*. Also, the configuration of the processor farm in Figure 4.13 is optimal, with regard to the placement of components, up to saturation point. Once this point has been reached, a re-arrangement of the components over the tasks of the processor farm may help to extend the saturation point.

4.7.3.1 Saturation Point

We can generalize (4.2) to represent the total processing time (ΔP) of the processor farm algorithm in Figure 4.13 as

$$minimum(\Delta P) \geq maximum \{\Delta P_i\}_{i=0}^{n} \tag{4.21}$$

where n is the total number of processors. Thus P_0 represents the master processor, and $P_1 \ldots P_n$ are the worker processors. As before ΔP_i is the total processing time on

processor P_i. Inequality (4.21) is similar to (4.2) except that it allows for the general case of n processors in the farm. It should also be noted that this equation is ignoring communication times for the present. From algorithm PV2 (Figure 4.13), and (4.21) we can set a minimum value for total processing time on the processors in the farm as follows

$$\begin{aligned} \Delta P_0 &\geq B(\delta T_{gblock}, \delta T_{lshift}\, \delta T_{quant}, \delta T_{code}, \delta T_{store}) \\ \Delta P_i &\geq k_i\, \delta T_{dct} \quad (\mathrm{i} \neq 0). \end{aligned} \tag{4.22}$$

In (4.22), B represents the number of blocks in the image, and k_i is the number of times t_{worker} on P_i was invoked, (i.e. the number of blocks processed by P_i). On a network of processors where the speed of the processors is uniform, it is fair to assume that over the compression of the entire image, a processor farm scheduler will load balancing with respect to the activities of the farm. That is, the scheduler will uniformly distribute the block processing over all processors $P_1 \ldots P_n$ in the farm. The transputer system does attempt to achieve this, [46] [45].

This implies that $k_i \approx (B/n)$. Thus

$$\Delta P_i \approx \frac{B}{n} \delta T_{dct} \quad (i \neq 0). \tag{4.23}$$

So, from (4.22) and (4.23) there must exist a value, sp, for the number of processors (n) in the farm such that

$$\begin{aligned} &\{\Delta P_0 < \Delta P_i \quad (n = sp - 1) \\ &\{\Delta P_0 \geq \Delta P_i \quad (n = sp) \end{aligned} \tag{4.24}$$

This value for n is the saturation point sp and is the point when the total processing on the master processor P_0 becomes greater than that of the worker processors $P_1 \ldots P_n$. Thus, ΔP_0 then becomes the lower bound for the minimization of ΔP in (4.21). This is plotted using a comparison bar chart shown in Figure 4.14.

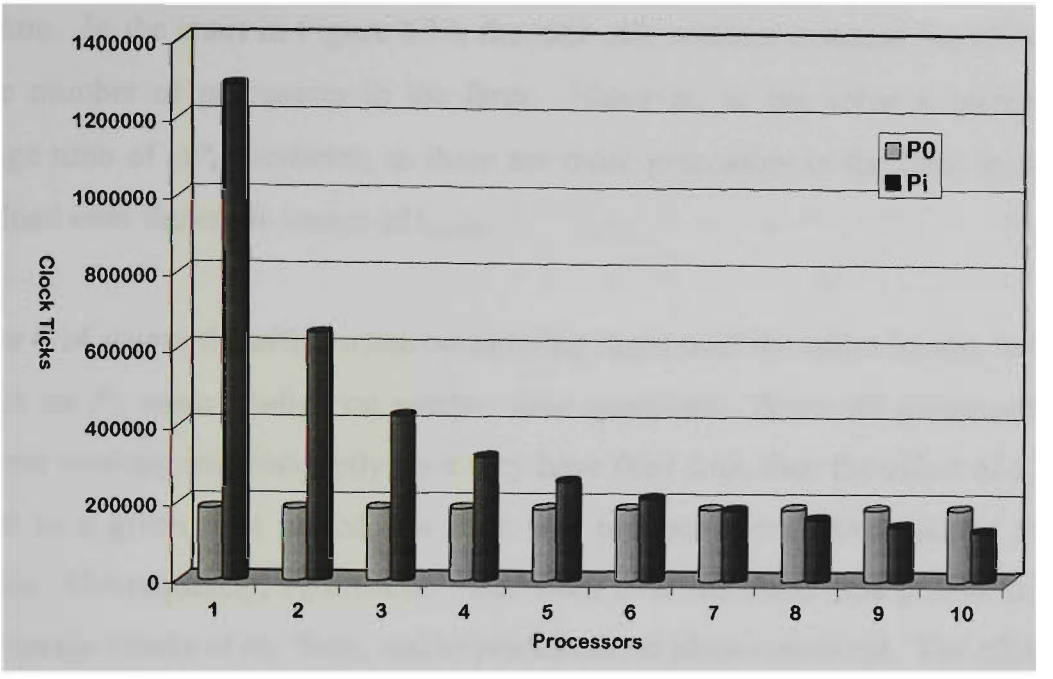


Figure 4.14 Comparison of overall processing times of P_0 vs P_i

From the algorithm PV2 and (4.22), ΔP_0 remains constant, regardless of how many processors are added to the farm. This figure is compared to ΔP_i ($i \neq 0$) from the farm, derived from the values of Table 4.2 and (4.23), using an image with 4096 blocks. From observation of the comparisons in Figure 4.14, the value *sp* appears to be seven.

The values used in Figure 4.14 are those from the original timing results of SV1 on P_0. For a processor farm implemented on the transputer system in Figure 4.1, the chart in Figure 4.14, does not take into account the varying processor times measured in section 4.6.1, nor the communication times as measured in section 4.6.2. These two factors will be considered when evaluating estimated processor farm times to be compared to actual measured times. The above chart assumes component processing times as measured in Table 4.2 remain uniform over all processors in the farm. It is expected that when accounting for variations in processor speed and communication times, the impact would be a reduction in the processing times of P_i in Figure 4.14, possibly resulting in a smaller value of *sp*.

The value of *sp* observed in Figure 4.14, can also be arrived at in a different manner. If the concept of a saturation point is correct, then the same value of *sp* should be derived by considering the measured idle time of processor P_0 over a given time

quantum. In the chart in Figure 4.14, the time ΔP_0 remains constant for all values of n, the number of processors in the farm. However, as the value n increases, the average time of ΔP_i decreases, as there are more processors in the farm to share the total load over the entire image of t_{worker}.

Figure 4.14 shows the effect when considering times over the entire image, but not the impact on P_0 when studied on smaller time quantums. Since all processors in the farm are working independently once they have their data, then the effect of a larger n is that in a given time period, the farm will produce more work packets for P_0 to process. Consequently, P_0 must do more work over the same time period to provide more image blocks to the farm, and to process those blocks received. The effect on P_0 is that the idle time decreases as n increases. Thus when the idle time on P_0 reaches zero, the processor farm has reached a point where the master processor cannot keep up with the rate at which the farm workers are producing and requiring work packets. This point should correspond to the saturation point observed in Figure 4.14.

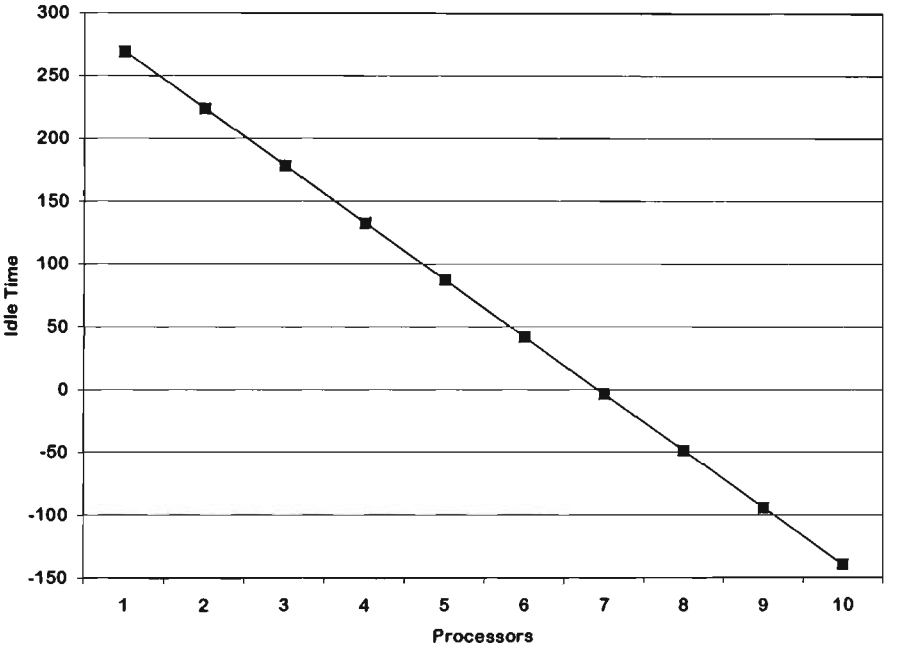


Figure 4.15 Expected idle time on P_0

Idle time on a processor over a given time period cannot be measured in itself, but can be derived by measuring or estimating the expected processor activity time over the same time period. Figure 4.15 plots the expected idle time on P_0 against the number

of processors n, in the farm. This graph is based on values from Table 4.2, and is derived from the expected activity on P_0 over the time quantum equivalent to δt_{worker}. Since we are assuming uniform processor speed in the processor farm, this is the time it will take the farm to produce n worker packets for P_0 to process. Once these packets are available for P_0, the master processor then sends the next n prepared blocks to the farm. In the next δt_{worker} time quantum, P_0 must then process these n transformed blocks, store them and prepare the next n image blocks for transfer. This is based on expected behaviour of algorithm PV2.

When n is small, from Figure 4.3, P_0 will have idle time when waiting for the farm to send work packets. As n increases, this idle time decreases as shown in Figure 4.15. This represents the extra activity P_0 must perform, associated with the extra processors, in order to keep up with the output from the processor farm. As in Figure 4.15, when n gets large enough, the output from the farm is too great for P_0 to process within the δt_{worker} time quantum. This point is represented in Figure 4.15 by the point where the graph crosses the zero-Idle time line. This is the saturation point, and in Figure 4.15 occurs at (n = 7). This agrees with the figure observed in Figure 4.14.

In Figure 4.15, the graph is approximately linear. This is to be expected, as for each new processor, there is an extra amount of work to do. From Table 4.2, this should increase as a simple multiple of the amount of work expected from one processor. The continuation of the graph after saturation point indicates negative idle time. Negative idle time in reality does not exist, but does represent the rate at which processing on P_0 falls behind that of the rest of the farm. This negative idle time on P_0 can also be viewed as the increase in idle time on the processor farm, resulting from P_0's inability to keep pace with the farm. This is explored further in Chapter 5.

4.7.3.2 Expected vs Measured times of PV2

We now need to compare the estimate for the processor farm timing against measured results in order to gauge the accuracy of predicted results for the saturation point above. To do this, some of the equations developed in section 4.6.2 need to be

enhanced to cater for a processor farm with n independent processors. Taking into account (4.23), (4.10) becomes

$$\Delta P_0 = \sum_{i=1}^{p} \delta t_0(i) + \frac{1}{2} \sum_{blocks} \delta C_{proc} \qquad (4.25)$$

$$\Delta P_i = \frac{B}{n} \delta T_{dct} + \frac{1}{2n} \sum_{blocks} \delta C_{proc} \quad (i \neq 0),$$

where B is the number of blocks processed by the farm, p is the number of tasks on P_0, and n is the number of processors in the farm. In (4.25), calculation of P_0 is the same, indicating that one half the processor involvement of communication is allocated to P_0. The other half is allocated to the farm processors. However, in the calculation of P_i, since there are n processors in the farm, this half of the processor communication involvement allocated to the farm is shared between all processors of the farm. This assumes a uniform distribution of image blocks to the farm processors.

When calculating ΔP, the ΔC_{tmit} component of the communication must also be taken into account, but it will depend on which quantity ΔP_0 or ΔP_i, is the greater as to how ΔC_{tmit} is distributed. For instance, since all blocks are communicated to and from the farm, if ΔP_0 is the greater, then when calculating ΔP the ΔC_{tmit} component can be seen as time when P_0 is waiting for communication from the farm, and thus should be added to ΔP_0 , as in section 4.6.2. However, if ΔP_i is the larger, then since on average B/n blocks are processed by P_i, then P_i will only have been waiting for $\Delta C_{tmit}/n$. Thus, the ΔC_{tmit} component is shared between the processors of the farm. Considering this, we can revise (4.11) for the processor farm to give

$$minimum(\Delta P) \geq maximum\left(\Delta P_0 + \Delta C_{tmit} , \Delta P_i + \frac{\Delta C_{tmit}}{n} \right). \qquad (4.26)$$

When calculating an expected figure for ΔP, since from section 4.6.1 the processors of the farm run at different speeds, the average time of δT_{dct} on P_1 and P_2 from Table 4.5 of 240.251 clock ticks will be used. In the actual transputer configuration of

Figure 4.1 there are 2 processors to use in the farm, and one processor for the master. Given the configuration of PV2 in Figure 4.13, we can calculate

$$\Delta C_{tmit} = 187,53.5 \text{ cts}$$
$$\Delta C_{proc} = 94,772 \text{ cts.}$$

Using the above and values ($n=2$) and ($B=4096$) in (4.25) and (4.26) we get ΔP is 503,780 cts.

The actual measured running times for PV2 was gained as an average of five trial runs as shown in Table 4.9. Using this average figure we see that our actual measured running time is 86.4% of the estimated running time of 503,780cts. Again, this is a better than expected actual running time, even taking into account the different processor speeds and communication times of the actual transputer in Figure 4.1.

Table 4.9 Overall timing results for PV2

Trial 1 (cts)	Trial 2 (cts)	Trial 3 (cts)	Trial 4 (cts)	Trial 5 (cts)
439,694	433,439	434,546	433,797	435,275
			average	435,350.2

Further investigation into the implementation of the processor farm paradigm on the transputer [68] reveals that during placement of the worker tasks $t_{worker} \rightarrow P_1 \ldots P_n$, a copy of t_{worker} is also placed on P_0. This copy of t_{worker} is run when there is idle time on P_0. Using the idle times calculated in section 4.7.3.1, it is estimated that with two processors in the farm, there is enough idle time on P_0 to process approximately 780 image blocks using the δT_{dct} on P_0 time from Table 4.5. Since the t_{worker} task on P_0 has no communication times then we can now calculate

$$\Delta C_{tmit} = 15,118.262$$
$$\Delta C_{proc} = 7,640.113.$$

Using the above and values ($n=2$) and ($B=3316$) in (4.25) and (4.26) we get ΔP is 407,805 clock ticks. This is 93.7% of the measured time.

However, while the result is much more accurate, the distance factor in the communication times has been ignored. This implies that all distances are one, and thus the processor farm topology is the star configuration shown in Figure 4.16, with $P_1 \ldots P_n$ a distance of one away from P_0. This is clearly not the case in the actual transputer configuration of Figure 4.1. If these extra transmission times are taken into account, then it is expected that the accuracy figure above would improve slightly. A preliminary figure for this was calculated at 95.8%. This figure was gained by adding to the 407,805cts above, the extra transmission and processing times of 7590cts and 3838cts respectively, in sending one half of the 3316 blocks processed on to a further node in the transputer network. This is an estimate of the behaviour of t_{worker} on P_0.

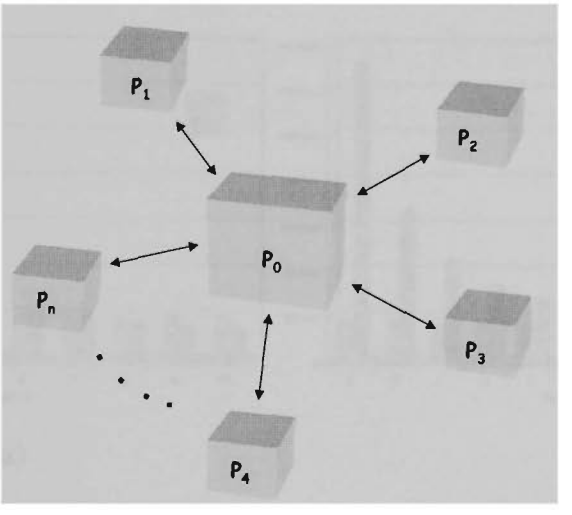


Figure 4.16 Assumed configuration of processor farm

4.7.3.3 Impact of Saturation Point on Optimal Task Placement

If communication times are ignored, and the star configuration of Figure 4.16 is assumed for the processor farm on which PV2 is implemented, then the effects which the saturation point, *sp*, has on the task placement in (4.17), (4.19) and (4.20), can be more easily investigated.

Figure 4.17 shows the comparison of overall processing times of P_0 vs P_i given a number of different component placements on task t_{worker}. Figure 4.17(a) is the same as Figure 4.14. It shows the comparison with the configuration of t_{worker} as in (4.17). This shows the saturation point sp, as in section 4.7.3.1, to be seven. Figure 4.17(b) is derived from the same values, except the configuration of t_{worker} is

$$T_{lshift}\ ,\ T_{dct} \rightarrow t_{worker}. \tag{4.27}$$

As can be seen in the comparison bar chart, by this rearrangement of components on the t_{worker} task, the saturation point has been increased to nine. However, it can also be seen that when the processor farm has the same or fewer processors than the saturation point value in part (a), the time of δP_i has increased thereby increasing the time of ΔP.

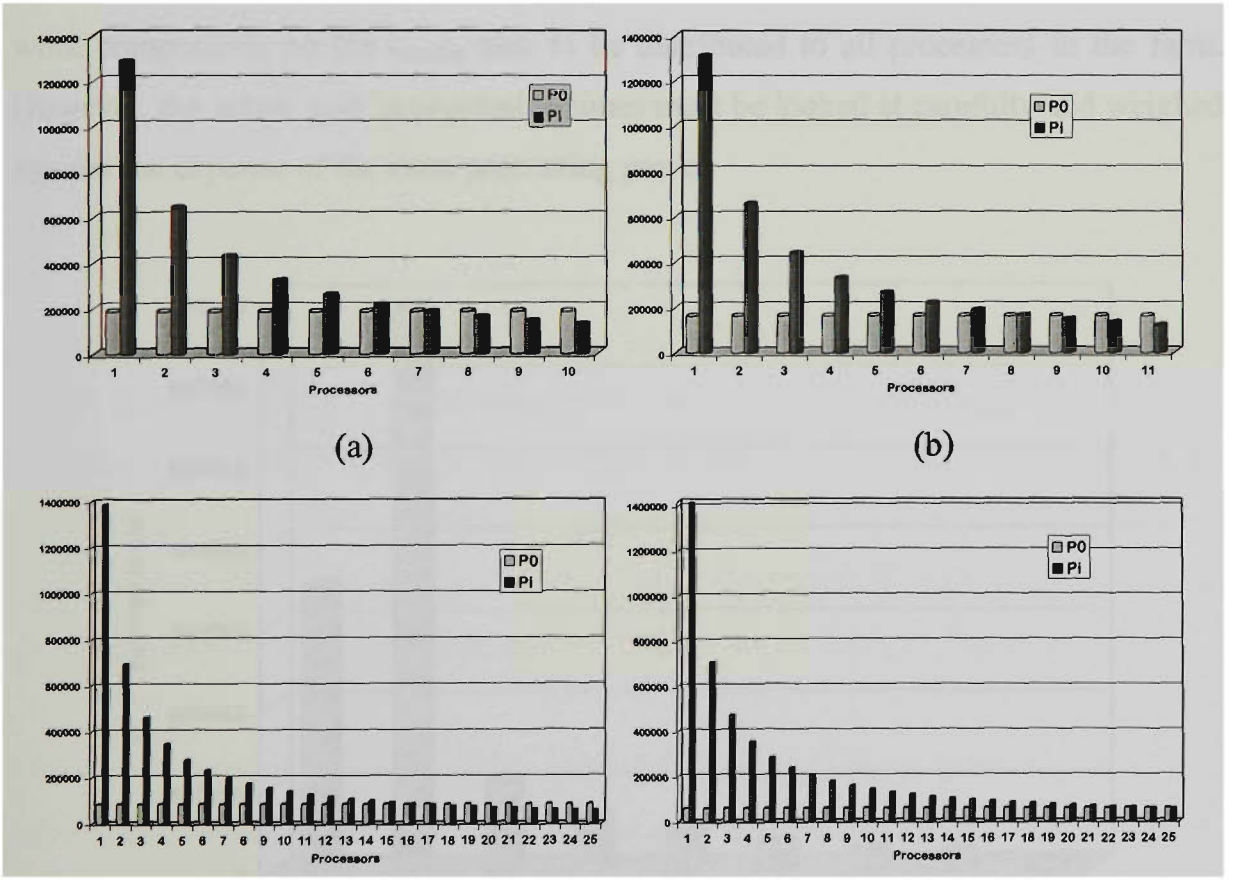


Figure 4.17 Comparison of saturation points with different task configurations

Similarly, Figure 4.17(c) is the chart representing the t_{worker} configuration of (4.28), while Figure 4.17(d) is the chart representing the t_{worker} configuration of (4.29). While in each case the saturation point has increased, to 17 processors in (c) and 24 processors in (d), the corresponding value of δP_i in each of the charts and hence that of ΔP increases, up to and including the saturation point of the previous chart.

$$T_{lshift}, T_{dct}, T_{quant} \rightarrow t_{worker} \tag{4.28}$$

$$T_{lshift}, T_{dct}, T_{quant}, T_{code} \rightarrow t_{worker} \tag{4.29}$$

Thus, the original configuration of components of PV2 in section 4.7.3 is optimal up to the saturation point of the number of processors in the farm. As stated before, the saturation point is the number of processors that the processor farm can effectively use. Once reached, the saturation point can be increased somewhat by placing more work components on the t_{worker} task to be distributed to all processors in the farm. However, the actual gain in processing times must be looked at carefully and weighed against the expense of the extra processing power.

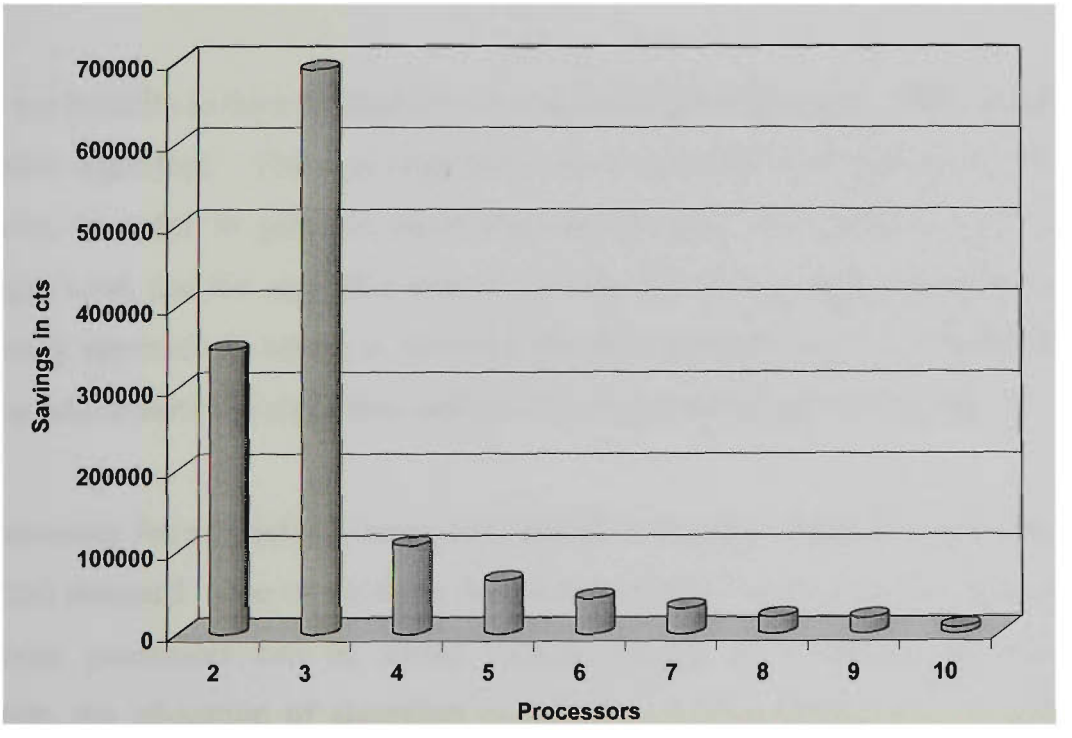


Figure 4.18 Progressive algorithm time savings vs number of processors

Figure 4.18 shows the progressive time savings gained at each step by adding another processor, from algorithm SV1. The savings gained for the number of processors of two and three in Figure 4.18 are based on the actual time savings of PV1 and PV2. All other values are estimated from timing values obtained from the different processors, assuming a uniform processor speed and ignoring communication times. The effect of those factors would be to decrease slightly the savings for processor numbers from 4 to 10. However, the difference would be insignificant.

As expected, adding the first two worker processors gives the most gain in time savings. There is a dramatic decrease in the time savings gained when adding the fourth processor, and savings continue to decrease up to the saturation point. When the saturation point is increased by allocation of more components to t_{worker}, the time savings continue to decrease up to the new saturation point. As the number of processors in the processor farm gets larger, the cost of incurring the expense of another processor must be weighed against the ever decreasing time savings.

4.8 Conclusion

There are benefits in time savings to be gained in implementing the JPEG standard as a parallel algorithm. These savings have been quantified throughout this chapter. However, in order to gain the most from parallelism, care must be taken in the paradigm used, and the algorithm design or "split up" on a multiprocessor system. If the wrong approach is taken, as with the pipeline method in 4.7.1, bottlenecks can develop which slow the algorithm and erode the main advantage of parallelism.

The processor farm paradigm seems well suited to image compression as outlined by the JPEG standard. One of the main advantages of this paradigm is that it is general, and more processors can be added without having to re-design the algorithm. However, the allocation of algorithm components to the worker task, as shown in sections 4.5 and 4.7 is very important. Allocating the wrong components to the worker task results in very little gain in the parallel algorithm over the single processor version.

There is a saturation point in the number of processors that can be effectively used in the processor farm paradigm. The saturation point is also irrespective of the image size. That is because the saturation point measures the maximum rate of work packets arriving from the farm against the rate at which the master processor can process the work. As shown in section 4.7.3.1 and 4.7.3.3, there is an optimal allocation of components to the farm, up to saturation point. Once this point has been reached, a reallocation of components can extend the saturation point, but at very little gain in overall performance. This is quantified in section 4.7.3.3.

The algorithm developed in section 4.7.3 was tested and run on the transputer system in section 4.3. This system had three processors, thus the data collected and measured from running the processor farm on available hardware could only be verified up to and including two processors in the farm. Results for more than that were obtained by extrapolating the measured times on the existing three processors. In order to validate these results, a method for simulating a processor farm is developed in Chapter 5, using the Java language, and the results of the simulation compared with initial results obtained in this chapter.

An area for further study arising from the work in this chapter would be to extend the algorithm PV2 for self-configuration. If the algorithm could detect the number of processors on initiation, then components could be allocated to the processor farm or the master processor to optimally configure itself. If more processors were available for the farm than the saturation number, then other components could be configured on the worker task to make better use of available processing power.

CHAPTER 5

PARALLEL JPEG SIMULATION USING JAVA

5.1 Introduction

The previous chapter presents a number of conclusions based on extrapolation of results obtained by the implementation of a parallel JPEG algorithm PV2, using three processors. It indicates a saturation point for the number of useful processors. To test the validity of the saturation point, the PV2 algorithm was simulated using Java. Experimental testing requires at least eight transputers, more than was actually available.

Java is a new programming language making in-roads into many areas of computing, particularly the Internet. Of the many features of Java, one of the least known is its multi-threaded capability. By exploiting Java's multi-threaded features, a simulation algorithm was developed which exploits the computing potential of the underlying platform, regardless of whether is single or multi-processor powered. The closeness of Java's association to the Internet, the ease of network programming that is provided by the language, and its platform independence suggests the potential for such an algorithm to use distributed processors. This would then be a JPEG distributed compression algorithm that is portable.

The simulation algorithm, JVS1, was implemented to mimic the processor farm based parallel algorithm PV2. A method was devised to measure the saturation point, and the results running JVS1 using a number of simulated processors are presented. To measure when the saturation point had been reached, the simulated message queues were monitored at the equivalent of 30 millisecond intervals on a multi-processor

system. The amount of accumulated messages in the queues was then used as a measure to indicate the proximity to the saturation point *sp*.

The main requirement of a good simulation is that it enables the production of phenomena likely to occur in operational circumstances, under test conditions. In simulating PV2, the main requirements are that of processor load and communication timing between the processors. In chapter four, it was seen that the communication times played an important role, but if the number of processors was small, the main consideration was that of processor load. The simulation JVS1, concentrates mainly on simulating processor load of the processor farm paradigm for the extension of PV2. However, a technique is discussed for the inclusion of communication times.

Sections 5.2 and 5.3 provides some preliminary work on implementation of the tasks of PV2 as Java threads and their scheduling in the Java Virtual Machine; this constitutes the heart of the Java simulation JVS1. Section 5.4 discusses the complete development of JVS1, and some of the problems encountered in the simulation are presented in 5.5. The simulation results and discussion are presented in section 5.6.

5.2 Implementation of Tasks as Java Threads

The processor farm paradigm is relatively easy to simulate in Java, since the worker tasks residing on processors $P_1 \ldots P_n$ are identical. The master processor P_0 is different, as indicated in Figure 4.13. The master task on P_0 is implemented as three independent Java threads, corresponding to the three parallel C tasks on the transputer, so each can run independently in the simulation. The first step in constructing the parallel simulation JVS1 was to therefore implement each transputer task as a Java thread. Threads were chosen rather than functions since they are relatively inexpensive to create in Java, and once instantiated they become separate entities.

The actual implementation of the tasks in Java was relatively straightforward. The transputer programs were written in a version of Parallel C [68], based on ANSI[1] C, and Java itself is based on the C/C++ language. Java is however a fully object-oriented language, so most of the transputer code could be used as is, but it had to be encapsulated within Java classes. Java classes extending the *thread* class were constructed to implement the four tasks identified above, and within the simulation, these classes were instantiated as Java objects. The thread class is a standard Java class. A constant n was uses to indicate the number of worker processors $P_1 \ldots P_n$, and the simulation simply created the n worker thread objects as required. The Java simulation code is included in Appendix C.

A thread, in general, is not the same as an operating system process, or a process that can be created in some programming languages, such as with the *exec()* function in C. A thread has its own execution path but shares memory with the process that created. It is therefore sometimes referred to as a lightweight process. A multi-threaded operating system allows each process to be divided into several components, which are called threads. The idea of threads is not a new one, especially the concept of a single thread, but multiple threads in a single program performing different tasks all executing at the same time has not yet become common.

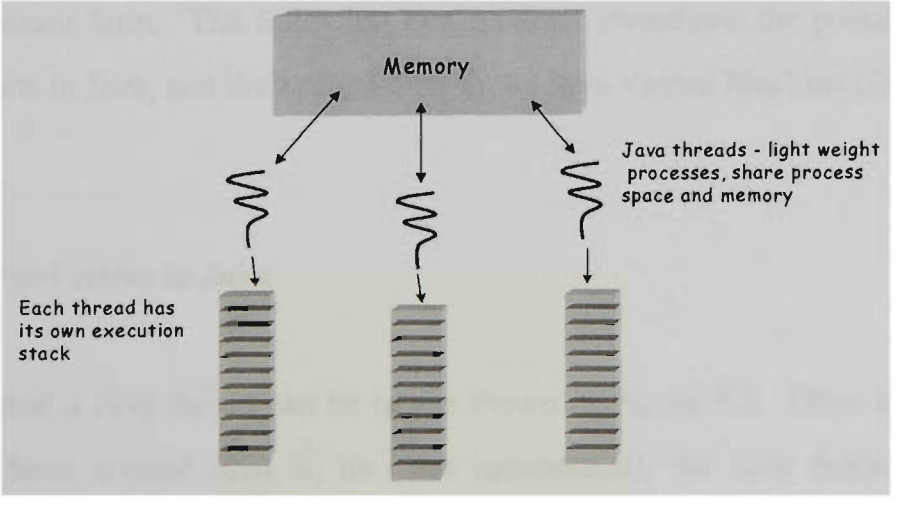


Figure 5.1 Java thread structure

[1] American National Standards Institute

A Java thread is a lightweight process that has its own execution stack, but shares memory and variables with the process that created it, as shown in Figure 5.1.

Transputer processes communicate by passing messages, and do not share memory. Since Java threads all share the same memory space, this has implications for the simulation of the communication aspects of the processor farm, when considering these communication aspects of a multi-processor system. A good simulation has to incorporate the delays due to processor communication. This component of the simulation is covered in more detail in a later section.

5.3 Thread Scheduling in Java

Once the four tasks identified in Section 5.2 are coded and instantiated as thread objects in Java, the threads need to be scheduled and run in a manner consistent with a correct simulation. On the transputer, low priority processes (including threads) are time-sliced to provide an even distribution of processor time between the processes [44]. If there are n processes, then the maximum latency between a process's time-slices is $2n-2$ with the timeslice period approximately 1ms. The scheduling of threads in Java needs to be investigated in order to devise a correct approach to the simulation of the processor farm. The following two sections investigate the possible states of thread objects in Java, and their scheduling in the Java Virtual Machine (Java VM).

5.3.1 Thread states in Java

The states that a Java thread can be in are shown in Figure 5.2. Once a new thread object has been created (that is, its class instantiated), the Java thread is initially placed in the *New Thread* state. In order to be instantiated (as a thread), the thread must implement the *public void run()* method. This is equivalent to a C program's *main()* function. It is the point where the thread begins execution.

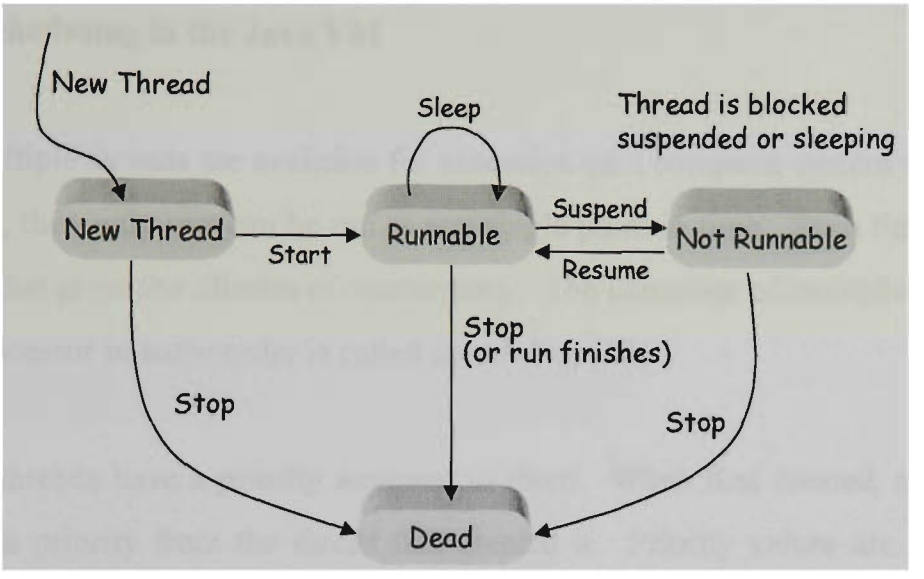


Figure 5.2 Java thread states

To place the thread in a *Runnable* state, that is the state where it will be scheduled for execution, the thread's *start()* method is invoked. From a Runnable state, a thread can move to a *Not Runnable* state when it becomes blocked on Input/Output (I/O) or a synchronized method; it has invoked its *sleep()* method; or it has been suspended by another thread. A thread dies when its *stop()* method is invoked or its *run()* method finishes execution.

A Java thread can control another Java thread (providing it has the security to do so), by invoking its *suspend()* and *resume()* methods. This will essentially move a thread to the Not Runnable state and back to the Runnable state. While these methods provide only limited control, they are flexible enough to allow the construction of a simulation algorithm.

When a thread is in a Runnable state, it is scheduled for execution along with all the other Runnable threads by the Java VM scheduler. Unfortunately, the behaviour of the scheduler is not clearly specified, and consequently may behave differently on different platform implementations.

5.3.2 Scheduling in the Java VM

When multiple threads are available for execution on a computer system with a single processor, then only one can be run at any single point in time. Such threads are run in a way that gives the illusion of concurrency. The execution of multiple threads on a single processor in some order is called *scheduling* [12].

All Java threads have a priority assigned to them. When first created, a Java thread inherits its priority from the thread that created it. Priority values are in the range from *MIN_PRIORITY* to *MAX_PRIORITY* (1 to 10) which are constants defined in the thread class *java.lang.Thread.* Thread priority can be dynamically changed by the programmer to any value between *MIN_PRIORITY* and *MAX_PRIORITY.*

The scheduling of threads in the Java VM follows a simple deterministic scheduling algorithm, known as *fixed priority scheduling* [30]. At any time, the thread that is currently executing is the thread with the highest priority amongst all the threads in a Runnable state. The scheduler is pre-emptive; therefore the currently executing thread will be pre-empted when a thread with a higher priority becomes Runnable. However, this is not guaranteed. The thread scheduler may choose to run a lower priority thread to avoid starvation.

When there is more than one thread in a Runnable state at the same highest priority, the Java VM thread scheduler chooses the next thread to run using a simple *non-preemptive* round robin scheduling order. These highest priority threads may or may not be preemptively time sliced. The Java VM does not implement time-slicing, and therefore does not guarantee time-slicing of equal highest priority threads. Instead, the Java VM relies on the architecture of the underlying operating system. This situation is depicted in Figure 5.3.

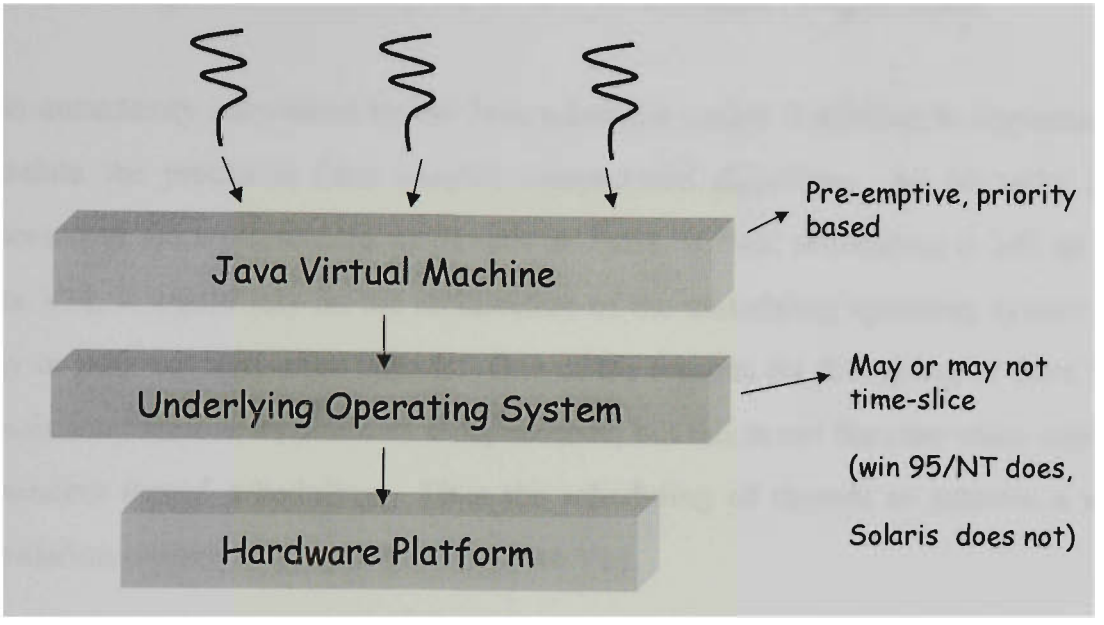


Figure 5.3 Java thread scheduling hierarchy

The "White Paper" on Java [30] states:

"Java's threads are pre-emptive, and depending on the platform on which the Java interpreter executes, threads can also be time-sliced. On systems that don't support time-slicing, once a thread has started, the only way it will relinquish control of the processor is if another thread of a higher priority takes control of the processor."

Windows 95 and Windows NT are both multi-threaded operating systems, and implement their own pre-emptive scheduling algorithms. On these platforms, multiple threads in Java running at the same highest priority will be time-sliced on a round robin basis, but there is no guarantee as to the order of this time-slicing. Java on Solaris does not time-slice [91]. On non time-sliced systems, a thread will run until it does one of the following:

- finishes,
- is terminated,
- is interrupted by a thread of higher priority,
- goes to sleep,
- blocks on I/O or a synchronized method,
- initiates its *yield()* method.

5.4 Development of Parallel JPEG Simulation Algorithm

This uncertainty introduced by the Java scheduler makes it difficult to implement and simulate the processor farm parallel compression algorithm. As all tasks (on all processors) are implemented as threads in JVS1, if their scheduling is left up to the Java VM, it would rely on the architecture of the underlying operating system. This may or may not time-slice threads. One of the reasons for the choice of Java for the simulation tool was its platform independence, but this is not the case when relying on consistent thread scheduling. Thus the scheduling of threads to achieve a correct simulation cannot be left solely to the Java VM.

Even if the Java VM could be relied upon for consistency in thread scheduling, once the thread objects have been created, we cannot just start all threads and let the Java VM scheduler take control. This would result in the simulation being skewed in so much as the simulated processors with more tasks would be allocated more simulated processor time than they would receive during normal operation. In the algorithm PV2, worker processors $P_1 \ldots P_n$ have only one task each, but processor P_0 has three tasks. In this situation, the simulation would allocate more time to P_0 distorting the observed results.

To see this, consider an abstract view of a multi-processor system shown in Figure 5.4. In this situation $P_0 \ldots P_n$ are running as independent processors. Thus after a specific time quantum Δt, each of the processors $P_0 \ldots P_n$ has been processing in parallel for time Δt. So for a particular processor P_i, the time quantum Δt would have been shared somehow between its tasks $T_1 \ldots T_p$. The sharing of Δt between $T_1 \ldots T_p$ is dependent on the behaviour of the process scheduler on processor P_i, but with small enough scheduling time-slices would be approximately $\Delta t/p$ per T_i. Similarly, if processor P_j had tasks $T_1 \ldots T_q$, Δt would have been shared between the tasks $T_1 \ldots T_q$, with approximately $\Delta t/q$ per T_j with small enough scheduling time-slices.

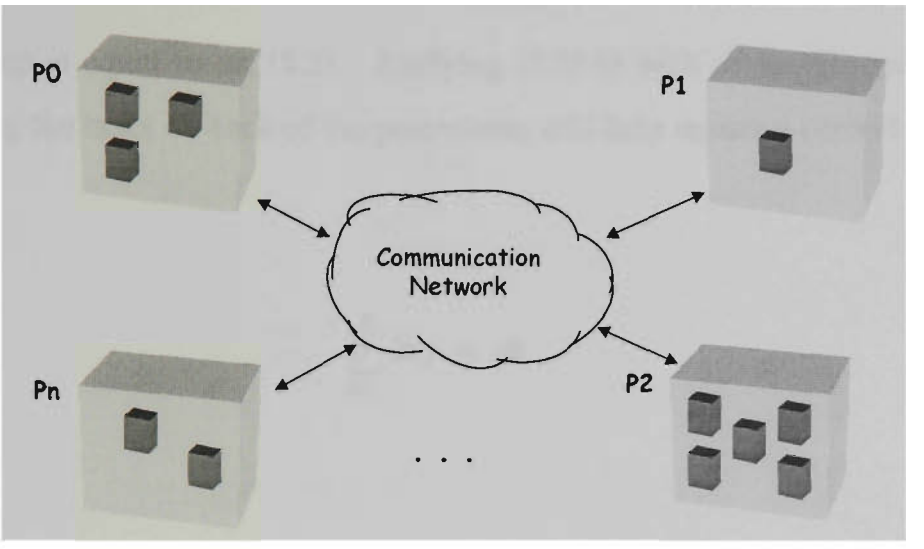


Figure 5.4 Abstract view of a multi-processor system

A distortion in a simulation of the above can occur if the tasks on all processors $P_0 \ldots P_n$ from Figure 5.4 are instantiated as threads and all allowed to be scheduled unsupervised by the Java VM (or underlying operating system) time-sliced scheduler. Assuming a set time quantum δt, each of the threads $\tau_1 \ldots \tau_p$, representing tasks $T_1 \ldots T_p$ from processor P_i, would be allocated the same time quantum, δt, in one round of scheduling. The set of threads $\tau_1 \ldots \tau_q$ representing tasks $T_1 \ldots T_q$ from processor P_j would also each be allocated a time δt. However if $p \neq q$, then

$$\sum_{1}^{p} \delta t \neq \sum_{1}^{q} \delta t \qquad (5.1)$$

Thus in the simulation, from (5.1), in one round of scheduling, one of the processors P_i, P_j would receive more processing time in the simulation, depending on which was the larger p or q.

To solve this problem, we need to be able to treat the set of threads $\tau_1 \ldots \tau_p$ representing tasks on processor P_i as a whole and allocate the time Δt to this group of threads as a whole. This will help ensure that over the simulated time period Δt, the threads $\tau_1 \ldots \tau_p$ belonging to the simulated processor P_i somehow share Δt. Each thread receives only a portion of Δt, and the sum of these portions over all the threads

in this group is equal to Δt, (5.2). Applying (5.2) to each of the groups of threads representing the tasks on each of the processors, will help ensure a correct simulation. Thus

$$\sum_{k=1}^{p} \delta\tau_k = \Delta t . \tag{5.2}$$

5.4.1 Thread Groups

Java provides for a convenient way with respect to the scheduling problem, in which to simulate the algorithm PV2, by using thread groups and their associated methods. Thread groups in Java are a way to combine particular threads together under a unifying umbrella. A thread can be placed into a thread group during thread creation, as in Figure 5.5. Once a member of a particular thread group, a thread is a permanent member of this group and cannot be moved to another.

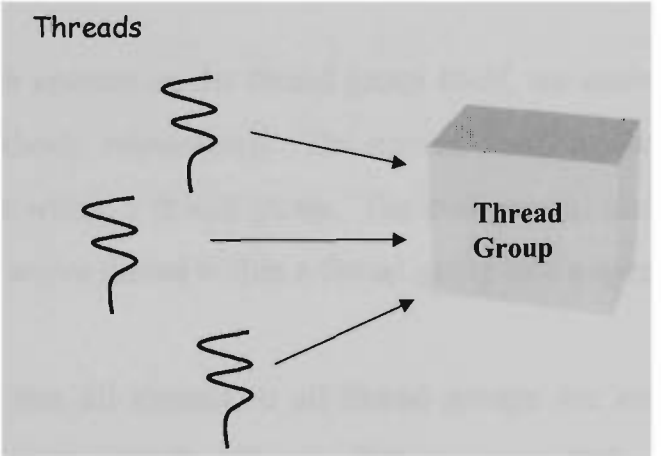


Figure 5.5 Threads assigned to thread groups

Unless a different thread group is specified, a thread is automatically placed into the same thread group as the thread that created it. The Java VM begins automatically with one thread group called *main*, and all new threads are placed in this group unless otherwise specified.

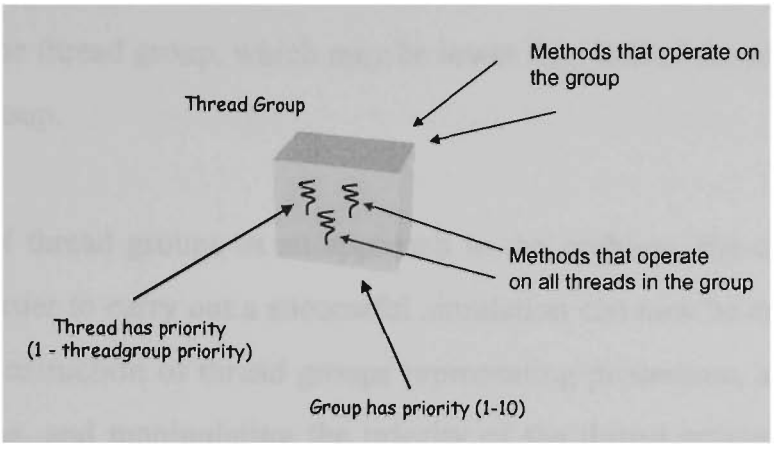


Figure 5.6 Operations on thread groups

To manipulate these thread groups and the threads contained within them, Java provides two classes of methods. These are methods that operate on the thread group itself, and methods that operate on the individual threads within the group, as depicted in Figure 5.6. Of the latter set, two, *suspend()* and *resume()*, are of particular use in constructing the parallel simulation with respect to the scheduling problem. These methods place all threads in a thread group into a Not Runnable or Runnable state respectively, thus effectively making the entire thread group either Not Runnable, or Runnable.

Two methods, which operate on the thread group itself, are *activeCount()* and one of the enumeration methods, *enumerate()*. The *activeCount()* method returns a count of all the active threads within a thread group. The *enumerate()* method is used to place a reference to every active thread within a thread group into a specified array.

It should be noted that all threads in all thread groups are treated equally by the scheduler, with respect to priority. That is, if there are multiple threads belonging to different thread groups, all Runnable, and all at the same highest priority, then all threads have an equal chance as being selected by the Java VM scheduler, again with respect to its scheduling algorithm. A thread group has a maximum priority associated with it, which specifies the maximum priority possible for all threads added to this thread group after this maximum priority limit was set. If one thread group has a maximum priority higher than another, it does not mean that its threads will be scheduled before the threads in another group. That depends on the actual priority of

the threads in the thread group, which may be lower than that of the maximum priority of the thread group.

With the use of thread groups as an approach to the problem, the objects necessary under Java in order to carry out a successful simulation can now be constructed. This involves the construction of thread groups representing processors, assigning threads to thread groups, and manipulating the priority of the thread groups. A scheduling algorithm drives the simulation by manipulating these Java objects.

5.4.2 Construction of Simulation Objects

To construct the objects necessary for the Java simulation, the initial simulation program first identifies itself, and its default main thread group, assigning the priority *MAX_PRIORITY* to both itself and the main thread group to which it belongs. This thread then becomes the controlling thread $\tau_{control}$. Thread groups $G_0 \ldots G_n$ are constructed which represent the processors $P_0 \ldots P_n$ in the processor farm paradigm, G_0 representing the master processor, P_0, and $G_1 \ldots G_n$ representing the worker processors $P_1 \ldots P_n$. Each of the thread groups $G_0 \ldots G_n$ are assigned the priority *NORM_PRIORITY*, thus all threads assigned to them will run at a lower priority than $\tau_{control}$.

The thread object τ_{worker}, representing worker task t_{worker} from PV2 is created n times and a copy placed in each of the worker thread groups, as in (5.3)

$$\tau_{worker} \rightarrow G_1 \ldots G_n \qquad (5.3)$$

Threads τ_{snd}, and τ_{rec} representing the *send* and *receive* tasks t_{snd} and t_{rec} of PV2, are created and assigned to the master thread group as in (5.4).

$$\tau_{snd}, \tau_{rec} \rightarrow G_0 \qquad (5.4)$$

Each of these threads automatically receive the priority of the thread group to which it is assigned (*NORM_PRIORITY*). When instantiated, all threads are placed into a Not Runnable state. This is achieved by initiating each thread's *start()* method immediately after creation and assignment to the appropriate thread group, then initiating its *suspend()* method. The thread will not sneak in any processing time as it is running with a priority less than the controlling thread $\tau_{control}$, and Java thread scheduling is fixed priority based.

The pseudocode for the creation of the objects for the Java simulation is given below in Figure 5.7.

```
// Create Simulation Objects

Identify current thread and its thread group
set priority of current thread and thread group to MAX_PRIORITY

create thread group G0 ... Gn with priority NORM_PRIORITY

create τsend thread and assign to G0
start τsend thread
suspend τsend thread
create τreceive thread and assign to G0
start τreceive thread
suspend τreceive thread

for each worker thread group G1 ... Gn
    create new instance of work thread τ and assign to Gi
    start τ thread
    suspend τ thread
```

Figure 5.7 Algorithm pseudocode for creation of simulation objects

When all thread groups representing processors and the threads corresponding to tasks on those processors have been instantiated, the situation is as depicted in Figure 5.8. There are now two general methods to proceed with the simulation. One involves letting the Java VM handle the scheduling of the threads within the thread groups, and allocating the thread groups a fixed time quantum. The other essentially involves writing a custom scheduling algorithm. Both of these approaches were implemented,

and their relative merits are discussed in the next section. The latter approach was adopted as it helps simulates the processor farm scheduling more accurately.

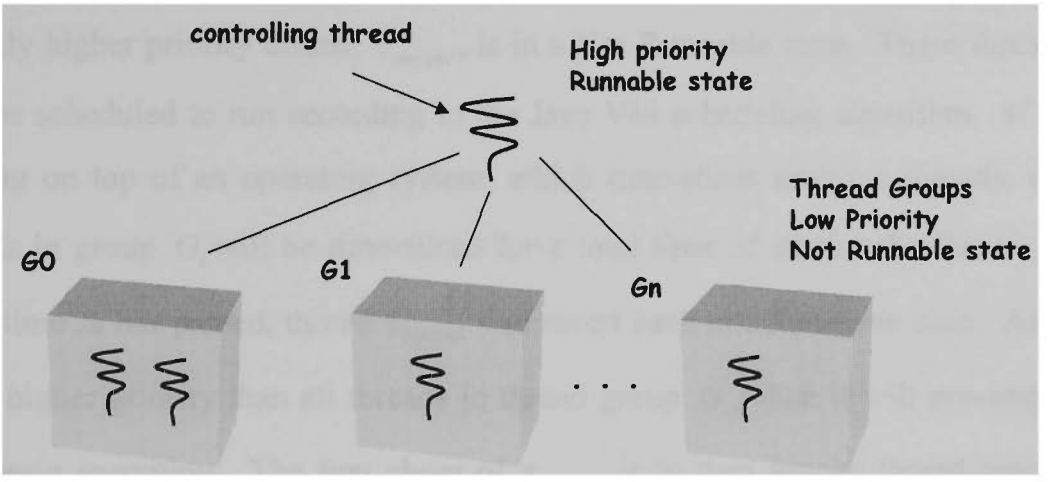


Figure 5.8 Structure of objects in Java simulation

5.4.3 Simulation Algorithm

To construct the simulation algorithm that will use the simulation objects created in the previous section, there are two possible approaches that were both implemented and tested in algorithms SIM1 and SIM2 respectively. Both of these are now discussed in detail.

Algorithm SIM1:

The simple approach of this algorithm is to allocate a specific time quantum, Δt, to each of the thread groups $G_0 \ldots G_n$, on a time-sliced round robin basis. To do this, the controlling thread $\tau_{control}$ must implement a simple loop that involves all the thread groups. Then $\tau_{control}$ calls in turn each of the thread group's, $G_0 \ldots G_n$, *resume()* methods, which places each thread in the group into a Runnable state. Now $\tau_{control}$ invokes its own *sleep()* method for the required time quantum Δt. This effectively places $\tau_{control}$ in a Not Runnable state for Δt milliseconds (assuming Δt represents a time quantum in milliseconds).

If thread group G_i is the one whose *resume()* method was just invoked, then all threads in this thread group are now in a Runnable state at equal highest priority, since the only higher priority thread, $\tau_{control}$, is in a Not Runnable state. These threads will now be scheduled to run according to the Java VM scheduling algorithm. If Java is running on top of an operating system, which time-slices multiple threads, then all threads in group G_i will be time-sliced for a total time of Δt (for the thread group). After time Δt has passed, thread $\tau_{control}$ will revert back to a *Runnable* state. As $\tau_{control}$ has a higher priority than all threads in thread group G_i, then it will pre-empt them and begin execution. The first chore of $\tau_{control}$ is to then invoke thread group G_i's *suspend()* method, which places all threads in that group back into a Not Runnable state.

Thus after one pass through the controlling loop in $\tau_{control}$, all processors have been simulated running for a time quantum of Δt, by allowing each to run for Δt on a round robin basis. No matter how many threads involved in thread group G_i, G_i is allocated Δt time to use among its threads according to the Java VM scheduling algorithm.

While SIM1 is simple to implement, this method has many problems. As discussed in section 5.3.2, the architecture of the underlying operating system may not support time-slicing multiple threads, in which case, the scheduling of threads within each thread group $G_0 \ldots G_n$ will be unfair. In the processor farm simulation, this would only be of concern in thread group G_0, since all worker thread groups only have one thread τ_{worker} assigned. Also, experimental trials [20] have indicated that if a thread group G_i of threads $\tau_1 \ldots \tau_n$ are given a time quantum Δt in which to run and then suspended, when resumed, processing begins back with thread τ_1, regardless of which thread was executing when G_i was suspended. Thus, even if the underlying operating system does time-slice multiple threads, if Δt is the wrong size, the scheduling

algorithm may not get around to giving each of the threads $\tau_1 \ldots \tau_n$ an equal share of the time quantum Δt. This again results in unfair thread scheduling.

Algorithm SIM2:

The approach of algorithm SIM2 is an extension of SIM1 and provides a general simulation algorithm that does not allow unfair thread scheduling. This provides a platform independent, simulation algorithm that can be used to simulate multi-processor paradigms other than the processor farm type paradigm used here.

As before, thread $\tau_{control}$ implements a simple loop to process each of the thread groups $G_0 \ldots G_n$, and each pass through this loop represents a simulation of the multi-processor system of one time quantum Δt. The time quantum Δt is not allocated to each thread group G_i as a whole, but a portion of it to each thread $\tau_1 \ldots \tau_n$ in G_i. For each thread group G_i, the number of active threads in the group is determined by calling the groups *activeCount()* method. This method returns the number of active threads in G_i, m_i. Using the value m_i, an array is constructed and G_i's *enumerate()* method is called, which places a reference to every active thread of G_i in the array. Another simple loop in $\tau_{control}$ now processes each of the active threads in this array by giving each of the threads τ_i, the specific time quantum of $\Delta t/m_i$ milliseconds to process.

Thus after one pass through this inner loop of $\tau_{control}$, all of the threads $\tau_1 \ldots \tau_n$ in G_i have been allocated an equal share of Δt. Note that threads are allocated time to run by using the same method used previously in SIM1 to allocate time to thread groups.

The pseudo-code for this simulation algorithm is shown in Figure 5.9. The simulation algorithm JVS1, is based on this method, SIM2, and the Java code for JVS1 can be viewed in Appendix C.

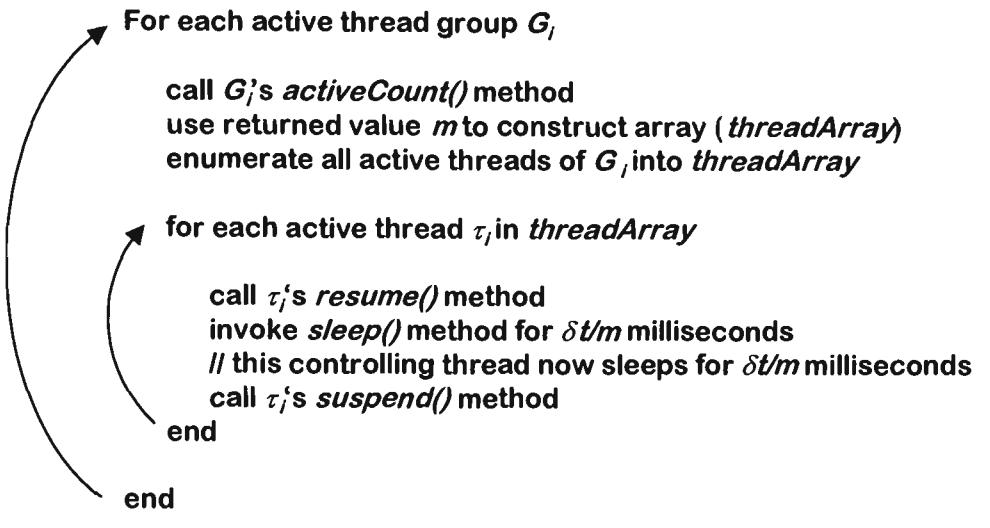


Figure 5.9 Pseudocode for simulation algorithm SIM2

Effectively this constitutes writing a new thread scheduling algorithm. It does however ensure a fairer distribution of the time quantum Δt amongst all threads in a thread group. The algorithm outlined in Figure 5.9 is also dynamic in that it takes into account the number of active threads m_i of a thread group when calculating the time quantum for a process, $\Delta t/m_i$. If a thread finishes, then m_i changes, thus altering the value of $\Delta t/m_i$. This allows us to simulate a multi-processor system correctly by ensuring fairer allocation of time to each simulated process in each simulated processor.

5.4.4 Processor Communication

A technique to simulate the message passing mechanism between processors used in the transputer processor farm implementation PV2, was developed in [20]. This technique involves the use of Java *Vectors* in the Java simulation. A Vector in Java is an implementation of a Dynamic Array. To simulate the transputer processor farm communication, the controlling thread $\tau_{control}$, must be able to make the next image block available to the first available worker thread τ_{worker}, in the farm. Also, each worker thread τ_{worker}, when finishing processing an image block, must be able to send its results to the controlling thread $\tau_{control}$, before grabbing the next image block to

process. This can be simulated by the implementation of two Java Vectors to act as I/O channels between the master processor G_0, and the worker processors $G_1 \ldots G_n$. This is shown Figure 5.10 below.

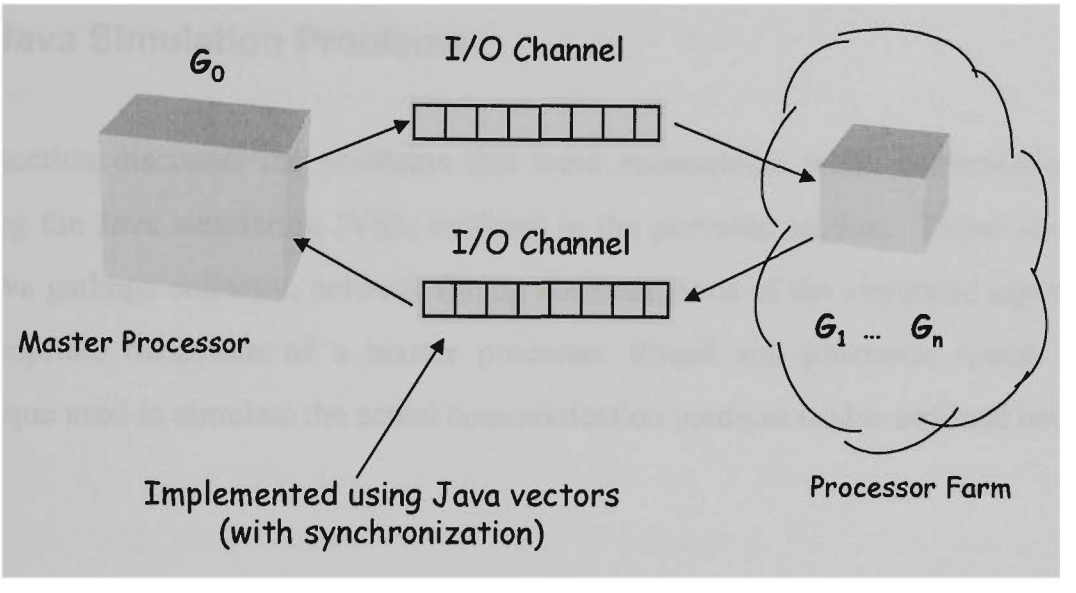


Figure 5.10 Use of Java vectors to simulate message passing

Java Vectors are well suited to this. Java provides methods to add elements to the rear of a Vector, and to remove elements from the front, effectively enabling the use of a Vector as a queue. Thus in the simulation shown in Figure 5.10, a thread in G_0 can add an image block to the rear of one Vector, treating it as an Output Channel, while any thread in any of the thread groups $G_1 \ldots G_n$ can remove an image block from the front of the same Vector, treating it as an Input Channel. In the same way, the thread groups $G_1 \ldots G_n$ can use the other Vector as an Output Channel, while G_0 can use it as an Input Channel.

There is another aspect of Vectors that particularly make them valuable as I/O channels in the simulation. Vectors can hold data of different types. This is unlike static arrays in other languages, which can only hold data elements all of the same type. Thus, like any true I/O channel linking processors, any type of data can be transmitted over the line, (i.e. any data object can be placed in a Vector). The only stipulation is of course that a thread reading an object from one of these vectors must

know what type of object it is reading. However this is also true of communicating processors, they must know what type of data they are transmitting to each other.

5.5 Java Simulation Problems

This section discusses the problems that were encountered when constructing and running the Java simulation JVS1, outlined in the previous section. These involved the Java garbage collector, network timing considerations of the simulated algorithm, inappropriate behaviour of a master processor thread and processor speed. The technique used to simulate the actual communication medium is also outlined here.

5.5.1 Garbage Collector

One of the most frustrating implementation problems encountered was the behaviour of the Java garbage collector. Java contains an automatic garbage collector, which runs periodically to gather free space left over by all memory objects that are no longer referenced. This is one of the attractive features of the language, which is also present in other object-oriented languages, and saves the programmer from having to specifically call a memory *free()* function, as is the case in *C*, after excessive use of *malloc()*. While this feature can normally be ignored, when trying to run the simulation and allocate time to threads in thread groups on a round robin basis, making sure that each thread received all the processing time allocated to it was crucial for the correctness of the simulation.

The Java garbage collector runs synchronously in a low priority thread when the system gets low on memory resources [12 pp 85]. However, due to the nature of thread scheduling in Java, (Section 5.3.2), a low priority thread can get processor time if the underlying operating system deems it necessary. This anomaly caused the garbage collector to interrupt the simulation often, causing the simulated task it

interrupted to miss out on some of its scheduled time quantum, thus slightly distorting the results of the simulation presented in the next section.

There is a work-around for this problem, which involves specifically calling the garbage collector in the program before a critical section of code. This was incorporated into the algorithm of the previous section but caused the Java VM to enter a deadlock situation. As a result of this research, this is now the subject of an as-yet unresolved *bug-report* with Sun[2] (*bug-report 19913*).

To try to alleviate this problem, a 25 millisecond total simulation delay was placed between each scheduled process time allocation, and a one second delay after each scheduled simulation time quantum (see Appendix C). It was hoped that during this system inactivity the garbage collector would be scheduled. The reason for the times chosen is that the garbage collector takes on average 20 milliseconds to run. This work-around was largely successful, however the garbage collector still ran at unscheduled times which has obviously affected some of the measured results in section 5.6. This was difficult to quantify, so the data from multiple runs of each test was collected, and the set with the least impact was chosen for analysing.

5.5.2 Inappropriate Behaviour of Master Processor Send Thread.

Because of the construction of the simulation outlined in section 5.4.3, the behaviour of thread τ_{snd} in thread group G_0 is not modelled correctly. In the actual parallel algorithm PV2 on the transputer, there are two tasks, T_{snd} and T_{rec} running together on the master processor. Because of the nature of the processor farm network communication (Chapter 4), task T_{snd} executes for almost the entire running time of the algorithm, preparing image blocks and transmitting them as required by the processor farm. Since T_{snd} can easily keep up with demand from the farm, it gets

[2] Sun Microsystems Incorporated - JavaSoft Division

processor time only when required, leaving the bulk of the processor time for task T_{rec}.

The simulation thread τ_{snd}, representing task T_{snd}, is scheduled with equal processor time to that of τ_{rec}, thus τ_{snd} finishes preparing all image blocks for processing by the farm early in the running time of the algorithm JVS1. It then enters a *Dead* state and is removed from the simulation. As a consequence, the thread τ_{rec} receives less processor time in the simulation than it should when τ_{snd} is alive, and more processor time in the simulation than it normally should after τ_{snd} dies.

The effect of this on the observed results is discussed in detail in section 5.6.

5.5.3 Network Timing Considerations

Since the simulation was run on a single processor system, and threads in Java all share the same process and memory space as in Figure 5.1, the simulation as described in section 5.4.3 does not take into account any communication delay between processors. As described in section 5.4.4, communication channels are simulated with the use of Java Vectors, and in order to build real-time communication network delays in the simulation, timing delays were placed at either end of these Vectors before access.

Delays were incorporated into the simulation to imitate data transmission over a 20Mbits/sec communication line between processors. The thread τ_{snd} places an 8×8 block of Java integers in the output queue, and the thread τ_{rec} removes an 8×8 block of Java double precision floating point numbers. Since a Java integer is four bytes, and a Java double is 8 bytes, this simulates transmitting blocks of 2048 bits and receiving blocks of 4096 bits.

Using the results from Table 4.8, the delays due to transmission of these amounts of data over a 20Mbits/sec line can be calculated. The figure of 40%, rounded from Table 4.8 has been used as a guide to the processor involvement percentage of the observed communication time. This percentage, while valid on the processor network it was measured on, will not be as high on the much faster processor on which the simulation was executed. However, the differences are extremely small in comparison to component processing time, and the discrepancy should have no appreciable effect on the simulation. These aspects are discussed in section 5.5.4.

Using the above-mentioned data, a processor involvement delay of 30 nanoseconds was introduced in the τ_{snd} and τ_{worker} threads. This simulated the processor involvement component of the communication delay with transmitting the 2048 bit blocks from τ_{snd} to the τ_{worker} threads in the processor farm, via the output communication channel. In addition, a 59 nanosecond delay was built into the τ_{worker} and τ_{rec} threads. This simulated the processor involvement component of the communication delay with transmitting the 4096 bit blocks to the τ_{rec} thread from the τ_{worker} threads of the processor farm, via the input communication channel.

The actual physical time the data is in transit over the communication line was also modelled. It was decided to model this from the master processor's perspective, and thus introduce two more delays in the τ_{worker} threads. A delay of 89 nanoseconds was introduced to account for the physical transmission of 2048 bits from the output channel, and a delay of 176 nanoseconds was introduced to account for the physical transmission of 4096 bits over the input channel. The term, input and output channel, here is used to refer to the Vectors simulating these channels, from the master processor's perspective. If the physical communication times were being modelled from the τ_{worker} thread's perspective, these communication times would be built into the τ_{snd} and τ_{rec} threads respectively, and not the τ_{worker} thread. These delays are shown in Figure 5.11.

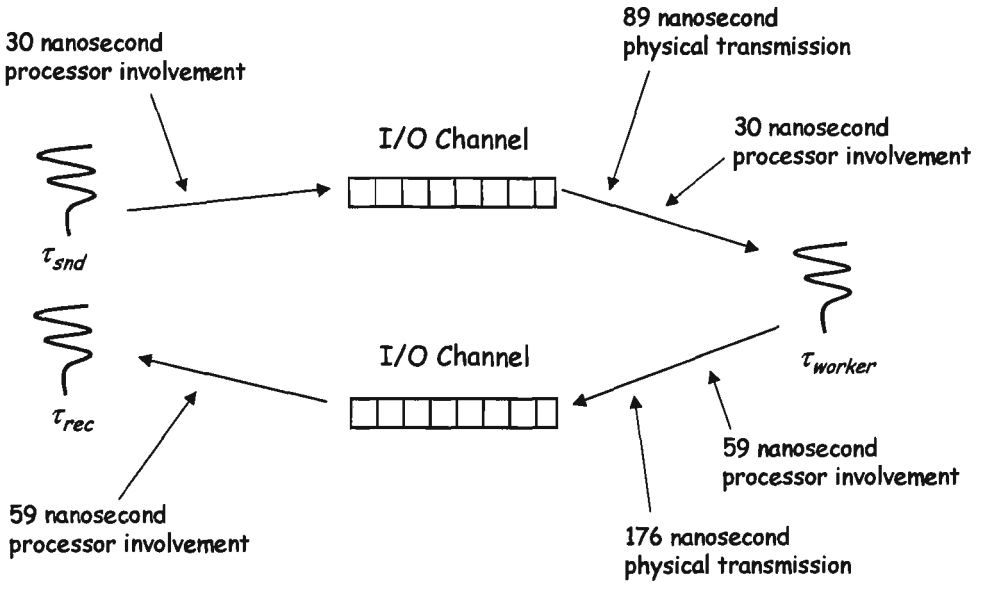


Figure 5.11 Delay times to simulate network communication delays

The allowance for communication times in Figure 5.11 is based on the investigations and calculations in Section 4.5. These calculations are based on adjacent communicating processors, with no allowance made for distance in the topology of the processor network. As discussed in Section 4.5, distance refers to the number of processors involved in the message routing, and a uniform distance of one is assumed. This is equivalent to using a star topology for the processor farm in relation to the master processor. This assumed configuration is depicted in Figure 5.12.

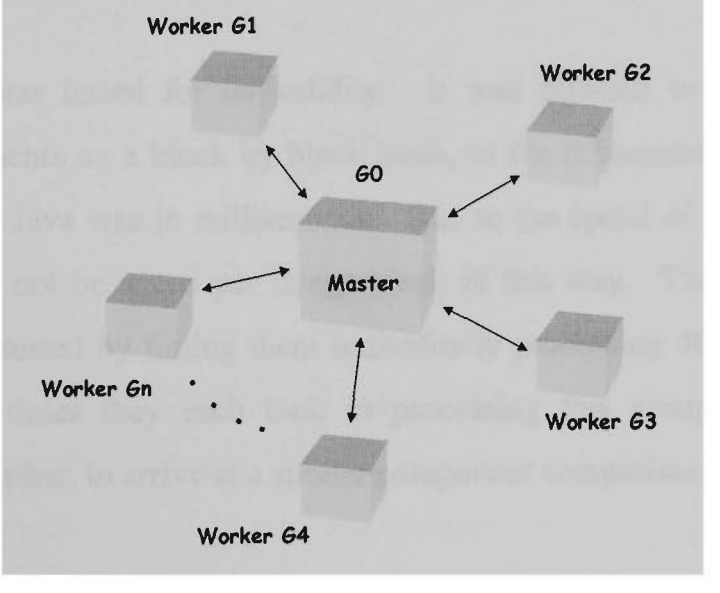


Figure 5.12 Assumed simulation network topology

It would be a relatively minor task to incorporate the simulated physical transmission delays into a distance function. The distance of a worker processor from the master processor could be specified when the appropriate thread group representing the worker processor is instantiated. As communication times appear to be a linear function of distance (Section 4.5), the basic delay factor could be multiplied by the distance. It would however be difficult to allow for the performance loss of intervening processors involved in the message routing. For this reason, no allowance in made for the slight degradation in performance of the simulated processor G_0 as the number of worker processors increases.

5.5.4 Processor Speed When Running Simulation

The Java simulation was performed on a personal computer with an Intel Pentium processor running at 160 MHz clock speed. It was hypothesized in Chapter 4 that the relative times of the JPEG algorithm components in Figure 4.2 should "scale down" when run on a newer generation, faster processor. That is, while the algorithm would run much faster, the relative times should remain approximately the same. Any difference in relative component times should come in the I/O components, where the speed of I/O technology has not increased at the same rate as processor speed.

This hypothesis was tested for its validity. It was difficult to test each of the individual components on a block by block basis, as the refinement of the processor timing function in Java was in milliseconds. Due to the speed of the processor, the components could not be tested per image block in this way. The JPEG algorithm components were tested by timing them individually processing 4096 image blocks. In this way, the times they each took in processing this group of blocks were compared to each other, to arrive at a similar component comparison as in Figure 4.3.

The figures obtained were then compared as overall percentages of image processing times to the same figures as measured on the transputer. These comparison figures are shown in Table 5.1. It can be seen that the DCT component remains as the

component that takes the largest amount of processing time. These results can also be seen represented in a comparative bar chart in Figure 5.13.

Table 5.1 Comparison of transputer and Pentium processor times

Processor	Get Block	Level Shift	DCT	Quantization	Huffman/ Block Store
Transputer	2.83	1.68	87.38	5.13	2.98
Pentium	6.27	2.29	69.32	10.7	11.42

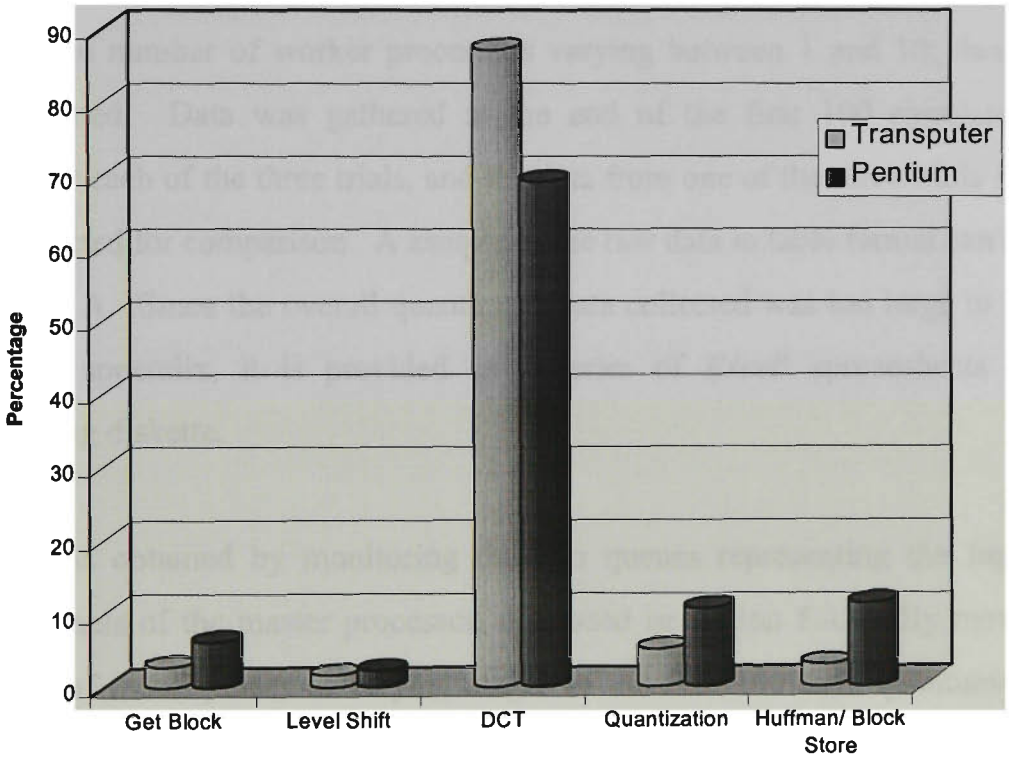


Figure 5.13 Comparative chart representation of Table 5.1

Although the figures represented in the chart in Figure 5.13 are similar to each other, there are some obvious differences that will have an effect on the simulation. These differences are discussed in Chapter 6. In order for the simulation to accurately reflect the processing within the transputer, timed delays were inserted in appropriate places in the simulation code. These delays then distorted the overall percentages of the JPEG components in the simulation, so that they were more accurately aligned with those representing the transputer as shown in Figure 5.13.

5.6 Simulation Results

Presented in this section are the results of running the Java simulation of the parallel processor farm algorithm constructed in the previous sections. The results of this section were obtained by the following procedure.

The Java simulation in Appendix C was run with the number of simulated worker processors ranging from one to ten, using a simulated time quantum Δt of 100 milliseconds allocated to each of the simulated processors in turn. For each of the runs, with the number of worker processors varying between 1 and 10, three trials were performed. Data was gathered at the end of the first 100 simulated time quantums for each of the three trials, and the data from one of the three trials for each run was selected for comparison. A sample of the raw data in table format can be seen in Appendix A. Since the overall quantity of data collected was too large to include fully in an appendix, it is provided as a series of *Excel*[3] spreadsheets on the accompanying diskette.

The data was obtained by monitoring the two queues representing the Input and Output channels of the master processor, discussed in section 5.4.4. By monitoring the lengths of these queues at varying stages of the first 100 time quantums, some indication of the saturation point introduced in Chapter 4 can be obtained. Only the first 100 time quantums were examined since it was during this period that thread τ_{snd} of thread group G_0 (representing the master processor) finished execution. That is, all the image blocks were gathered, level shifted and placed in the output queue. Once this occurred, thread τ_{snd} entered a Dead state, leaving thread group G_0 with only one thread, τ_{rec}. During the remaining processing time, this thread group, and the thread groups $G_1 \ldots G_n$ (representing the worker processors) with thread τ_{worker}, remained stable.

[3] Excel is a product of the Microsoft corporation

Unfortunately, this event during the simulation as discussed, section 5.5, does not reflect the actuality of the real multi-processor system algorithm PV2. In the real system, task T_{snd} runs occasionally only sending another block to the processor farm when one of the worker tasks becomes free. Most of the time, this task is idle allowing task T_{rec} most of the processor time to deal with incoming packets of work from the processor farm. In the simulation constructed in section 5.4.3, thread τ_{snd} representing task T_{snd} is scheduled regularly and thus finishes early. This both affects the simulation before and after this event in that before, thread τ_{rec} in the simulation is not getting as much time as its counter-part T_{rec} would, and after, τ_{rec} is getting more time than its counter-part T_{rec}. However, by examining the behaviour of the input and output queue lengths before and after this event, this simulation anomoly can be taken into account to arrive at a close approximation to the saturatiuon point.

Table 5.2 Overall averages of the first 100 time quantums

Number of Worker Processors	Increase in Master Output Queue Length	Difference in Master Input Queue Length	Blocks Transformed
1	39.00	0.02	1.95
2	37.34	0.04	3.6
3	35.45	0.06	5.48
4	33.46	0.08	7.46
5	31.55	0.1	9.36
6	29.34	0.12	11.56
7	27.94	0.14	12.95
8	26.58	1.11	14.45
9	23.91	3.14	16.96
10	22.44	5.22	18.42

Table 5.2 measures the overall averages of three quantities for the first 100 time quantums, with the number of simulated worker processors varying from 1 to 10. At each time quantum the increase in the output queue from the previous time quantum is measured and averaged. This gives an indication of the rate at which τ_{snd} is gathering image blocks and performing a level shift, thus also giving some indication if more worker processors are having an effect on this queue's rate increase. At each time quantum the difference in the input queue from the previous time quantum is measured and averaged. This gives an indication as to what point in adding more worker processors, that the master processor, (and in particular τ_{rec}) can no longer

cope (or keep up with) with the increase in worker packets from the processor farm. This measure in particular is used to help locate the saturation point. Since τ_{snd} finishes early, this measure is looked at before and after the event of τ_{snd}'s death. Table 5.2 also displays the average number of blocks processed by the processor farm for varying numbers of processors. This helps when comparing the previous two quantities in Table 5.2 to the increase in workload expected of τ_{rec}. The figures in Table 5.2 are presented as a comparative line chart in Figure 5.14.

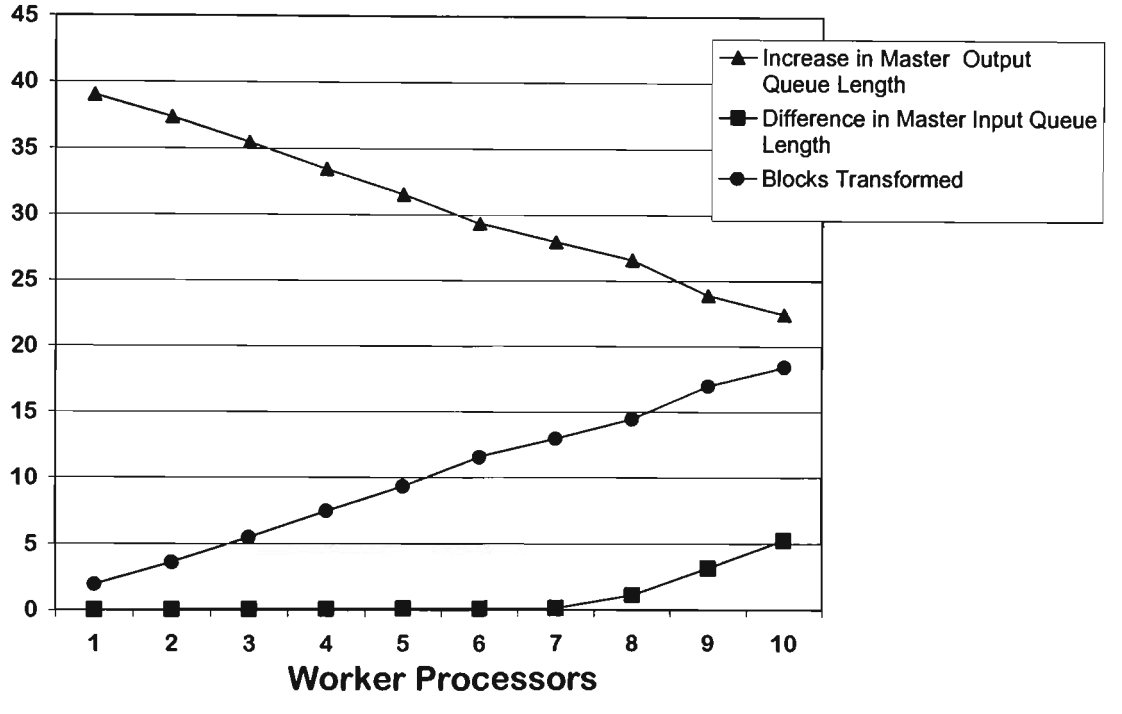


Figure 5.14 Comparative line chart representation of Table 5.2

From Figure 5.14 it can be seen that as the number of the worker processors in the farm increases, the average number of image blocks processed by the farm each time quantum, and hence available to τ_{rec} for processing, also increases, in an almost linear fashion. This is expected, as the more workers available in the farm, the more the farm should be able to produce. As a consequence of this, the increase per time quantum in the master processors output queue decreases as more worker processors are added, again, in a similar linear fashion. This is also expected, as more processors in the farm mean more image blocks can be taken from the output queue for processing each time quantum.

It should be pointed out that the averages graphed in Figure 5.14, are averages over all the 100 time quantums. Thus the averages for the increase in the output queue include all negative figures from when thread τ_{snd} dies. In the figures in Appendix A and the accompanying diskette, it is at the point when the individual numbers turn negative for this measure, that we can acertain that τ_{snd} has died.

The first indication of the saturation point can be seen in both the values for the difference in the input queue length in Table 5.2, and the corresponding line chart in Figure 5.14. There is an overall slight rise in the length of the input queue, which rises in an almost linear fashion until the size of the processor farm is seven worker processors. After more than seven workers are added to the farm, the rise sharply increases indicating that the simulation has reached a stage where τ_{rec} cannot process the worker packets faster, or at the same rate that the farm is producing them. This is the saturation point as described in Chapter 4. It should also be noted that until saturation point, it was expected that no rise at all should be seen in the master processor's input queue length. To understand this, some further analysis of the results is needed.

The data in Table 5.3 is similar to that shown in Table 5.2, except that it is shown only up to the point of the death of thread τ_{snd}. If the averages shown for the output and input queue lengths in Table 5.2 are examined separately before and after the death of τ_{snd}, some further insights can be gained. Both of these sets of averages are shown below. Table 5.3 shows the average length of the queues before this event, while Table 5.4 shows the averages after the event.

The figures corresponding to the average length of the input queue in Table 5.3 and Table 5.4 now make more sense, when viewed both before and after τ_{snd} death. While τ_{snd} is processing, it is filling up the output queue much faster than the processor farm can process these packets. When τ_{snd} dies, the input queue then begins to empty out at exactly the rate which the processor farm can then process

these work packets, which happens in an almost linear fashion according to the number of processors in the farm. Before the death of τ_{snd}, the output queue increase reduces at a rate comensurate with the number of worker processors retrieving packets from the queue. Both Table 5.3 and Table 5.4 can be viewed as comparative line charts which are shown in Figure 5.15 and Figure 5.16 respectively.

Table 5.3 Averages of first 100 time quantums up to τ_{snd} death

Number of Worker Processors	Increase in Master Output Queue Length	Difference in Master Input Queue Length
1	126.09	0.06
2	124.47	0.13
3	119.42	0.18
4	113.19	0.24
5	115.82	1.39
6	112.85	3.97
7	108.65	5.06
8	116.66	5.28
9	107.42	8.79
10	107.64	9.33

Table 5.4 Averages of first 100 time quantums after τ_{snd} death

Number of Worker Processors	Increase in Master Output Queue Length	Difference in Master Input Queue Length
1	-1.99	0
2	-3.66	0
3	-5.91	0
4	-7.98	0
5	-9.66	-0.54
6	-11.79	-1.78
7	-13.64	-2.39
8	-15.81	-0.85
9	-17.22	0.36
10	-19.52	3.19

By adding more workers to the processor farm, the number of packets processed per time quantum from the master processors output queue can be increased. By adding enough, the increase in the output queue length can be reduced to zero. However, this would take us far beyond the saturation point because the τ_{snd} thread runs for only a

relatively short time in the simulation, corresponding to task T_{snd} using relatively little processor time in the actual parallel version PV2. It is then up to thread τ_{rec} to process the output from the processor farm, and would be overwhelmed with the output from this number of processors.

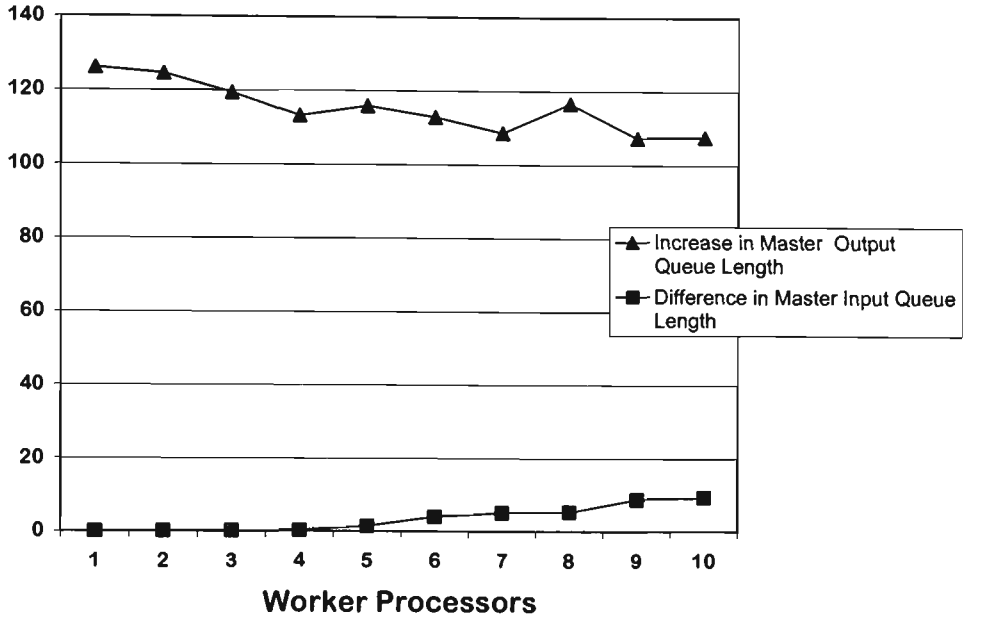


Figure 5.15 Comparative line chart representation of Table 5.3

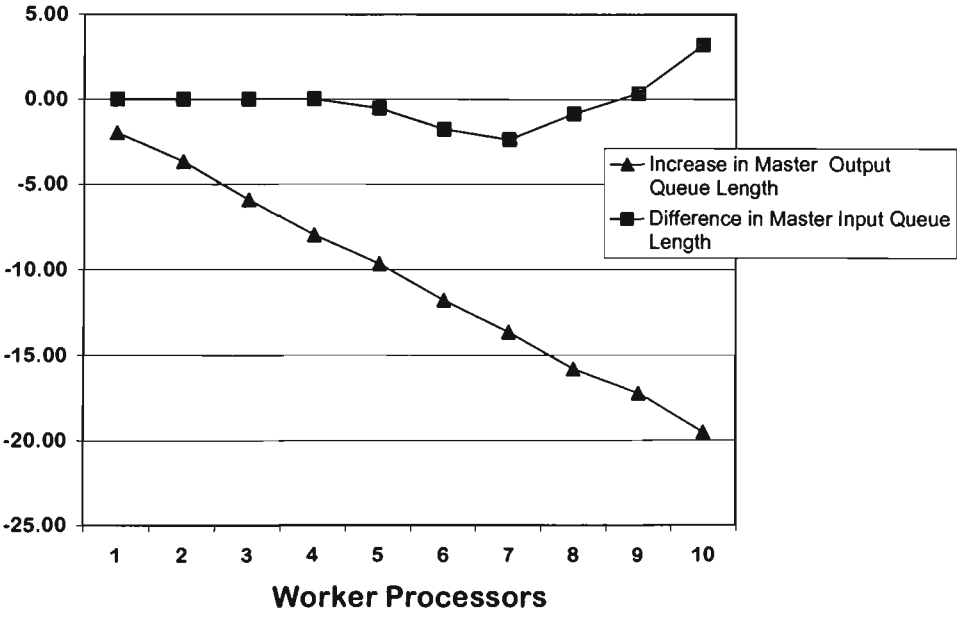


Figure 5.16 Comparative line chart representation of Table 5.4

It is the behaviour of thread τ_{rec} and its effect on the master threads input queue that more accurately shows the saturation point, (or its close proximity). If the line charts representing the difference in the input queue length in both Figure 5.15 and Figure 5.16 are placed together, they can be compared more easily by using a smaller scale as shown in Figure 5.17. If the trends in both of these plots are examined, it can be seen that both are relatively stable up to approximately four processors in the processor farm. The two plots diverge at five processors, but both have much sharper rises later. The plot representing the queue length before τ_{snd} finishes has a sharp rise after eight processors, while the plot representing queue length after τ_{snd} finishes, rises sharply after seven processors.

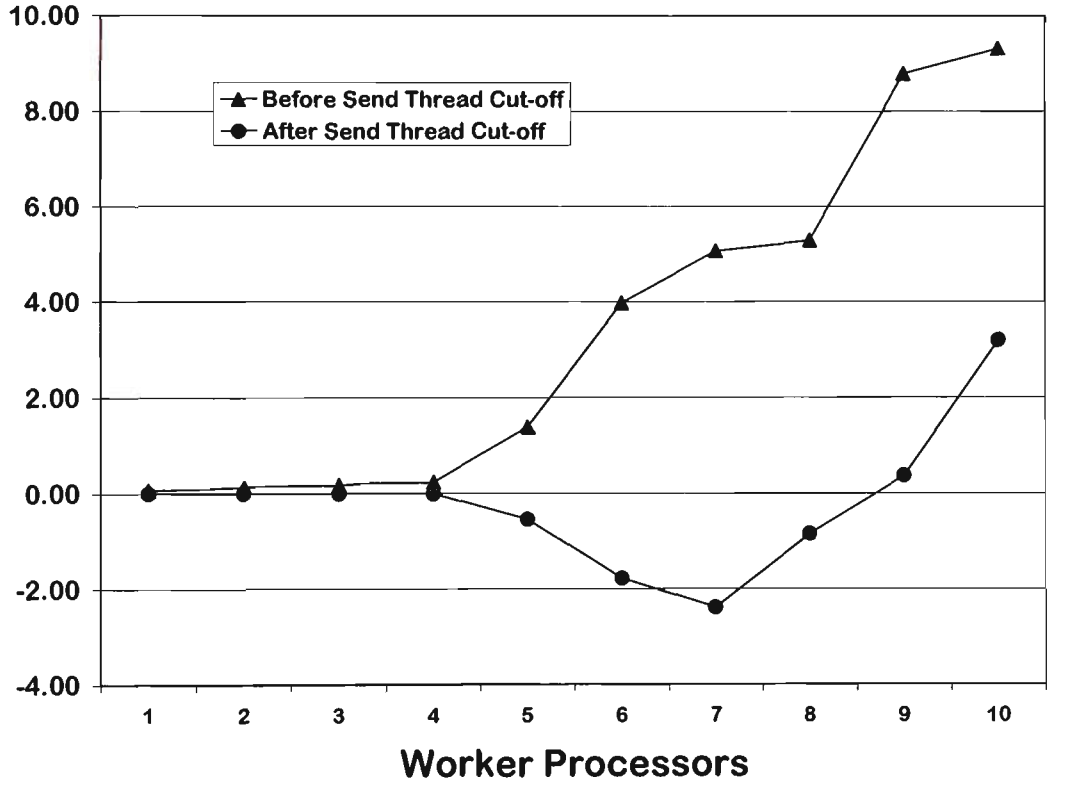


Figure 5.17 Comparison line charts of trends in input queue length

It should have been expected that, were the simulation behaving 100% correctly, both of these plots would have remained stable at zero, and both risen positively once saturation point was reached. Both do remain stable at zero, but diverge in opposite directions, well before saturation point. Both of these anomalies can be accounted for in the behaviour of the Java simulation discussed in section 5.5.

The rise in the plot before τ_{snd} cut-off is due to τ_{snd} taking too much processor time. The time quantum Δt (of 100 milliseconds) is shared equally between the two threads of simulated processor G_0 (see Appendix C). As can be seen from Figure 5.15, τ_{snd} is supplying the processor farm with far more work packets than it can handle, hence the rise in the input queue of well over 100 packets per time quantum. In the actual parallel processor farm algorithm, packets are only assembled and delivered as needed by the farm. Thus, the extra processor time thread τ_{snd} is using assembling these un-needed packets in the simulation, means that thread τ_{rec} has less time to process the incoming packets from the farm. While the overall saturation point for the parallel algorithm has not yet been reached, until τ_{snd} dies, thus allowing τ_{rec} more processing time, it cannot process fast enough, the incoming packets from the farm once the farm size becomes greater than four. The steeper rise after eight processors in the farm is an indication of the saturation point.

Again, after there are four processors in the farm, the plot representing the length of the input queue after thread τ_{snd} dies drops below zero, and keeps decreasing slightly until the farm has seven processors. This seems extradoniary, but it must be remembered that this plot represents the difference in the input queue length between time quantums, and not the actual queue length. This anomoly is actually caused by the thread τ_{snd} getting too much processor time when it was alive, as just discussed.

With up to four processors in the farm, τ_{rec} is getting enough processor time to empty out the input queue as fast as the farm can fill it, hence the plot remains stable at approximately zero. After four processors, as previously discussed, before thread τ_{snd} dies, thread τ_{rec} is not getting enough time to empty out the input queue, thus it accumulates entries until thread τ_{snd} dies. Once this happens, τ_{rec} is alone in thread group G_0, and then gets 100% of the simulated processor time. Until overall saturation point is reached, thread τ_{rec} can now empty out the input queue at the same rate or slightly quicker than the processor farm produces packets. The slight decrease

in the difference of the input queue length between time quantums, can be explained by τ_{rec} processing farm packets at a slightly quicker rate than the farm can produce them. The reason that the plot continues to decrease until there are seven workers in the farm is that for the same number of processors in the farm, τ_{snd} causes the input queue to increase at a faster rate. Thus there are more data packets which can be cleared from the input queue while τ_{rec} is still able to process these faster than they are received.

It should also be noted that these plots are constructed from data from the first 100 time quantums. If the data regarding the difference in the length of the master processors input queue was taken over all time quantums, then once τ_{rec} managed to clear accumulated farm packets from the input queue, there would be many entries in the difference column at or near zero. This can be seen in the tables in Appendix A and the provided files on the accompanying diskette. This would move the averages for the time after the death of τ_{snd} closer to zero. To test this hypothesis, data for seven worker processors was gathered from all the time quantums, and the average length of the input queue after τ_{snd} died for the first 100 time quantums was -1.85, yet taken over all time quantums was -0.5 (see Appendix A for this data). Thus the plots for the differences in the queue length after τ_{snd}'s death would move up.

From Figure 5.17 it can be seen that the turning point for this trend is after seven processors in the farm, and with nine processors in the farm, the differences in the length of the input queue is definitely increasing. At this point, is is fair to assume that saturation point has definitely been passed.

5.7 Conclusions

A number of conclusions were made in the previous chapter based on a limited set of results obtained from a parallel algorithm running on a transputer with three processors. In order to substantiate these conclusions they had to be tested on a larger

network of processors. In the absence of such a network, a simulation was devised using the Java language. The simulation devised took advantage of the multi-threaded nature of the language and exploited the unique feature of thread groups to simulate a multi-processor system.

Results were obtained from the simulation and presented in the previous section. These results support the idea of the saturation point of the number of useful processors that a parallel JPEG algorithm based on the processor farm paradigm can effectively use. In fact, with the data obtained from the simulation, the saturation point can be identified as being present at around seven or eight processors. This agrees with the preliminary calculation of the saturation point of approximately seven, from the original data in Chapter 4.

Data obtained from the simulation, also supports the relative overall processing times of the JPEG algorithm components, originally obtained from the older transputer processors. This also supports the idea that the processor farm paradigm would yield superior results to another parallel paradigm such as a pipeline.

CHAPTER 6

CONCLUSION

6.1 Summary

This research has demonstrated the existence of a practical limit to the number of processors that can effectively be used in the processor farm paradigm. JPEG image compression, which decomposes an image into blocks, is well suited to this type of processing. This is due to the independent component processing of the image blocks and the relative processing times of the components, and in particular the dominance of the DCT component, as shown in Figures 4.3 and 5.13. The research has further shown, by extrapolation of the experimental results in Chapter 4, and followed by the Java simulation in Chapter 5, that this limit is around 7 processors. This limit can be increased by re-arrangement of the tasks on the processors, as demonstrated in Section 4.7.3.3. However, in view of the diminishing return in terms of speedup time against the number of processors, which was shown in Figures 4.17 and 4.18, re-distribution of the tasks among the processors may not provide further effective speedup.

The saturation point in the number of useful processors is independent of the size of the digital image to be compressed. The saturation point is a measure of the master processor's ability to process the output of the worker processors, in the processor farm paradigm. If the saturation point is exceeded, then the master processor falls behind in processing due to overload in incoming result packets, Figure 4.12. To the processor farm, an image becomes just a stream of work packets to be processed, and result packets to be collated.

6.2 Critical Appraisal

This research has been useful in providing valuable data to formulate the hypothesis of the existence of the saturation point, and then to measure it. The extrapolations derived in Chapter 4 were obtained with data gathered from using at most three processors. Using more processors would have given extra confidence but it is believed that the results are sufficiently accurate. As the findings are dependent on accurate timings, it was necessary to take account of different processor speeds in the farm. However, once used for confidence testing, it seemed reasonable to normalize these speeds to simplify calculations, and this was done using an average of the processor speeds in the farm.

Some of the other obstacles, that made it difficult to measure times, were the performance of the *filter* and *afterver* tasks on the root and host processors respectively, Figure 3.18. The filter task took time away from the master processor in the processor farm and this was difficult to measure. The behaviour of the *afserver* task on the host processor was unknown, but was responsible for the I/O to the host I/O devices. To try to reduce any erratic behaviour of this task, multiple time trials were taken so an average could be used.

When calculating speedup with the introduction of more processors in 4.7.3.2, the performance gain with the placement of a copy of the worker task on the master processor was difficult to account for. However, it is believed that the method used in this research gave a reasonable estimate. This consisted of estimating the time it would take, to process the expected number of result packets from the farm, then calculating the idle time of the master processor. This was then used as an indication of the extra blocks that could be processed by the worker task on the master processor.

The use of the Java language for the simulation of the processor farm in Chapter 5 may also be questioned. Java is not as efficient as other languages such as C, for use when real-time behaviour is expected. In hindsight, the use of C would have avoided the many problems encountered in Section 5.5, particularly those involving the Java garbage collector. However, this was a simulation, and was not meant to provide real-

time behaviour. It did achieve its objectives in allowing the modeling of the worker processors as threads, so the saturation point could be observed. The in-depth experience gained in using Java for the simulation was invaluable, and this knowledge was subsequently used during the course of my own teaching.

To model the physical message passing of the transputer, Java vectors were used. When using these vectors, some of the initial data values obtained from the simulation of processor scheduling time quantums seemed dubious. When it was realized that the garbage collector was impacting on these, many trials were performed and the values observed. It was clear when the garbage collector had had an impact, as very few image blocks were processed by a simulated processor during that time quantum. Thus over many trials, those displaying this effect were discarded, yielding a set of reliable results. From the trials selected for further processing, reasonable results were obtained.

Another thing to note is that the Pentium processor used to conduct the simulation was faster than the transputer processors. There was concern that the times taken to process the components of the JPEG algorithm would not be in the same ratios as with the transputer. After testing, Figure 5.13 indicates that, while quicker on the Pentium, the DCT component was still the most time consuming. However, the Pentium processor took more time with the other components. This suggests that the Java environment, with its use of object oriented techniques for creating and obtaining these objects from Vectors played a part in this. On a per block basis these components could not be measured because of the coarseness of the timing function. The times were obtained by processing all 4096 blocks for each component and then averaging these. Again, the garbage collector will have played a role in distorting some of these times. The DCT processing was faster, possibly because of the far greater efficiency of the Pentium floating point processor, and with these shorter times, not being interrupted as much by the garbage collector.

If a Transputer system with more processors were available, some further testing should be performed to confirm the accuracy of the simulation. Algorithm PV2 would then be tested with a larger processor farm by progressively adding more processors, enabling a number of the predictions to be confirmed for accuracy.

Algorithm PV1 need not be tested further as it was constructed to test the algorithm with one extra processor. Predicted values to be checked would be the overall processing times of the master processor vs. the average worker processor times, and the progressive time saving gained by the introduction of more processors. This would enable the predictions in Figures 4.14 and 4.18 to be verified. Verification of these values would also be valuable in determining the accuracy of the predicted result for the saturation point, and of the simulation values in Section 5.6.

I believe that the methods used, outlined above have helped overcome any of the problems encountered with this research. The research has been successful as it has helped to quantify limitations of a particular parallel processing paradigm used for JPEG standard image compression. The insights and experience gained have been valuable, and the process of creating the simulation has initiated an interest in distributed processing using the internet. Some answers have been found, but also more questions posed. Some of these questions are outlined in the areas for further research.

6.3 Further Research

Research by Wagner et.al. [85], indicates that if the problem to be parallelised fits the processor farm paradigm, then there is probably no need to seek more elaborate parallelization schemes. The processor farm scheme provides good performance up to saturation point. With up to seven processors, estimates based on figures from Chapter 4 indicates JPEG compression time on the transputer for small pictures of 512 × 512 pixels, using the processor farm paradigm is in the order of 11 seconds. This time will be reduced when faster processors are used.

However, in some circumstances, huge pictures are generated and even with seven processors, real-time processing of these images using this paradigm would not be possible. Some sources of these large images are space probes, the *Hubble* telescope and geographical data collected from orbiting satellites. In the last example, multiple type information can be collected for each picture. Landsat satellites collect image

data on 4 spectral bands and some images can be up to 300 megabytes in size. Some digital images made available at the beginning of this research from Kodak[1] were over 20 megabytes in size.

For JPEG compression, if the processor farm paradigm is near optimal [85], and this research has demonstrated the existence of a saturation point, the next question, is how do we make better use of more processors? This research has shown that, extension of the saturation point only brings minimal gains for speedup. One area for possible future research would be the investigation of the possibility of *multiple processor farms* to increase speedup. The root transputer of a transputer system is often used as a gateway to more elaborate configuration schemes [59] [43]. Thus, rather than involve the root transputer in a processor farm paradigm, it could be used as a gateway to multiple processor farms, as depicted in Figure 6.1.

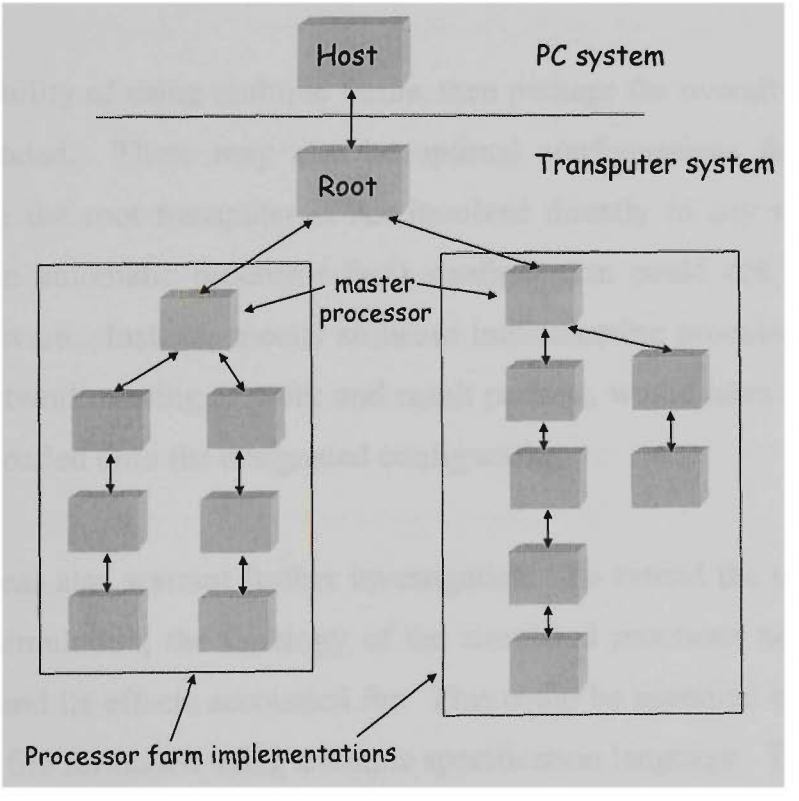


Figure 6.1 Possible use of multiple processor farms

As each transputer has four external links, then if one of these is allocated to the host processor, this leaves three links to provide gateways to other processor farms. It

[1] Kodak Australasia Pty. Ltd

would be expected that under such a configuration, the links from the root processor would become bottlenecks. The research in [85] investigates the best performance in the processor farm with respect to configuration. However, with multiple processor farms, there is then the question of optimal configuration for performance of the parallel system, with a limited number of processors allocated to a number of processor farms.

The research in Section 4.7.3.3 shows more dramatic speedup when adding the first two processors to the farm than there is for the rest. Thus, another interesting question is, would it be better to have three processor farms with 3 or 4 processors, or two farms configured up to saturation point? The root transputer could be used solely to allocate portions of the original image to the processor farms and then to collate the results from the farms. The communication overhead on the root transputer would then have to be taken into account when allocating image portions.

With the possibility of using multiple farms, then perhaps the overall saturation point could be extended. There may also be optimal configurations for different size images. Since the root transputer is not involved directly in any of the processor farms, then the automatic processor farm configuration could not be used by the transputer software. Instead, special software implementing processor farms, which handled the network routing of work and result packets, would have to be developed in order to be loaded onto the designated configuration.

Some other areas also warrant further investigation. To extend the usefulness of the Java parallel simulation, the topology of the simulated processor network could be made known, and its effects accounted for. This could be specified in advance in the form of a data file formatted using a simple specification language. This file could be read by the controlling thread of the simulation and used when creating the thread groups representing the processors. Each of the thread groups could be passed distance and load factor figures, representing the distance from the master processor, and a load factor indicating the number of other processors that will use this processor in their message routing.

The simulation algorithm could be generalized to allow easy simulation of other parallel computations using a processor farm paradigm. The worker thread already resides in its own class, and thus would only need to be modified to reflect its new task, similarly for the send and receive threads. However, the master or controlling thread does contain some application specific code. This could be removed and placed in an initialization thread, which again could be modified on a per application basis. This would leave the controlling thread with the sole responsibility of the simulation scheduling. This algorithm could further be modified to enable the simulation parallel paradigms other than the processor farm. Instead of creating n copies of the worker thread, the controlling thread could be used to create n threads of possibly different tasks. However, the communication aspects would then need to be re-addressed.

One area that would require intensive study is the possibility of using Java to actually distribute processor load to various processors over the Internet. That is, to investigate the possibility of using the Internet as a large, distributed parallel processor. Java was developed for extensive use on the Internet, and in fact, has a rich set of classes for distributed processor communication. This language is ideally suited for this particularly due to its platform independence. One of the major impediments to this idea is the current limitation on bandwidth. However, assuming this situation will improve in future, it remains an interesting proposition. The entire Internet could possibly be treated as one large distributed multi-processor. Of course, security issues would play a major role in any research into this area.

BIBLIOGRAPHY

References

[1] Abramson, N., *Information Theory and Coding*, McGraw-Hill, New York, 1963.

[2] Ahmed, N., Natarajan, T., and Rao, K.R., "Discrete Cosine Transform", *IEEE Transactions on Computers*, C-23:90-3, Jan 1974.

[3] Aizawa, K., Harashima, H., and Miyakawa, H., "Adaptive Discrete Cosine Transform Coding with Vector Quantization for Color Images", *Proceedings of ICASSP 86*, Tokyo, pp 985-988.

[4] Al-Dabass, D., "Distributed Parallel Processing using the Synchronous Farms Technique", http://www.doc.ntu.ac.uk/PS/Papers/p1.html , 20/11/97.

[5] Altmann, J., "Surfing the Wavelets", http://www.monash.edu.au/cmcm/wavelet/wavelet.htm, 23/1/98.

[6] Andrews, H.C., and Pratt, W.K., "Fourier transform Coding of Images", *Proceedings Hawaii International Conference on System Science*, pp 677-679, Jan 1968.

[7] Barnsley, M., *Fractals Everywhere*, Academic Press, 1988.

[8] Barnsley, M., Ervin, V., Hardin, D., and Lancaster, J., "Solution of an Inverse Problem for Fractals and other Sets", *Proceedings of National Academy of Science 83*, Apr 1986, pp 1975-1977.

[9] Barnsley, M., Jacquin, A., Malassenet, F., Reuter, L., and Sloan, A., "Harnessing Chaos for Image Synthesis", *Computer Graphics*, Vol. 22 No. 4, pp 131-140, 1988.

[10] Bauer, B.E., *Practical Parallel Programming*, Academic Press, 1992.

[11] Brislawn, C., "The FBI Fingerprint Image Compression Standard", http://www.c3.lanl.gov/~brislawn/FBI/FBI.html, 21/2/98.

[12] Campione, M., and Walrath, K., *The Java Tutorial, Object-Oriented Programming for the Internet*, The Java Series, Addison-Wesley, 1996.

[13] Carriero, N., and Gelernter, D., *How to Write Parallel Programs: A First Course*, MIT Press, 1990.

[14] Chen, W.H., Smith, C.H., and Fralick, S.C., "A Fast Computational Algorithm for the Discrete Cosine Transform", *IEEE Transactions on Communications*, Vol. 25, Sep 1977.

[15] Clarke, R., *Transform Coding of Images*, Academic Press, 1985.

[16] Cochran, W., Hart, J., and Flynn, P., "Fractal Volume Compression", *Research Paper, School of Electrical Engineering and Computer Science*, Washington State University, October 18, 1995.

[17] Cochran, W., Hart, J., and Flynn, P., "Principal Component Classification for Fractal Volume Compression", *Research Paper, School of Electrical Engineering and Computer Science*, Washington State University, October 18, 1995.

[18] Cooley, J.W., and Tuckey, J.W., "An algorithm for the Machine Calculation of Complex Fourier Series", *Mathematics of Computers*, Vol. 19, pp 297-301, 1965.

[19] Culter, C.C., *Differential Quantization of Communication Systems*, U.S. patent 2605361, July 29, 1952.

[20] Darbyshire, P., "Using Java to Simulate Multi-processor Systems", *Department of Information Systems Working paper series*, Victoria University of Technology (in print), http://busfa.vut.edu.au/papers/pgd02.htm, 8/8/97.

[21] Darbyshire, P., Pleasants, A., and Wang, J., "Parallel Implementation of the JPEG Still Picture Compression Algorithm", *Proceedings of Australian Pattern Recognition Society Student Conference*, 1996.

[22] Darbyshire P., and Wang J., "Implementation of a Parallel Image Compression Algorithm Using Java", *Proceedings DICTA '97*, Massey University, Auckland NZ, 1997.

[23] Daubechies, I., "The Wavelet Transform, Time- Frequency Localization and Signal Analysis", *IEEE Transactions on Information Theory*, Vol. 36, No. 5 pp 961-1005, 1990.

[24] Dept. Engineering, Purdue University, *Annual Research Summary: Laboratory Facilities and Research Centers*, Part II, July 1995 – June 1996.

[25] Dougherty, E.R., et.al., *Digital Image Processing Methods*, Marcel Dekker Inc, 1994.

[26] Flynn, M.J., "Some Computer Organizations and Their Effectiveness", *IEEE Transactions on Computers*, Vol 21, No. 9 Sept 1972 pp 948-960.

[27] Fumitaka, O., et. al., "Bi-Level Image Coding with Melcode, A Comparasion of Block Type Code and Arithmetic Type Code", *Proceedings Globecom '89*, 1989.

[28] Gonzalez, R., and Wintz, P., *Digital Image Processing*, second edition, Addison Wesley, 1987.

[29] Gonzalez, R., and Woods, R., *Digital Image Processing*, Addison-Wesley, 1993.

[30] Gosling, J., and McGilton, H., "The JAVA Language Environment, A White Paper", May 1996, http://java.sun.com/docs/white/langenv/ , 5/7/97.

[31] Graps, A., "An Introduction to Wavelets", http://www.amara.com/IEEEwave/IEEEwavelet.html, 21/2/1998.

[32] Gray, R., "Vector Quantization", *IEEE Acoustics Speech and Signal Processing Magazine*, Apr 1984, pp4-29.

[33] Gray, R.M., *Entropy and Information Theory*, Springer-Verlag, 1990.

[34] Habibi, A., "Comparison of m^{th} order DPCM Encoder with Linear Transformation and Block Quantization Techniques", *IEEE Transactions on Communication Technology*, Vol 19, No. 6, pp 948-956, 1971.

[35] Haque, M.A., "A Two-Dimensional Fast Cosine Transform", *IEEE Transactions on Acoustics Speech and Signal Processing*, Vol. 33, No. 6, Dec 1985.

[36] Hart, J., "Fractal Image Compression and the Inverse Problem of Recurrent Iterated Function Systems", *Research Paper, School of Electrical Engineering and Computer Science*, Washington State University, Feb 9, 1995.

[37] Hartley, R.V.L., "Transmission of Information", *Bell System Technical Journal*, 7:535-563, 1928.

[38] Hoare, C.A.R., "Communicating Sequential Processes", *Communications of the ACM*, Vol 21 No. 8, Aug 1978 pp 666-677.

[39] Hord, R.M., *Parallel Supercomputing in SIMD Architectures*, CRC Press, 1990.

[40] Hord, R.M., *Parallel Supercomputing in MIMD Architectures*, CRC Prsss, 1993.

[41] Hudson, G., Yasuda, H., and Sebestyen, I., "The International Standardization of a Still Picture Compression Technique", *Proceedings Globecom '88*, 1988.

[42] Huffman, D.A., "A Method for the Construction of Minimal Redundancy Codes", *Proceedings of the IRE*, Vol. 40, No. 10, pp1098-1101, 1952.

[43] Hull, M.E.C., Crookes, D., and Sweeney, P.J., *Parallel Processing, The Transputer and its Applications*, Addison Wesley, 1994.

[44] Inmos, *Communicating Process Architecture*, Prentice Hall, 1988.

[45] Inmos, *IMS B008 User Guide and Reference Manual*, INMOS Limited, 1988.

[46] Inmos, *Transputer Technical Notes*, Prentice Hall, 1989.

[47] Jacquin, A.E., "Image Coding based on a Fractal Theory of Iterated Contractive Image Transformations", *IEEE Transactions on Image Processing*, Vol 1, No. 1, pp 18-30, Jan 1992.

[48] Jain, A., "Image Data Compression : A Review", *Proceedings of the IEEE*, Vol. 69, No. 3, Mar 1981, pp349-389.

[49] Jain, A. K., *Fundamentals of Digital Image Processing*, Prentice Hall, 1987.

[50] Jaisimha, M.Y., et. al., "Data Compression Techniques for Maps", *Proceedings Southeastcon 89*, 1989.

[51] Johnson, E., "Independent Performance Modeling of Parallel Architectures and Algorithms", *Proceedings ICCI'93*, 1993.

[52] Joint Technical Committee ISO/IEC JTC-1, "Information technology - Digital compression and coding of continuous-tone still images: Requirements and guidelines", Reference number ISO/IEC 10918-1:1994.

[53] Joint Technical Committee ISO/IEC JTC-1/SC 29/WG 10 N 188, "Information technology - Digital compression and coding of continuous-tone still images: Compliance testing", Reference number ISO/IEC 10918-2:1994.

[54] Karhunen, K., "Ueber lineare methoden in der Wahrscheinlichkeitsrechnung", *Ann. Academy Science Fennicae*, Serial Maths and Physics, Vol 37, 1947.

[55] Kunt, M., "Second-Generation Image-Coding Techniques", *Proceedings of the IEEE*, Vol. 73, No. 4, Apr 1985, pp549-574.

[56] Kunt, M., and Leonardi, R., "Recent Results in High-Compression Image Coding", *IEEE Transactions on Circuits and Systems*, Vol. 34, No. 11, Nov 87, pp 1306-1336.

[57] Lee, C., and Shin, K., "Optimal Task Assignment in Homogeneous Networks", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 8. No. 2, 1997.

[58] Leger, A., Mitchell, J., and Yamazaki, Y., "Still Picture Compression Algorithms Evaluated for International Standardization", *Proceedings Globecom '88*, 1988.

[59] Lerman, G., and Rudolph, L., *Parallel Evolution of Parallel Processors*, Plenum press, New York, 1993.

[60] Lewis, T.G., *Foundations of Parallel Programming : A Machine Independent Approach*, IEEE Computer Society Press, 1993.

[61] Linde, Y., Buzo, A., and Gray, R.M., "An algorithm for Vector Quantizer Design", *IEEE Transactions on Communications*, Vol. 28, No. 1, Jan 1980, pp 84-95.

[62] Lloyd, S.P., "Least Squares Quantization in PCM", Bell Laboratories 1957, *IEEE Transactions on Information Theory*, Vol. 28, pp 129-137, Mar 1982.

[63] Loeve, M., *Probability Theory*, Van Nostrand, Princeton, NJ, 1960.

[64] Max, J., "Quantizing for Minimum Distortion", *IRE Transactions on Information Theory*, Vol. 6, pp 7-12, Mar 1960.

[65] MHPCC (Maui High Performance Computing Center), "Introduction to Parallel Programming", http://www.ciril.fr/CIRIL/Logice...allele.docs/general/, 30/11/97.

[66] Morse, B.S., "Introduction to Digital Signal and Image Processing", http://iul.cs.byu.edu/450/, 10 Dec 1996.

[67] Nelson, M.R., "Arithmetic Coding and Statistical Modeling", *Dr. Dobbs Journal*, Feb 1991.

[68] Parallel Systems Research, *Parallel C User Guide*, Knowledge Dynamics Corporation, Ver 2.2.2, 1991.

[69] Pennebaker, W.B., and Mitchell, J.L., *JPEG Still Image Data Compression Standard*, Van Nostrand Reinhold, 1993.

[70] Rao, K.R., and Yip, P., *Discrete Cosine Transform: algorithms, advantages, and applications*, Academic Press, 1990.

[71] Rosenfeld, A., and Kak, A.C., *Digital Picture Processing*, 2nd ed., Vol 1, Academic Press, 1982.

[72] Russ, J.C., *The Image Processing Handbook*, 2nd ed., CRC Press, 1995.

[73] Santini, S., "Why Wavelets?", http://www.cse.ucsd.edu/users/ssantini/cse228f/wavelet/sample.html, 23 Jan 1997.

[74] Schouten, B.A.M., "Fractal Image Compression", CWI National Research Institute for Mathematics and Computer Science, Netherlands, http://www.cwi/nl/~bens/, 21/2/98.

[75] Schreiber, W.F., "The Measurement of Third Order probability Distributions of Television Signals", *IRE Transactions on Information Theory*, Vol. IT-2, pp 94-105, Sept 1956.

[76] Schultz, R., "Comparison of Compression Algorithms for Image Data", Computer Graphics Institute, Rostock University, Denmark, http://www.icg.informatik.uni-rostock.de/Projekte/MoVi/rschultz/kompve/index.html, 4/2/98.

[77] Sevior, Charles, F., "An Overview of Video Compression Techniques", *General Television Corporation*, Pty. Ltd.

[78] Shannon, C.E., "The Mathematical Theory of Communication", *Bell Systems Technical Journal*, Vol. 27, pp 379 and 623, 1948.

[79] Shannon, C.E., and Weaver, W., *The Mathematical Theory of Communication*, Illini Books, 1963.

[80] Sloan, A.D., "The Fractal Image Format and JPEG", *Proceedings Electronic Imaging International Conference*, 1991, pp 460-465.

[81] Storer, J.A., *Data Compression, methods and theory*, Computer Science Press, 1988.

[82] Takahashi, K., Sato, Y., and Ishii, N., "A Level-Plane Coding Scheme for Progressive Transmission of Gray-Scale Images", *Proceedings Globecom '89*, 1989.

[83] Thede, L., and Kwatra, S., "A Hybrid Data Compression Scheme Using Quaternary Decomposition and Selective Multistage Vector Quantization", *Proceedings Globecom '89*, 1989.

[84] Thiebaut, D., "Parallel Programming in C for the Transputer", http://cs.smith.edu/~thiebaut/transputer/descript.html , 15/2/98.

[85] Wagner, A., Sreekantaswamy, H., and Chanson, S., "Performance Models for the Processor Farm Paradigm", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 8, No. 5, 1997.

[86] Wallace, G., Vivian, R., and Poulsen, H., "Subjective Testing Results for Still Picture Compression Algorithms for International Standardization", *Proceedings Globecom '88*, 1988.

[87] Wallace, G., "The ISO/CCITT Standard for Continuous-Tone Image Compression", *Proceedings Globecom '89*, 1989.

[88] Wallace, G., "The JPEG Still Picture Compression Standard", *Communications of the ACM*, Vol.34, No. 4, April 1991.

[89] Wang, J.D., and Pleasants, A., "Arithmetic Coding for Finite State Sources", *Proceedings DICTA '91*, Melbourne, 1991, pp 538-545.

[90] Watkinson, J., *Compression in Video and Audio*, Butterworth-Heinemann Ltd., 1995.

[91] Whitney, R., "CS 596 Client-Server Programming Threads", San Diego State University, http://www.eli.sdsu.edu/courses/spring97/cs596/notes/threads/threads.html 5/7/97, http://www.eli.sdsu.edu/courses/spring97/cs596/notes/threads2/threads2.html 5/7/97.

[92] Zomaya, A.Y., *Parallel Computing: Paradigms and Applications*, International Thompson Computer press, 1996.

APPENDIX A

IMAGES AND TABLES

A.1 Introduction

The following pages in this appendix contain the images used for testing the algorithms developed in Chapter 4 and Chapter 5, and some of the collected data from experimental runs. Many of the tables appearing in Chapter 4 are summarized versions of the original data included here. Also included is some of the raw data collected from simulation runs discussed in Chapter 5. The inclusion of all this data in printed format is both unnecessary and costly in terms of space, so the raw data files are provided as Excel files on the accompanying diskette. Some of the JPEG procedure diagrams are also included in Section A.4.

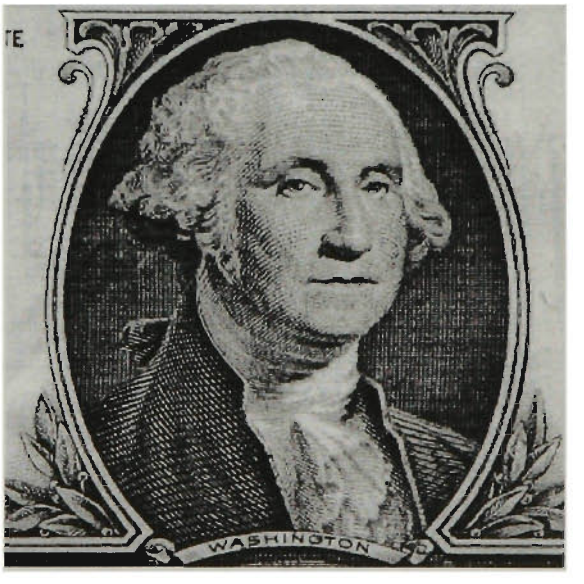
A.2 Images

The following images are those used for testing in Chapters 4 and 5.

(a) Adam

(b) Plates

(c) Preps



(d) George

All the above images are displayed at one half size, and are *512* × *512* pixels, thus each has 4096, *8* × *8* image blocks. The sample size of each image is 8 bit greyscale, with image (d) *George*, having the most distinct number of grey shades, 247. Image (a) has 236 gray shades, (b) has 235 and (c) has 244. Images (a) – (c) are 24 bit true colour scans reduced to 8 bit grey shades, while image (d) is a 256 grey shade scan of

a portion of the U.S. one dollar bill. Because of the nature of the engraving, there are more edges and contours in image (d) than the others.

A.3 Chapter 4 Tables

The following table contains timing data for five time trials of Algorithm SV1 on Processor P_0 in Chapter 4. For each time trial, the total clock ticks over all image blocks and the average clock ticks per image block is shown for each of the 6 components from SV1.

Table A.1 Timing data for algorithm SV1 components

Algorithm Component		Trial 1	Trial 2	Trial 3	Trial 4	Trial 5
Get Block	Total	41666	41725	41642	41598	41646
	Average/Block	10.17236	10.18677	10.1665	10.15576	10.16748
Level Shift	Total	24789	24772	24805	24801	24803
	Average/Block	6.052	6.04785	6.05591	6.05493	6.05542
FDCT	Total	1286877	1286911	1286885	1286880	1286874
	Average/Block	314.17896	314.18726	314.18091	314.17969	314.17822
Quantization	Total	75610	75565	75547	75650	75567
	Average/Block	18.45947	18.44849	18.44409	18.46924	18.44897
Huffman Coding	Total	26223	26127	26135	26208	26160
	Average/Block	6.4021	6.37866	6.38062	6.39844	6.38672
Block Storage	Total	17567	17705	17695	17545	17664
	Average/Block	4.28882	4.32251	4.32007	4.28345	4.3125

Table A.2 below contains timing data for the four non I/O algorithm components, T_{lshift}, T_{dct}, T_{quant}, and T_{code}, on processors P_1 and P_2. Timing data for these tasks on processor P_0 has already been obtained and shown in Table A.1. The figures in Table A.2 are compared to the equivalent figures of Table A.1 within Chapter 4.

Table A.2 Timing of SV1 non I/O components on processors P1 and P2

FDCT	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average	Average/Block
Processor P1	1094370	1094352	1094326	1094344	1094371	1094352.6	267.1759
Processor P2	873797	873790	873807	873766	873784	873788.8	213.3273
Level Shift	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average	Average/Block
Processor P1	21573	21511	21526	21569	21523	21540.4	5.258887
Processor P2	17235	17228	17221	17211	17238	17226.6	4.205713
Quantization	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average	Average/Block
Processor P1	65654	65649	65686	65646	65680	65663	16.03101
Processor P2	52429	52437	52461	52452	52460	52447.8	12.80464
Huffman Coding	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average	Average/Block
Processor P1	21102	21104	21065	20852	21062	21037	5.135986
Processor P2	16773	16601	16675	16690	16706	16689	4.074463

The next four tables contain processor communication timing data. The times are in clock ticks (cts) and represent the total time it takes to send and receive 4096 messages, formatted as 8 × 8 arrays (or blocks) between processor P_0 and processors P_1 and P_2. The size of each element in the block is varied between the four tables.

Table A.3 to Table A.6 show the timing for transmission and receipt of 4096 blocks, where each element in the block consists of the stated number of bytes. In Table A.6, there is a difference in the size per element of the transmitted block and the received block. All times are measured in cts units.

Table A.3 Transmission & receipt times of 4096 blocks (1 byte/element)

Trial	P1	P2
1	5192	10374
2	5193	10374
3	5192	10374
4	5192	10374
5	5192	10373
Average	5192.2	10373.8

Table A.4 Transmission & receipt times of 4096 blocks (4 bytes/element)

Trial		P1	P2
1		19021	38494
2		19021	38495
3		19020	38495
4		19020	38494
5		19019	38494
Average		19020.2	38494.4

Table A.5 Transmission & receipt times of 4096 blocks (8 bytes/element)

Trial		P1	P2
1		37442	75962
2		37441	75961
3		37441	75962
4		37441	75961
5		37441	75961
Average		37441.2	75961.4

Table A.6 Transmission & receipt times of 4096 blocks (4/8 bytes/element)

Trial		P1	P2
1		28236	56923
2		28237	56923
3		28234	56922
4		28237	56923
5		28236	56923
Average		28236	56922.8

A.4 JPEG Procedure Diagrams

Figure A.1 is the JPEG procedure diagram for the generation of the table of Huffman code sizes. These code sizes are then used to generate the table of Huffman codes. The procedure diagram for the generation of the table of Huffman codes is shown in Figure A.2. These procedure diagrams are reproduced from Annex C [52].

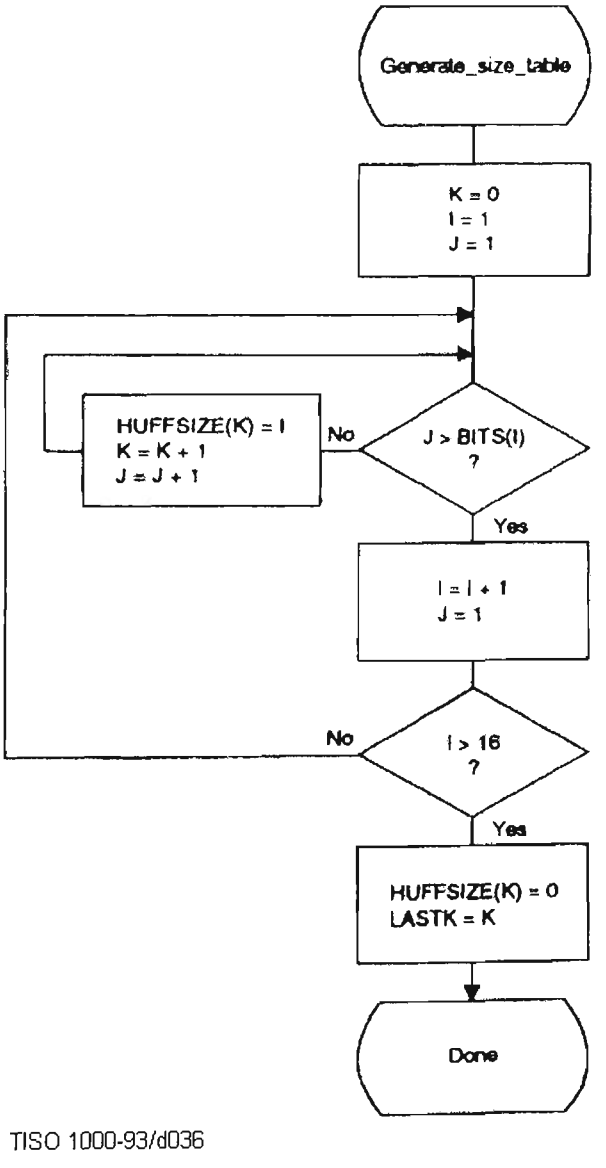


Figure A.1 Generation of table of Huffman code sizes

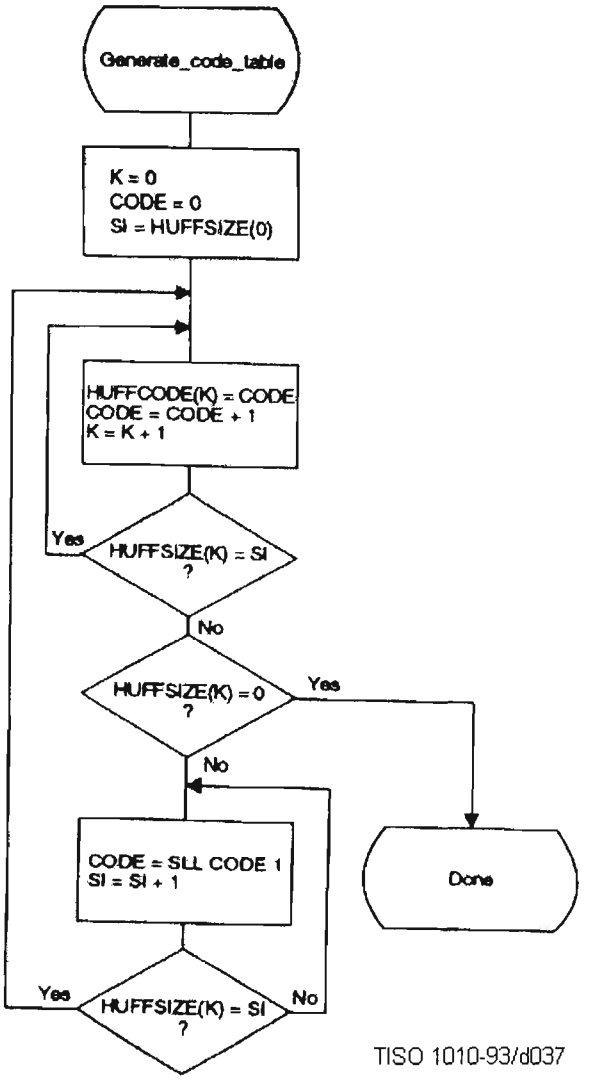


Figure A.2 Generation of table of Huffman codes

The procedure diagram shown in Figure A.3 is the procedure of the JPEG sequential encoding of AC coefficients using Huffman coding. The procedure diagram in Figure A.4 is the associated procedure for the encoding of non-zero AC coefficients. This procedure is called from the procedure shown in Figure A.3. These procedure diagrams are reproduced from Annex F [52]. The research algorithms used a modified version of Figure A.3 to encode the DC coefficient along with the AC coefficients. This was achieved by modification of the loop boundaries in Figure A.3.

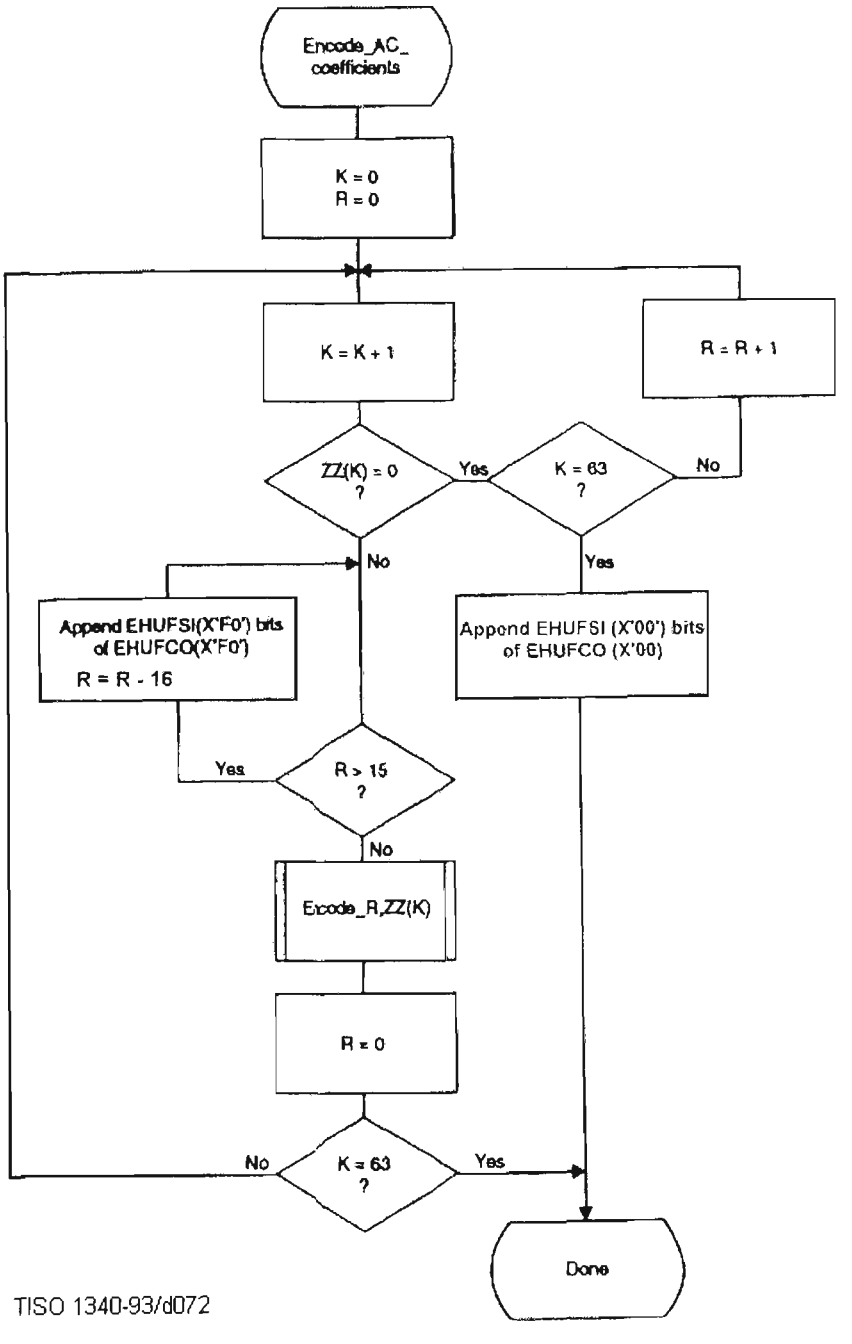


Figure A.3 Sequential encoding of AC coefficients with Huffman coding

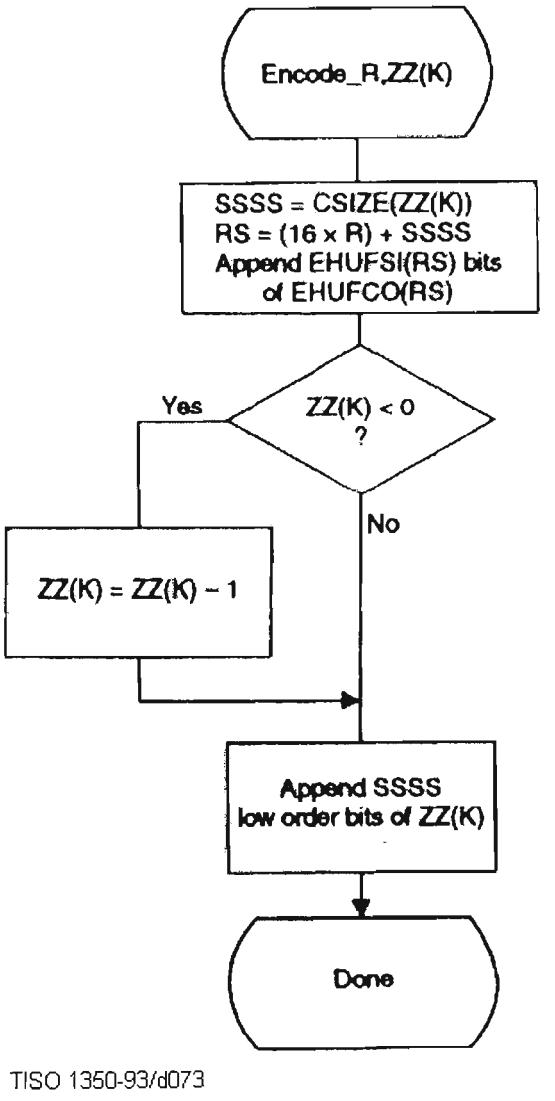


Figure A.4 Sequential encoding of a non-zero AC coefficient

A.5 Simulation Run Data

Table A.7 shows the trial run data for the timing of the JPEG compression components on the Pentium processor used for the simulation. All times are in milliseconds and represent the times for the processing of all 4096 image blocks. The average times, and the relative percentages they represent in overall image processing are displayed, and these are the figures shown in Section 5.5.4 Chapter 5 against the comparative transputer percentages.

Table A.7 Timing trials for JPEG compression components

Trial	Get Block (ms)	Level Shift (ms)	DCT (ms)	Quantization (ms)	Huffman /Store Block (ms)
1	880	262	9536	1516	1586
2	820	372	9596	1476	1526
3	820	312	9596	1406	1576
4	930	372	9536	1466	1576
5	880	262	9606	1526	1626
averages	866	316	9574	1478	1578
perc (%)	6.27	2.29	69.32	10.70	11.42

The following table, Table A.8, is a sample of the data collected for processing from the simulated trial runs in Chapter 5. For each run with a number of worker processors, three groups of trial data were collected for the first 100 simulated time quantums, see Section 5.6 Chapter 5. Of these trials, one was selected for further processing, from which the increase and differences in the output and input queue lengths were obtained.

The data in Table A.8, shows the data collected for one trial of the first 100 simulated time quantums, for seven simulated worker processors. The difference and increase in the output and input queue lengths are not shown here, but can be seen in the trial data files on the accompanying diskette.

Table A.8 Sample data for trial 1 of simulation of 7 worker processors.

Schedule Round	Output Queue Length	Input Queue Length	Transformed Blocks	Completed Blocks	
1	120	7	7	0	
2	246	14	21	7	
3	372	19	35	15	
4	499	24	49	24	
5	632	30	56	25	
6	755	35	63	27	
7	880	41	77	35	
8	1007	47	91	43	
9	1132	57	105	47	
10	1257	62	119	56	
11	1267	68	133	64	
12	1392	73	147	73	
13	1418	79	161	81	
14	1553	85	168	82	
15	1680	83	174	90	
16	1798	89	188	98	
17	1925	95	202	106	
18	2043	100	215	114	
19	2170	105	229	123	
20	2264	111	243	131	
21	2392	116	257	140	
22	2513	121	270	148	
23	2636	127	284	156	
24	2752	141	298	156	
25	2886	147	305	157	
26	3004	147	312	164	
27	3127	153	326	172	
28	3253	157	339	181	
29	3372	162	352	189	
30	3501	168	366	197	
31	3627	173	380	206	
32	3676	180	394	213	
33	3681	186	408	221	
34	3667	183	422	238	
35	3654	179	435	255	
36	3641	175	448	272	
37	3634	165	455	289	
38	3627	156	462	305	
39	3613	153	476	322	
40	3599	150	490	339	
41	3585	147	504	356	
42	3572	143	517	373	
43	3558	140	531	390	
44	3544	138	545	406	
45	3530	135	559	423	
46	3516	132	573	440	

47	3502	129	587	457	
48	3488	126	601	474	
49	3475	122	614	491	
50	3461	119	628	508	
51	3447	116	642	525	
52	3433	113	656	542	
53	3419	110	670	559	
54	3405	107	684	576	
55	3391	104	698	593	
56	3378	100	711	610	
57	3365	97	724	626	
58	3351	94	738	643	
59	3337	91	752	660	
60	3323	88	766	677	
61	3309	85	780	694	
62	3295	83	794	710	
63	3281	80	808	727	
64	3268	76	821	744	
65	3254	73	835	761	
66	3240	70	849	778	
67	3226	68	863	794	
68	3212	66	877	810	
69	3198	64	891	826	
70	3184	61	905	843	
71	3170	58	919	860	
72	3156	55	933	877	
73	3143	51	946	894	
74	3129	48	960	911	
75	3115	45	974	928	
76	3101	42	988	945	
77	3087	39	1002	962	
78	3073	37	1016	978	
79	3059	34	1030	995	
80	3045	31	1044	1012	
81	3031	28	1058	1029	
82	3018	24	1071	1046	
83	3004	22	1085	1062	
84	2990	19	1099	1079	
85	2976	16	1113	1096	
86	2962	14	1127	1113	
87	2948	14	1141	1127	
88	2934	14	1155	1141	
89	2920	14	1169	1155	
90	2906	14	1183	1169	
91	2892	14	1197	1183	
92	2878	22	1211	1188	
93	2864	20	1225	1204	
94	2850	17	1239	1221	
95	2836	14	1253	1238	
96	2822	14	1267	1253	
97	2808	14	1281	1267	
98	2794	14	1295	1281	
99	2780	14	1309	1295	
100	2766	14	1323	1309	

To show all data collected for all trials would occupy too much space. A diskette is provided with all data collected for the simulation runs of Chapter 5. This data is

collected into a number of *Excel* spreadsheet files. Each file contains the collected data for the three trial runs for a specific number of simulated worker processors. Each of the files further contains the derived figures of the increase and difference in the output and input queue lengths from one simulated time quantum to the next, for the trial selected for further processing.

The files contained on the diskette are

chpW1trials.xls - trial data for 1 simulated worker processor
chpW2trials.xls - trial data for 2 simulated worker processors
chpW3trials.xls - trial data for 3 simulated worker processors
chpW4trials.xls - trial data for 4 simulated worker processors
chpW5trials.xls - trial data for 5 simulated worker processors
chpW6trials.xls - trial data for 6 simulated worker processors
chpW7trials.xls - trial data for 7 simulated worker processors
chpW8trials.xls - trial data for 8 simulated worker processors
chpW9trials.xls - trial data for 9 simulated worker processors
chpW10trials.xls - trial data for 10 simulated worker processors
chpW7trialsSpec.xls - trial data for 7 simulated worker processors, all time quantums

The last file represents data collected for a simulation run with 7 simulated worker processors, including all simulated time quantums. This data was collected to test a hypothesis made in Section 5.6 Chapter 5, regarding averages over all time quantums versus the first 100 time quantums.

APPENDIX B

PARALLEL C TRANSPUTER CODE

B.1 Introduction

The following pages in this Appendix contain the code for the Parallel C programs implementing the parallel JPEG algorithms developed and discussed in Chapter 4.

B.2 Configuration File for Sequential Program SV1

```
!
!   Configuration file for 3 transputer cpu System  DCT Transform
!
!  file -> dct.cfg  for SV1.c

processor HOST    ! host PC with transputer board
processor ROOT    ! root transputer node connected to host
processor P001    ! exists but is not used in SV1
processor P002    ! exists but is not used in SV1

wire ? ROOT[0] HOST[0]
wire ? P001[1] ROOT[2]
wire ? P002[1] P001[2]

task afserver ins=1 outs=1
task filter   ins=2 outs=2 data=10k
task sv1      ins=3 outs=3

place afserver HOST
place filter   ROOT
place sv1      ROOT

connect ? afserver[0] filter[0]
connect ? filter[0]   afserver[0]
connect ? filter[1]   sv1[1]
connect ? sv1[1]      filter[1]
```

B.3 Parallel C code of program SV1

```
/*      SV1.c   Newer Faster version

  The size of this two dimensional (M x N) matrix is  8 x 8,
  ie  M = 8, and N = 8.

  Compression Algorithm C8F

  sv1 - Sequential Version 1  based on JPEG standard

  added the DCT.CFG file to this, the main CHAN parameters, and compiled
  and ran it like the parallel versions   26/2/97
*/

#include <stdio.h>
#include <math.h>
#include <float.h>
#include <stdlib.h>
#include <string.h>
#include "\masters\include\standard.h"
#include "\masters\include\admv3.h"
#include "\masters\include\cmpv3.h"
#include "\masters\include\uquant.h"
#include <chan.h>

/* Function Prototypes */

    void    Initialize(void);
    void    GetBlock(void);
    void    BuildCodes();
    void    Huffman();
    void    FDCT();
    void    DCT(double p[8]);

   /*
      note :  y is the image source block,  Y is the FDCT transform of y
          q  is the quantized block
   */

   int y [BlockSize][BlockSize];
   int q[BlockSize][BlockSize];
   long c[30];
   double   Y [BlockSize][BlockSize];
   double   X [BlockSize];
   char c1;


   /* variables for Huffman encoding procedures */
   FILE    *hufftab;  /* file containing Huffman Table specification */
   int     *EHUFSI;   /* huffman encoding tables  */
   ULint32 *EHUFCO;
   ULint32 buff;      /* buffer to which huffman codes are written */
   int32   count;     /* keeps track of how many bits used in buff */
   ULint32 total;     /* records total number of bits used to encode image */


   FILE *ADMfp, *CMPfp;
   char ADMfile[40], CMPfile[40], *p;

   /* Input Image File BUFFER and associated variable declarations */
   unsigned char *ifb;     /*  ifb - Image File Buffer */
   char ofb[5000];         /*  ofb - Output File Buffer */
   int Bused;              /*  blocks used from the buffer */

   struct CMPhead  cmpheader;
   struct ADMhead  admheader;

/*-------------------------------------------------------------------------*/
void main(int argc, char *argv[], char *envp[],
      CHAN *in[], int ins, CHAN *out[], int outs) {

   int NumBlocks,BlocksProcessed, j,k;
```

```
OpenFiles(argc, argv);
   Initialize();

   NumBlocks = cmpheader.BlkAcross * cmpheader.BlkDown;
   printf("Encoding %i Blocks\n",NumBlocks);
   for (BlocksProcessed = 0; BlocksProcessed < NumBlocks; BlocksProcessed++) {
      GetBlock();

      /* Level shift */
      for (j=0;j<8;j++) for (k=0;k<8;k++) y[j][k] = y[j][k]-128;

      FDCT();

      /* Quantization using the uniform quantization table in quant.h */
      for (j=0;j<BlockSize;j++)
        for (k=0;k<BlockSize;k++)
            q[j][k] = (Y[j][k]/Q[j][k])+ (Y[j][k]>0 ? 0.5 : -0.5);

      /* do the Huffman encoding as defined in ISO/IEC 10918-1 Annex F */
      Huffman();
   }

   /* flush the buffer if there is anything in it */
   if (count > 0) {
      /* shift it left to the MSB */
      buff <<= 32-count;
      fwrite(&buff,4,1,CMPfp);
      total += count;
   }
   fclose(CMPfp);
   fclose(ADMfp);
    printf("\nThe entire image was encoded in %7lu bits\n",total);
}

   /*--------------------------------------------------------------------------*/

   void OpenFiles(int argc, char *argv[])
   {
      if (!(argc >1)) {
        printf("Usage - code filename");
        exit(1);
      }

      /* get the filenames */
      strcpy(ADMfile,argv[1]);
      p = strrchr(ADMfile,92);
      if (p == NULL) strcpy(CMPfile,ADMfile); else  strcpy(CMPfile,++p);
      strcat(ADMfile,".ADM");
      strcat(CMPfile,".CMP");

      /* Try to open the ADM file */
      ADMfp = fopen(ADMfile,"rb");
      if (ADMfp == NULL) {
        printf("Error opening ADM file %s \n",ADMfile);
        exit(1);
      }

      /* and now open the CMP file (the empty compressed file) */
      CMPfp = fopen(CMPfile,"wb");
      if (CMPfp == NULL) {
        printf("Error opening CMP file %s \n",CMPfile);
        exit(1);
      }
      setvbuf(CMPfp, ofb, _IOFBF, 5000);
   }

   /*--------------------------------------------------------------------------*/

   void Initialize()
   {
      /* get the image header */
      fread(&admheader,sizeof(struct ADMhead),1,ADMfp);

      /* Now set the compressed file header and write it out */
      cmpheader.Version = ver;
      strcpy(cmpheader.Description,admheader.Description);
      cmpheader.BlkSize = BlockSize;
      cmpheader.BlkAcross = admheader.PixAcross / BlockSize;
      cmpheader.BlkDown = admheader.PixDown / BlockSize;
      cmpheader.DCBlk1 = 0;     /* for the moment  re-write the header later */
```

```
   strcpy(cmpheader.Mode,"Gray");
   fwrite(&cmpheader,sizeof(struct CMPhead),1,CMPfp);

   /* allocate the Image File Buffer */
   ifb = malloc(admheader.PixAcross*BlockSize);
   if (ifb == NULL) {
     printf("Could not Alloctae Image File Buffer\n");
     exit(1);
   }
   /* set the number of blocks used so as to initiate a ifb read */
   Bused = cmpheader.BlkAcross;

   /* initially start with empty buffer */
   buff = 0;  count = 0;  total = 0;

   /* Hauffman initialization stuff */
   BuildCodes();
}

/*-------------------------------------------------------------------*/

void GetBlock()
{
   int i,j, offset;

   if (Bused >= cmpheader.BlkAcross) {  /* initiate a read from the file */
     fread(ifb,admheader.PixAcross*BlockSize,1,ADMfp);
     Bused=0;    }

   offset = Bused * BlockSize;
   for (i=0; i < BlockSize; i++) {
         for (j=0; j < BlockSize; j++) {
         y[i][j] = ifb[offset+j]; }
         offset += admheader.PixAcross;
   }
   Bused++;
}

/*-------------------------------------------------------------------*/

        /* Huffman Code Table initialization Routines */

void BuildCodes() {

   int BITS[16], *HUFFVAL, *HUFFSIZE;
   unsigned int  *HUFFCODE;
   int i, j, k, lastk, code, si, HVTsze=0;

   printf("\n Building Huffman Code Tables\n");
   hufftab = fopen("ACtable.dat","r");
   if (hufftab == NULL) {
      printf("Error opening Huffman Code Table \n");
      exit(1);
   }

   /* read the code lengths  (for luminance ) Annex K ISO/IEC 10918-1 */
   for (i=0;i<16;i++) fscanf(hufftab," %x",&BITS[i]);

   /* Now read the Huffman values associated with the code lengths */
   for (i=0;i<16;i++) HVTsze += BITS[i];

   HUFFVAL = malloc(sizeof(int)*HVTsze);
   for (i=0;i<HVTsze;i++) fscanf(hufftab," %x",&HUFFVAL[i]);

   fclose(hufftab);

   /* now follow procedures in Annex C to generate the following tables */

   /* HUFFSIZE table  - slight changes because BITS indexed from 0*/
   HUFFSIZE = malloc((sizeof(int)*HVTsze)+1);
   k = 0;  i = 1;  j = 1;
   do {
     while (j <= BITS[i-1]) {
     HUFFSIZE[k] = i;
     k++;
     j++;
     }
     i++;
     j = 1;
   } while (i <= 16 );
   HUFFSIZE[k] = 0;
```

```
    lastk = k;

    /* HUFFCODE table */
    HUFFCODE = malloc(sizeof(unsigned int)*HVTsze);
    k = 0;  code = 0;  si = HUFFSIZE[0];
    do {
      do {
          HUFFCODE[k] = code;
          code = code + 1;
          k = k + 1;
      } while (HUFFSIZE[k] == si);
      If (HUFFSIZE[k] == 0) break;
      do {
          code = code << 1;
          si = si + 1;
      } while (HUFFSIZE[k] != si);
   } while (HUFFSIZE[k] == si);

    /* Now create encoding procedure code tables  EHUFCO & EHUFSI */
    EHUFCO = malloc(sizeof(ULint32)*256);
    EHUFSI = malloc(sizeof(int)*256);
    for (i=0;i<256;i++) {
      EHUFCO[i] = 0;
      EHUFSI[i] = 0;  }

    k = 0;
    do {
      i = HUFFVAL[k];
      EHUFSI[i] = HUFFSIZE[k];
      /* code component - align bits to the MSB end of the 32 bit int */
      EHUFCO[i] = (ULint32)(HUFFCODE[k]) << (32-EHUFSI[i]);
      k++;
    } while (k < lastk);

    /* reallocate some of the memory back */
    free(HUFFVAL);
    free(HUFFSIZE);
    free(HUFFCODE);
}

/*-------------------------------------------------------------------------*/

/*     Huffman encoding routines based on quantised block q */

/* Append assumes the code parameter has all bits aligned to the MSB */
void Append(int bits, ULint32 code) {

    int i;

    for (i=1; i<=bits; i++) {
      buff <<= 1;
      buff += (code >> 31) & 1;
      code <<= 1;
      count ++;

      if (count == 32) {                /* 32 bit integer */
          fwrite(&buff,4,1,CMPfp);       /* 32 bits = 4 bytes */
          count = 0;
          buff = 0;
          total += 32; }
    }
}

Csize (int coeff) {
    if (abs(coeff) == 1) return 1;
    else if (abs(coeff) <= 3)   return 2;
    else if (abs(coeff) <= 7)   return 3;
    else if (abs(coeff) <= 15) return 4;
    else if (abs(coeff) <= 31) return 5;
    else if (abs(coeff) <= 63) return 6;
    else if (abs(coeff) <= 127) return 7;
    else if (abs(coeff) <= 255) return 8;
    else if (abs(coeff) <= 511) return 9;
    else if (abs(coeff) <= 1023) return 10;
    else {
          printf("problen encountered in Csize function, coeff out of range.\n");
          exit(1);  }
}

void Huffman() {
    int ZZ[64], k, r, ssss, rs;
```

```
      ULint32 cde;

      /* first re-arrange quantized coefficients in Zig-Zag sequence */
      ZZ[0]=q[0][0];  ZZ[1]=q[0][1];  ZZ[2]=q[1][0];  ZZ[3]=q[2][0];
      ZZ[4]=q[1][1];  ZZ[5]=q[0][2];  ZZ[6]=q[0][3];  ZZ[7]=q[1][2];
      ZZ[8]=q[2][1];  ZZ[9]=q[3][0];  ZZ[10]=q[4][0]; ZZ[11]=q[3][1];
      ZZ[12]=q[2][2]; ZZ[13]=q[1][3]; ZZ[14]=q[0][4]; ZZ[15]=q[0][5];
      ZZ[16]=q[1][4]; ZZ[17]=q[2][3]; ZZ[18]=q[3][2]; ZZ[19]=q[4][1];
      ZZ[20]=q[5][0]; ZZ[21]=q[6][0]; ZZ[22]=q[5][1]; ZZ[23]=q[4][2];
      ZZ[24]=q[3][3]; ZZ[25]=q[2][4]; ZZ[26]=q[1][5]; ZZ[27]=q[0][6];
      ZZ[28]=q[0][7]; ZZ[29]=q[1][6]; ZZ[30]=q[2][5]; ZZ[31]=q[3][4];
      ZZ[32]=q[4][3]; ZZ[33]=q[5][2]; ZZ[34]=q[6][1]; ZZ[35]=q[7][0];
      ZZ[36]=q[7][1]; ZZ[37]=q[6][2]; ZZ[38]=q[5][3]; ZZ[39]=q[4][4];
      ZZ[40]=q[3][5]; ZZ[41]=q[2][6]; ZZ[42]=q[1][7]; ZZ[43]=q[2][7];
      ZZ[44]=q[3][6]; ZZ[45]=q[4][5]; ZZ[46]=q[5][4]; ZZ[47]=q[6][3];
      ZZ[48]=q[7][2]; ZZ[49]=q[7][3]; ZZ[50]=q[6][4]; ZZ[51]=q[5][5];
      ZZ[52]=q[4][6]; ZZ[53]=q[3][7]; ZZ[54]=q[4][7]; ZZ[55]=q[5][6];
      ZZ[56]=q[6][5]; ZZ[57]=q[7][4]; ZZ[58]=q[7][5]; ZZ[59]=q[6][6];
      ZZ[60]=q[5][7]; ZZ[61]=q[6][7]; ZZ[62]=q[7][6]; ZZ[63]=q[7][7];

      /* Encode AC cofficients - procedure from annex F ISO/IEC 10918-1 */
      k = -1;   /* index into zig-zag sequence : -1 codes DC coefficient */
      r = 0;    /* run length of zero coefficients */

      jump1:  /* jump point for goto.... I KNOW! but it was the easiest way */

      k++;
      if (ZZ[k] == 0) {
        if (k == 63) {
            Append(EHUFSI[0],EHUFCO[0]);
            goto jump2;
        }
        r++;
        goto jump1;
      }
      while (r > 15) {
        Append(EHUFSI[240],EHUFCO[240]);
        r -= 16;  }

      /* Encode R & ZZ(k) at this point */
      ssss = Csize(ZZ[k]);
      rs = (r * 16) + ssss;
      Append(EHUFSI[rs],EHUFCO[rs]);
      if (ZZ[k] < 0) ZZ[k]--;
      cde = (ULint32)(ZZ[k]) << (32-ssss);
      Append(ssss,cde);

      r = 0;
      if (k < 63) goto jump1;

      jump2:  /* jump point for jump out of loop  */
      ;
   }

   /*-------------------------------------------------------------------*/

   /*             Fast Discrete Cosine Transform      */

void FDCT()
{  int i,j;

   /* set up double matrix Y which is to be transformed / be the result */
   for (i = 0; i < BlockSize; i++) {
      for (j = 0; j < BlockSize; j++) Y[i][j] = (double)y[i][j];
   }

   /* now do the 2D-FDCT() */

   /*  first dimension - down the columns  */
   for (i = 0; i < BlockSize; i++) {
      for (j = 0; j < BlockSize; j++) {
        X[j] = Y[j][i]; }
      DCT(X);
      for (j = 0; j < BlockSize; j++) Y[j][i] = X[j];
   }

   /*  second dimension - accross the rows */
   for (i = 0; i < BlockSize; i++) {
      for (j = 0; j < BlockSize; j++) {
        X[j] = Y[i][j]; }
      DCT(X);
```

```
      for (j = 0; j < BlockSize; j++) Y[i][j] = X[j];
   }
}

   /*              Fast Discrete Cosine Transform      */

   void DCT(double Z[BlockSize])
   {   double a[BlockSize], b[BlockSize], c[BlockSize], d[BlockSize];

      /*  C8F Algorithm  */
      /* first butterfly loop */
      a[0] = Z[0] + Z[7];
      a[1] = Z[1] + Z[6];
      a[2] = Z[2] + Z[5];
      a[3] = Z[3] + Z[4];
      a[4] = Z[3] - Z[4];
      a[5] = Z[2] - Z[5];
      a[6] = Z[1] - Z[6];
      a[7] = Z[0] - Z[7];

      /* second butterfly loop */
      b[0] = a[0] + a[3];
      b[1] = a[1] + a[2];
      b[2] = a[1] - a[2];
      b[3] = a[0] - a[3];
      b[4] = a[4];
      b[5] = (a[6] - a[5])*cos(pi/4.0);
      b[6] = (a[6] + a[5])*cos(pi/4.0);
      b[7] = a[7];

      /* third butterfly loop */
      c[0] = (b[0] + b[1]) * cos(pi/4.0);
      c[1] = (b[0] - b[1]) * cos(pi/4.0);
      c[2] = b[2]*sin(pi/8.0) + b[3]*cos(pi/8.0);  /* found mistake here ? */
      c[3] = b[3]*cos(3.0*pi/8.0) - b[2]*sin(3.0*pi/8.0);
      c[4] = b[4] + b[5];
      c[5] = b[4] - b[5];
      c[6] = b[7] - b[6];
      c[7] = b[7] + b[6];

      /* fourth & final butterfly loop */
      d[4] = c[4]*sin(pi/16.0) + c[7]*cos(pi/16.0);
      d[5] = c[5]*sin(5.0*pi/16.0) + c[6]*cos(5.0*pi/16.0);
      d[6] = c[6]*cos(3.0*pi/16.0) - c[5]*sin(3.0*pi/16.0);
      d[7] = c[7]*cos(7.0*pi/16.0) - c[4]*sin(7.0*pi/16.0);

      /* now calculate normalized Forward Transform Coefficients */
      Z[0] = c[0]/4.0;
      Z[1] = d[4]/4.0;
      Z[2] = c[2]/4.0;
      Z[3] = d[6]/4.0;
      Z[4] = c[1]/4.0;
      Z[5] = d[5]/4.0;
      Z[6] = c[3]/4.0;
      Z[7] = d[7]/4.0;
   }
```

B.4 Configuration File for Parallel Program PV1

```
!
!   Configuration file for 3 transputer cpu System  DCT Transform
!
!  file -> dct.cfg   for PV1.c

processor HOST    ! host PC with transputer board
processor ROOT    ! root transputer node connected to host
processor P001    ! transputer node to do the worker DCT task
processor P002    ! exists but is not used in PV1

wire ? ROOT[0] HOST[0]
wire ? P001[1] ROOT[2]
wire ? P002[1] P001[2]

task afserver ins=1 outs=1
task filter   ins=2 outs=2 data=10k
task dctmain  ins=3 outs=3
task dcttask  ins=1 outs=1

place afserver HOST
place filter   ROOT
place dctmain  ROOT
place dcttask  P001

connect ? afserver[0] filter[0]
connect ? filter[0]   afserver[0]
connect ? filter[1]   dctmain[1]
connect ? dctmain[1]  filter[1]
connect ? dctmain[2]  dcttask[0]
connect ? dcttask[0]  dctmain[2]
```

B.5 Parallel C code of Program PV1

B.5.1 Code for Processor P_0

```
/*     code.c   Newer Faster version
  The size of this two dimensional (M x N) matrix is  8 x 8,
  ie  M = 8, and N = 8.
  Compression Algorithm C8F
  pv1 - parallel Version 1 dctmain task based on JPEG standard
*/

#include <stdio.h>
#include <math.h>
#include <float.h>
#include <stdlib.h>
#include <string.h>
#include "\masters\include\standard.h"
#include "\masters\include\admv3.h"
#include "\masters\include\cmpv3.h"
#include "\masters\include\uquant.h"
#include <chan.h>

/* Function Prototypes */
    void    Initialize(void);
    void    GetBlock(void);
    void    BuildCodes(void);
    void    Huffman(void);

   /*
      note :  y is the image source block,  Y is the FDCT transform of y
          q  is the quantized block
   */
   int y [BlockSize][BlockSize];
   int q[BlockSize][BlockSize];
   long c[30];
   double   Y [BlockSize][BlockSize];
   double   X [BlockSize];
   char c1;

   /* variables for Huffman encoding procedures */
   FILE    *hufftab;  /* file containing Huffman Table specification */
   int     *EHUFSI;   /* huffman encoding tables  */
   ULint32 *EHUFCO;
   ULint32 buff;      /* buffer to which huffman codes are written */
   int32   count;     /* keeps track of how many bits used in buff */
   ULint32 total;     /* records total number of bits used to encode image */

   FILE *ADMfp, *CMPfp;
   char ADMfile[40], CMPfile[40], *p;

   /* Input Image File BUFFER and associated variable declarations */
   unsigned char *ifb;     /*  ifb - Image File Buffer */
   char ofb[5000];         /*  ofb - Output File Buffer */
   int Bused;              /*  blocks used from the buffer */

   struct CMPhead  cmpheader;
   struct ADMhead  admheader;

/*-------------------------------------------------------------------------*/
main(int argc, char *argv[], char *envp[], CHAN *in[], int inlen,
     CHAN *out[], int outlen) {

   int NumBlocks,BlocksProcessed, j,k;
   OpenFiles(argc, argv);
   Initialize();
   NumBlocks = cmpheader.BlkAcross * cmpheader.BlkDown;
   printf("Encoding %i Blocks\n",NumBlocks);

   /* Get the First block and send it off to the worker task */
   GetBlock();
   /* Level shift */
   for (j=0;j<8;j++) for (k=0;k<8;k++) y[j][k] = y[j][k]-128;
   chan_out_message((sizeof(int)*BlockSize*BlockSize),y,out[2]);
```

```
   /* set Blocks Processed = 1 in for loop since already sent one */
   for (BlocksProcessed = 1; BlocksProcessed < NumBlocks; BlocksProcessed++) {
      /* Assemble the next block to be processed */
      GetBlock();
      /* Level shift */
      for (j=0;j<8;j++) for (k=0;k<8;k++) y[j][k] = y[j][k]-128;
      /* wait for a block to come back from the worker task */
      chan_in_message(sizeof(double)*BlockSize*BlockSize,Y,in[2]);

      /* Now send off the next block */
      chan_out_message(sizeof(int)*BlockSize*BlockSize,y,out[2]);

      /* Now process the block just recieved */
      /* Quantization using the uniform quantization table in quant.h */
      for (j=0;j<BlockSize;j++) for (k=0;k<BlockSize;k++)
            q[j][k] = (Y[j][k]/Q[j][k])+ (Y[j][k]>0 ? 0.5 : -0.5);
      /* do the Huffman encoding as defined in ISO/IEC 10918-1 Annex F */
      Huffman();
   }

   /* now get the last straggler block */
   /* wait for a block to come back from the worker task */
   chan_in_message(sizeof(double)*BlockSize*BlockSize,Y,in[2]);
   /* Now process the block just recieved */
   /* Quantization using the uniform quantization table in quant.h */
   for (j=0;j<BlockSize;j++) for (k=0;k<BlockSize;k++)
        q[j][k] = (Y[j][k]/Q[j][k])+ (Y[j][k]>0 ? 0.5 : -0.5);
   /* do the Huffman encoding as defined in ISO/IEC 10918-1 Annex F */
   Huffman();


   /* flush the buffer if there is anything in it */
   if (count > 0) {
      /* shift it left to the MSB */
      buff <<= 32-count;
      fwrite(&buff,4,1,CMPfp);
      total += count;  }

   fclose(CMPfp);
   fclose(ADMfp);
   printf("\nThe entire image was encoded in %7lu bits\n",total);
}

   /*-------------------------------------------------------------------------*/

   void OpenFiles(int argc, char *argv[])   {

      if (!(argc >1)) {
        printf("Usage - code filename");
        exit(1);   }

      /* get the filenames */
      strcpy(ADMfile,argv[1]);
      p = strrchr(ADMfile,92);
      if (p == NULL) strcpy(CMPfile,ADMfile); else  strcpy(CMPfile,++p);
      strcat(ADMfile,".ADM");
      strcat(CMPfile,".CMP");

      /* Try to open the ADM file */
      ADMfp = fopen(ADMfile,"rb");
      if (ADMfp == NULL) {
        printf("Error opening ADM file %s \n",ADMfile);
        exit(1); }

      /* and now open the CMP file (the empty compressed file) */
      CMPfp = fopen(CMPfile,"wb");
      if (CMPfp == NULL) {
        printf("Error opening CMP file %s \n",CMPfile);
        exit(1);   }
      setvbuf(CMPfp, ofb, _IOFBF, 5000);
   }

   /*-------------------------------------------------------------------------*/

   void Initialize()
   {
      /* get the image header */
      fread(&admheader,sizeof(struct ADMhead),1,ADMfp);

      /* Now set the compressed file header and write it out */
```

```
    cmpheader.Version = ver;
    strcpy(cmpheader.Description,admheader.Description);
    cmpheader.BlkSize = BlockSize;
    cmpheader.BlkAcross = admheader.PixAcross / BlockSize;
    cmpheader.BlkDown = admheader.PixDown / BlockSize;
    cmpheader.DCBlk1 = 0;     /* for the moment  re-write the header later */
    strcpy(cmpheader.Mode,"Gray");
    fwrite(&cmpheader,sizeof(struct CMPhead),1,CMPfp);

    /* allocate the Image File Buffer */
    ifb = malloc(admheader.PixAcross*BlockSize);
    if (ifb == NULL) {
      printf("Could not Alloctae Image File Buffer\n");
      exit(1);  }
    /* set the number of blocks used so as to initiate a ifb read */
    Bused = cmpheader.BlkAcross;

    /* initially start with empty buffer */
    buff = 0;  count = 0;  total = 0;

    /* Hauffman initialization stuff */
    BuildCodes();
}

/*--------------------------------------------------------------------*/

void GetBlock()
{
    int i,j, offset;

    if (Bused >= cmpheader.BlkAcross) {  /* initiate a read from the file */
      fread(ifb,admheader.PixAcross*BlockSize,1,ADMfp);
      Bused=0;  }
    offset = Bused * BlockSize;
    for (i=0; i < BlockSize; i++) {
          for (j=0; j < BlockSize; j++) { y[i][j] = ifb[offset+j]; }
          offset += admheader.PixAcross;     }
    Bused++;
}

/*--------------------------------------------------------------------*/

        /* Huffman Code Table initialization Routines */

void BuildCodes() {

    int BITS[16], *HUFFVAL, *HUFFSIZE;
    unsigned int  *HUFFCODE;
    int i, j, k, lastk, code, si, HVTsze=0;

    printf("\n Building Huffman Code Tables\n");
    hufftab = fopen("ACtable.dat","r");
    if (hufftab == NULL) {
       printf("Error opening Huffman Code Table \n");
       exit(1);    }

    /* read the code lengths  (for luminance ) Annex K ISO/IEC 10918-1 */
    for (i=0;i<16;i++) fscanf(hufftab," %x",&BITS[i]);

    /* Now read the Huffman values associated with the code lengths */
    for (i=0;i<16;i++) HVTsze += BITS[i];

    HUFFVAL = malloc(sizeof(int)*HVTsze);
    for (i=0;i<HVTsze;i++) fscanf(hufftab," %x",&HUFFVAL[i]);

    fclose(hufftab);

    /* now follow procedures in Annex C to generate the following tables */

    /* HUFFSIZE table  - slight changes because BITS indexed from 0*/
    HUFFSIZE = malloc((sizeof(int)*HVTsze)+1);
    k = 0;  i = 1;  j = 1;
    do {
          while (j <= BITS[i-1]) {
          HUFFSIZE[k] = i;
          k++;
          j++; }
          i++;
          j = 1;
    } while (i <= 16 );
    HUFFSIZE[k] = 0;
```

```
    lastk = k;

    /* HUFFCODE table */
    HUFFCODE = malloc(sizeof(unsigned int)*HVTsze);
    k = 0;  code = 0;  si = HUFFSIZE[0];
    do {
          do {
              HUFFCODE[k] = code;
              code = code + 1;
              k = k + 1;
          ) while (HUFFSIZE[k] == si);
          if (HUFFSIZE[k] == 0) break;
          do {
              code = code << 1;
              si = si + 1;
          ) while (HUFFSIZE[k] != si);
    ) while (HUFFSIZE[k] == si);

    /* Now create encoding procedure code tables  EHUFCO & EHUFSI */
    EHUFCO = malloc(sizeof(ULint32)*256);
    EHUFSI = malloc(sizeof(int)*256);
    for (i=0;i<256;i++) {
          EHUFCO[i] = 0;
          EHUFSI[i] = 0;  )
    k = 0;
    do {
          i = HUFFVAL[k];
          EHUFSI[i] = HUFFSIZE[k];
          /* code component - align bits to the MSB end of the 32 bit int */
          EHUFCO[i] = (ULint32)(HUFFCODE[k]) << (32-EHUFSI[i]);
          k++;
    } while (k < lastk);

    /* reallocate some of the memory back */
    free(HUFFVAL);
    free(HUFFSIZE);
    free(HUFFCODE);

}

/*-------------------------------------------------------------------------*/

/*     Huffman encoding routines based on quantised block q */

/* Append assumes the code parameter has all bits aligned to the MSB */
void Append(int bits, ULint32 code) {

    int i;

    for (i=1; i<=bits; i++) {
          buff <<= 1;
          buff += (code >> 31) & 1;
          code <<= 1;
          count ++;
          if (count == 32) {                /* 32 bit integer */
              fwrite(&buff,4,1,CMPfp);       /* 32 bits = 4 bytes */
              count = 0;
              buff = 0;
              total += 32;  }
    )
)

Csize (int coeff) {
    if (abs(coeff) == 1) return 1;
    else if (abs(coeff) <= 3)   return 2;
    else if (abs(coeff) <= 7)   return 3;
    else if (abs(coeff) <= 15) return 4;
    else if (abs(coeff) <= 31) return 5;
    else if (abs(coeff) <= 63) return 6;
    else if (abs(coeff) <= 127) return 7;
    else if (abs(coeff) <= 255) return 8;
    else if (abs(coeff) <= 511) return 9;
    else if (abs(coeff) <= 1023) return 10;
    else {
          printf("problen encountered in Csize function, coeff out of range.\n");
          exit(1);  )
)

void Huffman() {
    int ZZ[64], k, r, ssss, rs;
    ULint32 cde;
```

```
    /* first re-arrange quantized coefficients in Zig-Zag sequence */
    ZZ[0]=q[0][0];  ZZ[1]=q[0][1];  ZZ[2]=q[1][0];  ZZ[3]=q[2][0];
    ZZ[4]=q[1][1];  ZZ[5]=q[0][2];  ZZ[6]=q[0][3];  ZZ[7]=q[1][2];
    ZZ[8]=q[2][1];  ZZ[9]=q[3][0];  ZZ[10]=q[4][0]; ZZ[11]=q[3][1];
    ZZ[12]=q[2][2]; ZZ[13]=q[1][3]; ZZ[14]=q[0][4]; ZZ[15]=q[0][5];
    ZZ[16]=q[1][4]; ZZ[17]=q[2][3]; ZZ[18]=q[3][2]; ZZ[19]=q[4][1];
    ZZ[20]=q[5][0]; ZZ[21]=q[6][0]; ZZ[22]=q[5][1]; ZZ[23]=q[4][2];
    ZZ[24]=q[3][3]; ZZ[25]=q[2][4]; ZZ[26]=q[1][5]; ZZ[27]=q[0][6];
    ZZ[28]=q[0][7]; ZZ[29]=q[1][6]; ZZ[30]=q[2][5]; ZZ[31]=q[3][4];
    ZZ[32]=q[4][3]; ZZ[33]=q[5][2]; ZZ[34]=q[6][1]; ZZ[35]=q[7][0];
    ZZ[36]=q[7][1]; ZZ[37]=q[6][2]; ZZ[38]=q[5][3]; ZZ[39]=q[4][4];
    ZZ[40]=q[3][5]; ZZ[41]=q[2][6]; ZZ[42]=q[1][7]; ZZ[43]=q[2][7];
    ZZ[44]=q[3][6]; ZZ[45]=q[4][5]; ZZ[46]=q[5][4]; ZZ[47]=q[6][3];
    ZZ[48]=q[7][2]; ZZ[49]=q[7][3]; ZZ[50]=q[6][4]; ZZ[51]=q[5][5];
    ZZ[52]=q[4][6]; ZZ[53]=q[3][7]; ZZ[54]=q[4][7]; ZZ[55]=q[5][6];
    ZZ[56]=q[6][5]; ZZ[57]=q[7][4]; ZZ[58]=q[7][5]; ZZ[59]=q[6][6];
    ZZ[60]=q[5][7]; ZZ[61]=q[6][7]; ZZ[62]=q[7][6]; ZZ[63]=q[7][7];

    /* Encode AC cofficients - procedure from annex F ISO/IEC 10918-1 */
    k = -1;    /* index into zig-zag sequence : -1 codes DC coefficient */
    r = 0;     /* run length of zero coefficients */

    jump1:  /* jump point for goto.... I KNOW! but it was the easiest way */

    k++;
    if (ZZ[k] == 0) {
          if (k == 63) {
              Append(EHUFSI[0],EHUFCO[0]);
              goto jump2;   }
          r++;
          goto jump1;
    }
    while (r > 15) {
          Append(EHUFSI[240],EHUFCO[240]);
          r -= 16;  }

    /* Encode R & ZZ(k) at this point */
    ssss = Csize(ZZ[k]);
    rs = (r * 16) + ssss;
    Append(EHUFSI[rs],EHUFCO[rs]);
    if (ZZ[k] < 0) ZZ[k]--;
    cde = (ULint32)(ZZ[k]) << (32-ssss);
    Append(ssss,cde);

    r = 0;
    if (k < 63) goto jump1;

    jump2:  /* jump point for jump out of loop  */

    ;
}

/*-------------------------------------------------------------------------*/
```

B.5.2 Code for Processor P_1

```
/*  pv1 - parallel version 1  dcttask prog */

#include <math.h>
#include <float.h>
#include <stdlib.h>
#include "\masters\include\standard.h"
#include <chan.h>

int      y[BlockSize][BlockSize];    /* level shifted image source block */
double   Y[BlockSize][BlockSize];    /* DCT Transformed double block */
double   X[BlockSize];

/*             Fast Discrete Cosine Transform      */

main(int argc, char *argv[], char *envp[], CHAN *in[], int inlen,
     CHAN *out[], int outlen) {

    int i,j;

    for (;;) {      /* infinite loop to recieve & process blocks */
        /* wait for input from the transputer root */
        chan_in_message(sizeof(int)*BlockSize*BlockSize,y,in[0]);

        /* set up double matrix Y which is to be transformed and be the result */
        for (i = 0; i < BlockSize; i++) {
            for (j = 0; j < BlockSize; j++) Y[i][j] = (double)y[i][j];
        }

        /* now do the 2D-FDCT() */

        /*  first dimension - down the columns  */
        for (i = 0; i < BlockSize; i++) {
            for (j = 0; j < BlockSize; j++) {
                X[j] = Y[j][i];
            }
            DCT(X);
            for (j = 0; j < BlockSize; j++) Y[j][i] = X[j];
        }

        /*  second dimension - accross the rows */
        for (i = 0; i < BlockSize; i++) {
            for (j = 0; j < BlockSize; j++) {
                X[j] = Y[i][j];
            }
            DCT(X);
            for (j = 0; j < BlockSize; j++) Y[i][j] = X[j];
        }

          /* now send the result back to the transputer root task */
          chan_out_message(sizeof(double)*BlockSize*BlockSize,Y,out[0]);

    }  /* end of infinite for loop */
}

/*-------------------------------------------------------------------------*/

/*             Fast Discrete Cosine Transform      */

void DCT(double Z[BlockSize]) {
    double a[BlockSize], b[BlockSize], c[BlockSize], d[BlockSize];

    /*  C8F Algorithm  */

    /* first butterfly loop */
    a[0] = Z[0] + Z[7];
    a[1] = Z[1] + Z[6];
    a[2] = Z[2] + Z[5];
    a[3] = Z[3] + Z[4];
    a[4] = Z[3] - Z[4];
    a[5] = Z[2] - Z[5];
    a[6] = Z[1] - Z[6];
    a[7] = Z[0] - Z[7];

    /* second butterfly loop */
    b[0] = a[0] + a[3];
    b[1] = a[1] + a[2];
```

```
    b[2] = a[1] - a[2];
    b[3] = a[0] - a[3];
    b[4] = a[4];
    b[5] = (a[6] - a[5])*cos(pi/4.0);
    b[6] = (a[6] + a[5])*cos(pi/4.0);
    b[7] = a[7];

    /* third butterfly loop */
    c[0] = (b[0] + b[1]) * cos(pi/4.0);
    c[1] = (b[0] - b[1]) * cos(pi/4.0);
    c[2] = b[2]*sin(pi/8.0) + b[3]*cos(pi/8.0);  /* found mistake here ? */
    c[3] = b[3]*cos(3.0*pi/8.0) - b[2]*sin(3.0*pi/8.0);
    c[4] = b[4] + b[5];
    c[5] = b[4] - b[5];
    c[6] = b[7] - b[6];
    c[7] = b[7] + b[6];

    /* fourth & final butterfly loop */
    d[4] = c[4]*sin(pi/16.0) + c[7]*cos(pi/16.0);
    d[5] = c[5]*sin(5.0*pi/16.0) + c[6]*cos(5.0*pi/16.0);
    d[6] = c[6]*cos(3.0*pi/16.0) - c[5]*sin(3.0*pi/16.0);
    d[7] = c[7]*cos(7.0*pi/16.0) - c[4]*sin(7.0*pi/16.0);

    /* now calculate normalized Forward Transform Coefficients */
    Z[0] = c[0]/4.0;
    Z[1] = d[4]/4.0;
    Z[2] = c[2]/4.0;
    Z[3] = d[6]/4.0;
    Z[4] = c[1]/4.0;
    Z[5] = d[5]/4.0;
    Z[6] = c[3]/4.0;
    Z[7] = d[7]/4.0;
}

/*-------------------------------------------------------------------------*/
```

B.6 Configuration File for Processor Farm Program PV2

```
!
!   Configuration file for 3 transputer cpu System  using processor Farm
!
!  file -> dct.cfg   for PV2.c

task master
task worker data=20k
```

B.7 Parallel C code of Processor Farm Program PV2

B.7.1 Code for Master task

```
/*      code.c   Newer Faster version
    The size of this two dimensional (M x N) matrix is  8 x 8,
    ie  M = 8, and N = 8.
    Compression Algorithm C8F
    pv2 - Parallel Version 2 master task
*/

#include <stdio.h>
#include <math.h>
#include <float.h>
#include <stdlib.h>
#include <string.h>
#include "\masters\include\standard.h"
#include "\masters\include\admv3.h"
#include "\masters\include\cmpv3.h"
#include "\masters\include\uquant.h"
#include <net.h>
#include <thread.h>
#include <par.h>
#include <sema.h>

/* Function Prototypes */

void    Initialize();
void    GetBlock();
void    OpenFiles(int, char *);
void    BuildCodes();
void    Huffman();
void    Receive();  /* used as a Thread to send Blocks */
void    Send();     /* used as a Thread to recieve Blocks */

int  y[BlockSize][BlockSize];  /* image source block */
double Y[BlockSize][BlockSize];
int q[BlockSize][BlockSize];
long c[30];
char c1;

/* variables for Huffman encoding procedures */
FILE    *hufftab;  /* file containing Huffman Table specification */
int     *EHUFSI;   /* huffman encoding tables  */
ULint32 *EHUFCO;
ULint32 buff;      /* buffer to which huffman codes are written */
int32   count;     /* keeps track of how many bits used in buff */
ULint32 total;     /* records total number of bits used to encode image */

FILE *ADMfp, *CMPfp;
char ADMfile[40], CMPfile[40], *p;

/* Input Image File BUFFER and associated variable declarations */
unsigned char *ifb;     /*  ifb - Image File Buffer */
char ofb[5000];         /*  ofb - Output File Buffer */
int Bused;              /*  blocks used from the buffer */

struct CMPhead  cmpheader;
struct ADMhead  admheader;

/* variables used to help synchronize between Main, Receive & Send threads */
int NumBlocks, BlocksProcessed;

/* Interface to Main Thread */
static SEMA main_resume;
/*-------------------------------------------------------------------------*/
void main(int argc, char argv[]) {

    int j,k;
```

```
    OpenFiles(argc, argv);
    Initialize();
    NumBlocks = cmpheader.BlkAcross * cmpheader.BlkDown;
    BlocksProcessed = 0;
    printf("Encoding %i Blocks\n",NumBlocks);
    /* now use par_sema beyond this point for access to C-RTL */
    /* initialize the resume semaphore */
    sema_init(&main_resume,0);

    /* Now start the Send & Receive Threads */
    thread_create(Send,10000,2,0,0);
    thread_create(Receive,10000,2,0,0);

    /* Now sit and wait for the Receive thread to finish */
    sema_wait(&main_resume);

    /* all threads are now stopped so no need for par_sema */
    printf("\nThe entire image was encoded in %7lu bits\n",total);
}
/*-------------------------------------------------------------------------*/
void Send() {
    int  done, nbytes;
    int  i,j, offset;

    for (done = 0; done < NumBlocks; done++) {

        /* Assemble next block to be processed block */
        if (Bused >= cmpheader.BlkAcross) {  /* initiate a read from the file */
            sema_wait(&par_sema);      /* wait for C RTL */
            fread(ifb,admheader.PixAcross*BlockSize,1,ADMfp);
            sema_signal(&par_sema);    /* release C RTL */
            Bused=0;
        }
        offset = Bused * BlockSize;
        for (i=0; i < BlockSize; i++) {
            for (j=0; j < BlockSize; j++) {
                y[i][j] = ifb[offset+j];
            }
            offset += admheader.PixAcross;
        }
        Bused++;

        /* Level shift */
        for (i=0;i<8;i++) for (j=0;j<8;j++) y[i][j] = y[i][j]-128;

        /* Now send the block on its way */
        nbytes = net_send(sizeof(int)*BlockSize*BlockSize,y,1);
    }
    sema_wait(&par_sema);       /* wait for C RTL */
    fclose(ADMfp);              /* closed the image file */
    sema_signal(&par_sema);     /* release C RTL */

    /* All the blocks of the image have been sent  STOP the Thread */
    thread_stop();

}
/*-------------------------------------------------------------------------*/
void Receive() {
    int nbytes, complete;
    int i, j;

    BlocksProcessed = 0;
    while (BlocksProcessed < NumBlocks) {    /* loop to recieve & process blocks */
        /* Receive a block from the Processor Farm */
        nbytes = net_receive(Y,&complete);
        /* Quantization using the uniform quantization table in quant.h */
        for (i=0;i<BlockSize;i++) for (j=0;j<BlockSize;j++)
            q[i][j] = (Y[i][j]/Q[i][j])+ (Y[i][j]>0 ? 0.5 : -0.5);
        /* do the Huffman encoding as defined in ISO/IEC 10918-1 Annex F */
        Huffman();
        BlocksProcessed++;
    }
    /* flush the buffer if there is anything in it */
    if (count > 0) {
        /* shift it left to the MSB */
        buff <<= 32-count;
        sema_wait(&par_sema);      /* wait for C RTL */
        fwrite(&buff,4,1,CMPfp);
        sema_signal(&par_sema);    /* release C RTL */
        total += count; }
```

```
    sema_wait(&par_sema);       /* wait for C RTL */
    fclose(CMPfp);              /* closed the compressed file */
    sema_signal(&par_sema);     /* release C RTL */
    sema_signal(&main_resume); /* signal the main thread to resume */
    thread_stop();              /* stop this thread */
}
/*-------------------------------------------------------------------------*/

void OpenFiles(int argc, char *argv[]) {
    /* No need to use par_sema here as other threads are not active yet */

    if (!(argc >1)) {
        printf("Usage - code filename");
        exit(1);  }
    /* get the filenames */
    strcpy(ADMfile,argv[1]);
    p = strrchr(ADMfile,92);
    if (p == NULL) strcpy(CMPfile,ADMfile); else  strcpy(CMPfile,++p);
    strcat(ADMfile,".ADM");
    strcat(CMPfile,".CMP");

    /* Try to open the ADM file */
    ADMfp = fopen(ADMfile,"rb");
    if (ADMfp == NULL) {
        printf("Error opening ADM file %s \n",ADMfile);
        exit(1);  }
    /* and now open the CMP file (the empty compressed file) */
    CMPfp = fopen(CMPfile,"wb");
    if (CMPfp == NULL) {
        printf("Error opening CMP file %s \n",CMPfile);
        exit(1);  }
    setvbuf(CMPfp, ofb, _IOFBF, 5000);
}
/*-------------------------------------------------------------------------*/
void Initialize() {
    /* No need to use par_sema here as other threads are not active yet */
    /* get the image header */
    fread(&admheader,sizeof(struct ADMhead),1,ADMfp);

    /* Now set the compressed file header and write it out */
    cmpheader.Version = ver;
    strcpy(cmpheader.Description,admheader.Description);
    cmpheader.BlkSize = BlockSize;
    cmpheader.BlkAcross = admheader.PixAcross / BlockSize;
    cmpheader.BlkDown = admheader.PixDown / BlockSize;
    cmpheader.DCBlk1 = 0;     /* for the moment  re-write the header later */
    strcpy(cmpheader.Mode,"Gray");
    fwrite(&cmpheader,sizeof(struct CMPhead),1,CMPfp);

    /* allocate the Image File Buffer */
    ifb = malloc(admheader.PixAcross*BlockSize);
    if (ifb == NULL) {
        printf("Could not Alloctae Image File Buffer\n");
        exit(1);  }
    /* set the number of blocks used so as to initiate a ifb read */
    Bused = cmpheader.BlkAcross;

    /* initially start with empty buffer */
    buff = 0;  count = 0;  total = 0;

    /* Huffman initialization stuff */
    BuildCodes();
}
/*-------------------------------------------------------------------------*/
/* Huffman Code Table initialization Routines */

void BuildCodes() {
    /* No need to use par_sema here as other threads are not active yet */

    int BITS[16], *HUFFVAL, *HUFFSIZE;
    unsigned int  *HUFFCODE;
    int i, j, k, lastk, code, si, HVTsze=0;

    printf("\n Building Huffman Code Tables\n");
    hufftab = fopen("ACtable.dat","r");
    if (hufftab == NULL) {
        printf("Error opening Huffman Code Table \n");
        exit(1);  }
    /* read the code lengths  (for luminance ) Annex K ISO/IEC 10918-1 */
    for (i=0;i<16;i++) fscanf(hufftab," %x",&BITS[i]);
```

```
    /* Now read the Huffman values associated with the code lengths */
    for (i=0;i<16;i++) HVTsze += BITS[i];

    HUFFVAL = malloc(sizeof(int)*HVTsze);
    for (i=0;i<HVTsze;i++) fscanf(hufftab," %x",&HUFFVAL[i]);

    fclose(hufftab);

    /* now follow procedures in Annex C to generate the following tables */
    /* HUFFSIZE table  - slight changes because BITS indexed from 0*/
    HUFFSIZE = malloc((sizeof(int)*HVTsze)+1);
    k = 0;  i = 1;  j = 1;
    do {
        while (j <= BITS[i-1]) {
            HUFFSIZE[k] = i;
            k++;
            j++;
        }
        i++;
        j = 1;
    } while (i <= 16 );
    HUFFSIZE[k] = 0;
    lastk = k;

    /* HUFFCODE table */
    HUFFCODE = malloc(sizeof(unsigned int)*HVTsze);
    k = 0;  code = 0;  si = HUFFSIZE[0];
    do {
        do {
            HUFFCODE[k] = code;
            code = code + 1;
            k = k + 1;
        } while (HUFFSIZE[k] == si);
        if (HUFFSIZE[k] == 0) break;
        do {
            code = code << 1;
            si = si + 1;
        } while (HUFFSIZE[k] != si);
    } while (HUFFSIZE[k] == si);

    /* Now create encoding procedure code tables  EHUFCO & EHUFSI */
    EHUFCO = malloc(sizeof(ULint32)*256);
    EHUFSI = malloc(sizeof(int)*256);
    for (i=0;i<256;i++) {
        EHUFCO[i] = 0;
        EHUFSI[i] = 0;
    }

    k = 0;
    do {
        i = HUFFVAL[k];
        EHUFSI[i] = HUFFSIZE[k];
        /* code component - align bits to the MSB end of the 32 bit int */
        EHUFCO[i] = (ULint32)(HUFFCODE[k]) << (32-EHUFSI[i]);
        k++;
    } while (k < lastk);

    /* reallocate some of the memory back */
    free(HUFFVAL);
    free(HUFFSIZE);
    free(HUFFCODE);
}
/*-------------------------------------------------------------------------*/
/*     Huffman encoding routines based on quantised block q */
/* Append assumes the code parameter has all bits aligned to the MSB */
void Append(int bits, ULint32 code) {

    int i;

    for (i=1; i<=bits; i++) {
        buff <<= 1;
        buff += (code >> 31) & 1;
        code <<= 1;
        count ++;

        if (count == 32) {              /* 32 bit integer 32 bits = 4 bytes */
            sema_wait(&par_sema);        /* wait for C RTL */
            fwrite(&buff,4,1,CMPfp);
            sema_signal(&par_sema);      /* release C RTL */
            count = 0;
            buff = 0;
```

```
            total += 32;
        )
    )
)

Csize (int coeff) {
    if (abs(coeff) == 1) return 1;
    else if (abs(coeff) <= 3) return 2;
    else if (abs(coeff) <= 7) return 3;
    else if (abs(coeff) <= 15) return 4;
    else if (abs(coeff) <= 31) return 5;
    else if (abs(coeff) <= 63) return 6;
    else if (abs(coeff) <= 127) return 7;
    else if (abs(coeff) <= 255) return 8;
    else if (abs(coeff) <= 511) return 9;
    else if (abs(coeff) <= 1023) return 10;
    else {
        sema_wait(&par_sema);     /* wait for C RTL */
        printf("problem encountered in Csize function, coeff out of range.\n");
        sema_signal(&par_sema);   /* release C RTL */
        exit(1);  }
}

void Huffman() {
    int ZZ[64], k, r, ssss, rs;
    ULint32 cde;

    /* first re-arrange quantized coefficients in Zig-Zag sequence */
    ZZ[0]=q[0][0];  ZZ[1]=q[0][1];  ZZ[2]=q[1][0];  ZZ[3]=q[2][0];
    ZZ[4]=q[1][1];  ZZ[5]=q[0][2];  ZZ[6]=q[0][3];  ZZ[7]=q[1][2];
    ZZ[8]=q[2][1];  ZZ[9]=q[3][0];  ZZ[10]=q[4][0]; ZZ[11]=q[3][1];
    ZZ[12]=q[2][2]; ZZ[13]=q[1][3]; ZZ[14]=q[0][4]; ZZ[15]=q[0][5];
    ZZ[16]=q[1][4]; ZZ[17]=q[2][3]; ZZ[18]=q[3][2]; ZZ[19]=q[4][1];
    ZZ[20]=q[5][0]; ZZ[21]=q[6][0]; ZZ[22]=q[5][1]; ZZ[23]=q[4][2];
    ZZ[24]=q[3][3]; ZZ[25]=q[2][4]; ZZ[26]=q[1][5]; ZZ[27]=q[0][6];
    ZZ[28]=q[0][7]; ZZ[29]=q[1][6]; ZZ[30]=q[2][5]; ZZ[31]=q[3][4];
    ZZ[32]=q[4][3]; ZZ[33]=q[5][2]; ZZ[34]=q[6][1]; ZZ[35]=q[7][0];
    ZZ[36]=q[7][1]; ZZ[37]=q[6][2]; ZZ[38]=q[5][3]; ZZ[39]=q[4][4];
    ZZ[40]=q[3][5]; ZZ[41]=q[2][6]; ZZ[42]=q[1][7]; ZZ[43]=q[2][7];
    ZZ[44]=q[3][6]; ZZ[45]=q[4][5]; ZZ[46]=q[5][4]; ZZ[47]=q[6][3];
    ZZ[48]=q[7][2]; ZZ[49]=q[7][3]; ZZ[50]=q[6][4]; ZZ[51]=q[5][5];
    ZZ[52]=q[4][6]; ZZ[53]=q[3][7]; ZZ[54]=q[4][7]; ZZ[55]=q[5][6];
    ZZ[56]=q[6][5]; ZZ[57]=q[7][4]; ZZ[58]=q[7][5]; ZZ[59]=q[6][6];
    ZZ[60]=q[5][7]; ZZ[61]=q[6][7]; ZZ[62]=q[7][6]; ZZ[63]=q[7][7];

    /* Encode AC cofficients - procedure from annex F ISO/IEC 10918-1 */
    k = -1;   /* index into zig-zag sequence : -1 codes DC coefficient */
    r = 0;    /* run length of zero coefficients */

    jump1:  /* jump point for goto.... I KNOW! but it was the easiest way */

    k++;
    if (ZZ[k] == 0) {
        if (k == 63) {
            Append(EHUFSI[0],EHUFCO[0]);
            goto jump2;
        }
        r++;
        goto jump1;
    }
    while (r > 15) {
        Append(EHUFSI[240],EHUFCO[240]);
        r -= 16;
    }

    /* Encode R & ZZ(k) at this point */
    ssss = Csize(ZZ[k]);
    rs = (r * 16) + ssss;
    Append(EHUFSI[rs],EHUFCO[rs]);
    if (ZZ[k] < 0) ZZ[k]--;
    cde = (ULint32)(ZZ[k]) << (32-ssss);
    Append(ssss,cde);

    r = 0;
    if (k < 63) goto jump1;

    jump2:  /* jump point for jump out of loop  */

    ;
}
/*-------------------------------------------------------------------------*/
```

B.7.2 Code for Worker task

```
/*   pv2 - parallel version 2  worker task   */

#include <math.h>
#include <float.h>
#include <stdlib.h>
#include "\masters\include\standard.h"
#include <net.h>

/* Function Prototypes */
void    DCT(double p[BlockSize]);

int      y[BlockSize][BlockSize];    /* level shifted image source block */
double   Y[BlockSize][BlockSize];    /* DCT Transformed double block */
double   X[BlockSize];

/*             Fast Discrete Cosine Transform      */

void main() {
    int i,j;
    int nbytes,complete;

    for (;;) {      /* infinite loop to recieve & process blocks */
        /* wait for a block to arrive from the Net */
        nbytes = net_receive(y,&complete);

        /* set up double matrix Y which is to be transformed / be the result */
        for (i = 0; i < BlockSize; i++) {
            for (j = 0; j < BlockSize; j++) Y[i][j] = (double)y[i][j];
        }

        /* now do the 2D-FDCT() */

        /*  first dimension - down the columns  */
        for (i = 0; i < BlockSize; i++) {
            for (j = 0; j < BlockSize; j++) {
                X[j] = Y[j][i];
            }
            DCT(X);
            for (j = 0; j < BlockSize; j++) Y[j][i] = X[j];
        }

        /*  second dimension - accross the rows */
        for (i = 0; i < BlockSize; i++) {
            for (j = 0; j < BlockSize; j++) {
                X[j] = Y[i][j];
            }
            DCT(X);
            for (j = 0; j < BlockSize; j++) Y[i][j] = X[j];
        }

        /* send processed block back to master task */
        nbytes = net_send((sizeof(double)*BlockSize*BlockSize),Y,1);

    }  /* end of infinite for loop */
}

/*-------------------------------------------------------------------------*/

/*             Fast Discrete Cosine Transform      */

void DCT(double Z[BlockSize]) {
    double a[BlockSize], b[BlockSize], c[BlockSize], d[BlockSize];

    /*  C8F Algorithm  */

    /* first butterfly loop */
    a[0] = Z[0] + Z[7];
    a[1] = Z[1] + Z[6];
    a[2] = Z[2] + Z[5];
    a[3] = Z[3] + Z[4];
    a[4] = Z[3] - Z[4];
    a[5] = Z[2] - Z[5];
    a[6] = Z[1] - Z[6];
    a[7] = Z[0] - Z[7];

    /* second butterfly loop */
```

```
    b[0] = a[0] + a[3];
    b[1] = a[1] + a[2];
    b[2] = a[1] - a[2];
    b[3] = a[0] - a[3];
    b[4] = a[4];
    b[5] = (a[6] - a[5])*cos(pi/4.0);
    b[6] = (a[6] + a[5])*cos(pi/4.0);
    b[7] = a[7];

    /* third butterfly loop */
    c[0] = (b[0] + b[1]) * cos(pi/4.0);
    c[1] = (b[0] - b[1]) * cos(pi/4.0);
    c[2] = b[2]*sin(pi/8.0) + b[3]*cos(pi/8.0);  /* found mistake here ? */
    c[3] = b[3]*cos(3.0*pi/8.0) - b[2]*sin(3.0*pi/8.0);
    c[4] = b[4] + b[5];
    c[5] = b[4] - b[5];
    c[6] = b[7] - b[6];
    c[7] = b[7] + b[6];

    /* fourth & final butterfly loop */
    d[4] = c[4]*sin(pi/16.0) + c[7]*cos(pi/16.0);
    d[5] = c[5]*sin(5.0*pi/16.0) + c[6]*cos(5.0*pi/16.0);
    d[6] = c[6]*cos(3.0*pi/16.0) - c[5]*sin(3.0*pi/16.0);
    d[7] = c[7]*cos(7.0*pi/16.0) - c[4]*sin(7.0*pi/16.0);

    /* now calculate normalized Forward Transform Coefficients */
    Z[0] = c[0]/4.0;
    Z[1] = d[4]/4.0;
    Z[2] = c[2]/4.0;
    Z[3] = d[6]/4.0;
    Z[4] = c[1]/4.0;
    Z[5] = d[5]/4.0;
    Z[6] = c[3]/4.0;
    Z[7] = d[7]/4.0;
}

/*-------------------------------------------------------------------------*/
```

B.8 Include Files

/* admv3.h */

```
/* file header for ADM files */
/* Requires a typedef statement for int32 equating it to a 32 bit number */

struct ADMhead  {  int32   Version;
           char    Description[32]; /* now ends on word boundary */
           int32   PixAcross;
           int32   PixDown;
           char    Mode[8];         /* only valid value is "Gray" */
           char    filler[76];
        };
```

/* cmpv3.h */

```
/* File Header for Compressed File */
/* Requires a typedef statement for int32 equating it to a 32 bit number */

struct CMPhead  {  int32   Version;
                   char    Description[32];  /* now ends on word boundary */
                   int32   BlkSize,          /* square blocks only */
                           BlkAcross,
                           BlkDown,
                           DCBlk1;           /* DC component of first block */
                   char    Mode[8];          /* only valid value is "Gray"  */
                   char    filler[68];
               };
```

/* standard.h */

```
/* file header for Image processing programs
   contains definitions used for standardisation between wordsizes of
   different CPU's */

/* Other constants used in the programs */
#define BlockSize  8
#define pi         3.14159
#define ver        2

/* type definition for 32-bit integer on local CPU */
typedef int int32;

/* type definition for 32 bit Unsigned integer */
typedef unsigned int ULint32;
```

/* uquant.h */

```
/* Uniform Quantization Table

   This is the Luminance Quantization Table K.1  specified in Annex K of
   document ISO/IEC 10918-1:1944    (used for 8 bit gray scale images) */

unsigned short int Q[8][8] = {
                {16, 11, 10, 16,  24,  40,  51,  61},
                {12, 12, 14, 19,  26,  58,  60,  55},
                {14, 13, 16, 24,  40,  57,  69,  56},
                {14, 17, 22, 29,  51,  87,  80,  62},
                {18, 22, 37, 56,  68, 109, 103,  77},
                {24, 35, 55, 64,  81, 104, 113,  92},
                {49, 64, 78, 87, 103, 121, 120, 101},
                {72, 92, 95, 98, 112, 100, 103,  99}
                  };
```

APPENDIX C

JAVA PARALLEL SIMULATION CODE

C.1 Introduction

The following pages in this Appendix contain the code for the Java classes used in the parallel simulation algorithm developed and discussed in Chapter 5.

C.2 Image file Header Class

```
/* admv3.java */

/* file header for ADM files  Image source file */

class Admv3 {

    int   Version;
    char  Description[] = new char[32];
    int   PixAcross;
    int   PixDown;
    char  Mode[] = new char[8];          /* only valid value is "Gray" */
    char  filler[] = new char[76];
}
```

C.3 Compressed file Header Class

```
/* cmpv3.java */

/* File Header for Compressed File */

class Cmpv3 {
    int    Version;
    char   Description[] = new char[32];  /* now ends on word boundary */
    int    BlkSize;                       /* square blocks only */
    int    BlkAcross;
    int    BlkDown;
    int    DCBlk1;                        /* DC component of first block */
    char   Mode[]   = new char[8];        /* only valid value is "Gray"  */
    char   filler[] = new char[68];
}
```

C.4 Token Object Class

```
/*  Token.java

   This class is the Token Object which is passed to each of the
   thread groups, and gives them easy access to control data in
   the simulation.  Also contains flags to aid in the avoidance
   of deadlock on synchronous objects (Vectors)
*/

class Token {

    private int blocksTransformed;
    int blocksFinished;
    private int blocksToProcess;
    private boolean critical;  // indicates if thread is in critical reigon
    private boolean time;      // scheduler wants thread to give up CPU
    int numWorkers;            // number of worker processors in simulation

    Token(int nworkers) {
        super();
        blocksTransformed = 0;
        blocksFinished = 0;
        blocksToProcess =0;
        critical = false;
        time = false;
        numWorkers = nworkers;
    }

    //-------------------------------------------------------
    public void setTotalBlocks(int x) {
        blocksToProcess = x;
    }

    public int totalBlocks() {
        return blocksToProcess;
    }

    public void addBlocks() {
        blocksTransformed++;
    }

    public int numBlocks() {
        return blocksTransformed;
    }

    //-------------------------------------------------------
    public void setCritical() {
        critical = true;
    }

    public void clearCritical() {
        critical = false;
    }

    public boolean inCritical() {
        return critical;
    }

    //-------------------------------------------------------
    public void setTime() {
        time = true;
    }

    public void clearTime() {
        time = false;
    }

    public boolean isTime() {
        return time;
    }

}
```

C.5 Worker thread Object Class

```
/*
    Dct.java

    This Thread object is the DCT worker task in the parallel
    simulation.  the FDCT is a fast version of Discrete Cosine
    Transform by Chen and Smith

    Note this containd built in delays for Processor
    communication times, and delay factors
*/

import java.util.Vector;
import Token;
import java.util.NoSuchElementException;
import java.lang.Math;


class Dct extends Thread {

    final double pi = 3.14159;
    int BlockSize;
    Vector blockQ, dctBlockQ;
    Token token;

    int      y[][];    /* level shifted image source block */
    double   Y[][];    /* DCT Transformed double block */
    double   X[];

       // DCT constructor
    Dct(int bSize, Vector bQueue, Vector tQueue, Token t, ThreadGroup group,
           String name) {

        super(group,name);
        BlockSize = bSize;
        blockQ = bQueue;
        dctBlockQ = tQueue;
        token = t;
    }

    public void run() {
        int i,j;
        int nbytes,complete;

        X = new double[BlockSize];

       /* loop to retreive & process blocks */
        while (token.numBlocks() < token.totalBlocks()) {

            // get block from the queue

            // first delay associated with receiving 8x8 4 byte per element block
            try { Thread.sleep(0,89); }   // physical transmission delay
                catch (Exception e)
               { System.out.println("Error W1");System.out.flush();} ;
            try { Thread.sleep(0,30); }   // processor end comms delay
                catch (Exception e)
               { System.out.println("Error W2");System.out.flush();} ;

            token.setCritical();
            try {
                y = (int[][]) blockQ.firstElement();
                blockQ.removeElementAt(0);}
            catch (NoSuchElementException e) {
                token.clearCritical();
                if (token.isTime()) yield();
                continue;
            }
            token.clearCritical();
            if (token.isTime()) yield();

            // create new double matrix Y
            Y = new double[BlockSize][BlockSize];
```

```
        /* set up double matrix Y which is to be transformed */
        for (i = 0; i < BlockSize; i++) {
            for (j = 0; j < BlockSize; j++) Y[i][j] = (double)y[i][j];
        }

        /* now do the 2D-FDCT() */

        /*  first dimension - down the columns  */
        for (i = 0; i < BlockSize; i++) {
            for (j = 0; j < BlockSize; j++) {
                X[j] = Y[j][i];
            }
            DCT(X);
            for (j = 0; j < BlockSize; j++) Y[j][i] = X[j];
        }

        /*  second dimension - accross the rows */
        for (i = 0; i < BlockSize; i++) {
            for (j = 0; j < BlockSize; j++) {
                X[j] = Y[i][j];
            }
            DCT(X);
            for (j = 0; j < BlockSize; j++) Y[i][j] = X[j];
        }

       // Time adjustment factor
       try { Thread.sleep(87); } catch (Exception e)
           { putil.writeln("Error 3");} ;

        // place transformed block in a special queue

        // first delay associated with sending 8x8 8 byte per element block
        try { Thread.sleep(0,59); }   // processor end comms delay
            catch (Exception e)
           { System.out.println("Error W3");System.out.flush();} ;
        try { Thread.sleep(0,176); }   // physical transmission delay
            catch (Exception e)
           { System.out.println("Error W4");System.out.flush();} ;

        token.setCritical();
        dctBlockQ.addElement(Y);
        token.addBlocks();
        token.clearCritical();
        if (token.isTime()) yield();

    }  /* end of infinite for loop */
}

/*--------------------------------------------------------------------------*/

/*             Fast Discrete Cosine Transform      */

void DCT(double Z[]) {
    double a[], b[], c[], d[];

    // allocate Arrays
    a = new double[BlockSize];
    b = new double[BlockSize];
    c = new double[BlockSize];
    d = new double[BlockSize];

    /*  C8F Algorithm  */

    /* first butterfly loop */
    a[0] = Z[0] + Z[7];
    a[1] = Z[1] + Z[6];
    a[2] = Z[2] + Z[5];
    a[3] = Z[3] + Z[4];
    a[4] = Z[3] - Z[4];
    a[5] = Z[2] - Z[5];
    a[6] = Z[1] - Z[6];
    a[7] = Z[0] - Z[7];

    /* second butterfly loop */
    b[0] = a[0] + a[3];
    b[1] = a[1] + a[2];
    b[2] = a[1] - a[2];
    b[3] = a[0] - a[3];
    b[4] = a[4];
    b[5] = (a[6] - a[5])*Math.cos(pi/4.0);
    b[6] = (a[6] + a[5])*Math.cos(pi/4.0);
```

```
        b[7] = a[7];

        /* third butterfly loop */
        c[0] = (b[0] + b[1]) * Math.cos(pi/4.0);
        c[1] = (b[0] - b[1]) * Math.cos(pi/4.0);
        /* found mistake in published code at this point */
        c[2] = b[2]*Math.sin(pi/8.0) + b[3]*Math.cos(pi/8.0);
        c[3] = b[3]*Math.cos(3.0*pi/8.0) - b[2]*Math.sin(3.0*pi/8.0);
        c[4] = b[4] + b[5];
        c[5] = b[4] - b[5];
        c[6] = b[7] - b[6];
        c[7] = b[7] + b[6];

        /* fourth & final butterfly loop */
        d[4] = c[4]*Math.sin(pi/16.0) + c[7]*Math.cos(pi/16.0);
        d[5] = c[5]*Math.sin(5.0*pi/16.0) + c[6]*Math.cos(5.0*pi/16.0);
        d[6] = c[6]*Math.cos(3.0*pi/16.0) - c[5]*Math.sin(3.0*pi/16.0);
        d[7] = c[7]*Math.cos(7.0*pi/16.0) - c[4]*Math.sin(7.0*pi/16.0);

        /* now calculate normalized Forward Transform Coefficients */
        Z[0] = c[0]/4.0;
        Z[1] = d[4]/4.0;
        Z[2] = c[2]/4.0;
        Z[3] = d[6]/4.0;
        Z[4] = c[1]/4.0;
        Z[5] = d[5]/4.0;
        Z[6] = c[3]/4.0;
        Z[7] = d[7]/4.0;
    }

}
```

C.6 Send thread Object Class of Master Processor

```
/*
   Send.java

   This is the Send thread of the master processor in the processor farm

   The purpose of this thread is to read the image file and break the image
   into 8 x 8 blocks it then places the blocks in an area where idle worker
   tasks pick them up and process them
*/

import java.util.Vector;
import Token;
import putil;
import java.io.FileInputStream;
import java.io.IOException;
import Admv3;
import Cmpv3;

public class Send extends Thread {
    // CONSTANTS
    int BlockSize;
    final int flip = 255;


    // Buffers
    byte ifb[];     // the image File buffer - Bused triggers an ifb read
    int y[][];      // the  8 x 8 image block of pixels
    Vector blockQ;  //  The raw image block Queue
    Token token;

    //Other variables
    int Bused;      // incdicates how many blocks have been used since last ifb read
    Admv3 admhead;
    Cmpv3 cmphead;
    FileInputStream image;



    Send(FileInputStream img, Cmpv3 cmp, Admv3 adm, int bSize, Vector bQueue,
            Token t, ThreadGroup group, String name) {


        super(group,name);
        BlockSize = bSize;
        blockQ = bQueue;
        cmphead = cmp;
        admhead = adm;
        image = img;
        token = t;
    }



    public void run() {
        int  done, nbytes=0;
        int  i,j, offset, NumBlocks;
        String tmp;

        //System.out.println("Block assemble and transmitter Thread running......");
        NumBlocks = cmphead.BlkAcross * cmphead.BlkDown;
        ifb  = new byte[admhead.PixAcross*BlockSize];   // allocate the ifb
        Bused = cmphead.BlkAcross;     // caused an ifg read initially

        // now assemble all blocks in the image and place in the processing Queue
        done = 0;
        while (done < NumBlocks) {

            /* Assemble next block to be processed block */
            if (Bused >= cmphead.BlkAcross) {      // initiate a read from the file
                try { image.read(ifb); }
                catch (IOException e) {System.out.println("Image read exception "+e);
                        return;};
                Bused=0;
            }
```

```
            y = new int[BlockSize][BlockSize];    // allocate the new image block
            offset = Bused * BlockSize;
            for (i=0; i < BlockSize; i++) {
                for (j=0; j < BlockSize; j++) {
                    y[i][j] = ifb[offset+j] & 255;  // because a java byte is signed
                }
                offset += admhead.PixAcross;
            }
            Bused++;


           // Time adjustment factor
           try { Thread.sleep(3); } catch (Exception e)
           { putil.writeln("Error 3");} ;

            /* Level shift */
            for (i=0;i<8;i++) for (j=0;j<8;j++) y[i][j] = y[i][j]-128;
           // Time adjustment factor
           try { Thread.sleep(2); } catch (Exception e)
           { putil.writeln("Error 3");} ;

            /* Now send the block on its way */
            try { Thread.sleep(0,30); }   // processor end comms delay
            catch (Exception e) { System.out.println("Error");System.out.flush();} ;
            token.setCritical();
            blockQ.addElement(y);
            done++;
            token.clearCritical();
            if (token.isTime()) yield();
        }
    }
}
```

C.7 Receive thread Object Class of Master Processor

```
/*
    Receive.java

   This is the Receiveend thread of the master processor in the processor farm

   This thread gets data from the network performs quantization, the JPEG
   huffman coding routines, and then stores the block

*/

import java.util.Vector;
import Token;
import putil;
import java.lang.Math;
import java.util.NoSuchElementException;
import java.io.FileOutputStream;
import java.io.DataOutputStream;
import java.io.IOException;

public class Receive extends Thread {

    // CONSTANTS
    int BlockSize;

    /* Uniform Quantization Table

    This is the Luminance Quantization Table K.1  specified in Annex K of
    document ISO/IEC 10918-1:1944    (used for 8 bit gray scale images) */

    short Q[][] = {
        {16, 11, 10, 16,  24,  40,  51,  61},
        {12, 12, 14, 19,  26,  58,  60,  55},
        {14, 13, 16, 24,  40,  57,  69,  56},
        {14, 17, 22, 29,  51,  87,  80,  62},
        {18, 22, 37, 56,  68, 109, 103,  77},
        {24, 35, 55, 64,  81, 104, 113,  92},
        {49, 64, 78, 87, 103, 121, 120, 101},
        {72, 92, 95, 98, 112, 100, 103,  99}
    };

    // Other variables
    int     EHUFSI[];   // huffman encoding tables
    int     EHUFCO[];
    FileOutputStream cimage;
    Token token;
    Vector dctBlockQ;  //  The Transformed Block Queue
    DataOutputStream dos;
    int buff = 0;              // the 32 bit buffer used to write huffman codes
    int count = 0;             // counts the number of bits used in buff
    int BlocksProcessed;

    long total = 0;     // counts the total number of bits in encoding
    double   Y[][];     /* DCT Transformed double block */
    int      q[][];     // Quantized block

    Receive(FileOutputStream cimg, int bSize, Vector dctQueue, Token t,
            int hctsi[], int hctco[], ThreadGroup group, String name) {

        super(group,name);
        EHUFSI = hctsi;
        EHUFCO = hctco;
        BlockSize = bSize;
        dctBlockQ = dctQueue;
        cimage = cimg;
        token = t;
        dos = new DataOutputStream(cimage);
    }

    public void run() {

        int nbytes, complete, numBlocks;
        int i, j;

        count = 0;
```

```
    numBlocks = token.totalBlocks();

   /* loop to recieve & process blocks */
    while (token.blocksFinished < numBlocks) {

        /* Receive a block from the Processor Farm */
        try { Thread.sleep(0,59); }   // processor end comms delay
        catch (Exception e) { System.out.println("Error R1");
             System.out.flush();} ;
        token.setCritical();
        try {Y = (double[][]) dctBlockQ.firstElement();
            dctBlockQ.removeElementAt(0);}
        catch (NoSuchElementException e) {
            token.clearCritical();
            if (token.isTime()) yield();
            continue;}
        token.clearCritical();
        if (token.isTime()) yield();

        /* Quantization using the uniform quantization table in quant.h */
        q = new int[BlockSize][BlockSize];
        for (i=0;i<BlockSize;i++)
          for (j=0;j<BlockSize;j++)
             q[i][j] = (int) ((Y[i][j]/Q[i][j])+ (Y[i][j]>0 ? 0.5 : -0.5));

       // Time adjustment factor
       try { Thread.sleep(5); }
       catch (Exception e) { putil.writeln("Error 3");} ;

        /* do the Huffman encoding as defined in ISO/IEC 10918-1 Annex F */
        Huffman();
       // Time adjustment factor  includes huffman and store block stuff
       try { Thread.sleep(3); }
       catch (Exception e) { putil.writeln("Error 3");} ;

        token.blocksFinished++;

    }
    /* flush the buffer if there is anything in it */
    if (count > 0) {
        /* shift it left to the MSB */
        buff <<= 32-count;
        try {dos.writeInt(buff);}
       catch (IOException e) {System.out.println("write error");}
        total += count;
    }

   /* closed the compressed file */
    try {dos.close();} catch (IOException e) {}
}

//-------------------------------------------------------------------------

/*     Huffman encoding routines based on quantised block q */

/* Append assumes the code parameter has all bits aligned to the MSB */
void Append(int bits, int code) {

    int i;

    for (i=1; i<=bits; i++) {
        buff <<= 1;
        buff += (code >> 31) & 1;
        code <<= 1;
        count ++;

        if (count == 32) {     /* 32 bit integer 32 bits = 4 bytes */
            try {dos.writeInt(buff);}
           catch (IOException e) {System.out.println("write error");}
            count = 0;
            buff = 0;
            total += 32;
        }
    }
}

int Csize (int coeff) {
    if (Math.abs(coeff) == 1)
        return 1;
    else if (Math.abs(coeff) <= 3)
        return 2;
```

```
        else if (Math.abs(coeff) <= 7)
            return 3;
        else if (Math.abs(coeff) <= 15)
            return 4;
        else if (Math.abs(coeff) <= 31)
            return 5;
        else if (Math.abs(coeff) <= 63)
            return 6;
        else if (Math.abs(coeff) <= 127)
            return 7;
        else if (Math.abs(coeff) <= 255)
            return 8;
        else if (Math.abs(coeff) <= 511)
            return 9;
        else if (Math.abs(coeff) <= 1023)
            return 10;
        else {
            System.out.println("problem encountered in Csize function,
                                coeff out of range.\n");
            System.exit(1);
        }
        return 0;    // this should never be reached, but must be here
    }

    void Huffman() {
        int ZZ[], k, r, ssss, rs;
        int cde;
        ZZ = new int[64];

        /* first re-arrange quantized coefficients in Zig-Zag sequence */
        ZZ[0]=q[0][0];  ZZ[1]=q[0][1];  ZZ[2]=q[1][0];  ZZ[3]=q[2][0];
        ZZ[4]=q[1][1];  ZZ[5]=q[0][2];  ZZ[6]=q[0][3];  ZZ[7]=q[1][2];
        ZZ[8]=q[2][1];  ZZ[9]=q[3][0];  ZZ[10]=q[4][0]; ZZ[11]=q[3][1];
        ZZ[12]=q[2][2]; ZZ[13]=q[1][3]; ZZ[14]=q[0][4]; ZZ[15]=q[0][5];
        ZZ[16]=q[1][4]; ZZ[17]=q[2][3]; ZZ[18]=q[3][2]; ZZ[19]=q[4][1];
        ZZ[20]=q[5][0]; ZZ[21]=q[6][0]; ZZ[22]=q[5][1]; ZZ[23]=q[4][2];
        ZZ[24]=q[3][3]; ZZ[25]=q[2][4]; ZZ[26]=q[1][5]; ZZ[27]=q[0][6];
        ZZ[28]=q[0][7]; ZZ[29]=q[1][6]; ZZ[30]=q[2][5]; ZZ[31]=q[3][4];
        ZZ[32]=q[4][3]; ZZ[33]=q[5][2]; ZZ[34]=q[6][1]; ZZ[35]=q[7][0];
        ZZ[36]=q[7][1]; ZZ[37]=q[6][2]; ZZ[38]=q[5][3]; ZZ[39]=q[4][4];
        ZZ[40]=q[3][5]; ZZ[41]=q[2][6]; ZZ[42]=q[1][7]; ZZ[43]=q[2][7];
        ZZ[44]=q[3][6]; ZZ[45]=q[4][5]; ZZ[46]=q[5][4]; ZZ[47]=q[6][3];
        ZZ[48]=q[7][2]; ZZ[49]=q[7][3]; ZZ[50]=q[6][4]; ZZ[51]=q[5][5];
        ZZ[52]=q[4][6]; ZZ[53]=q[3][7]; ZZ[54]=q[4][7]; ZZ[55]=q[5][6];
        ZZ[56]=q[6][5]; ZZ[57]=q[7][4]; ZZ[58]=q[7][5]; ZZ[59]=q[6][6];
        ZZ[60]=q[5][7]; ZZ[61]=q[6][7]; ZZ[62]=q[7][6]; ZZ[63]=q[7][7];

        /* Encode AC cofficients - procedure from annex F ISO/IEC 10918-1 */
        k = -1;    /* index into zig-zag sequence : -1 codes DC coefficient */
        r = 0;     /* run length of zero coefficients */

        while (true) {    // jump1:

            k++;
            if (ZZ[k] == 0) {
                if (k == 63) {
                    Append(EHUFSI[0],EHUFCO[0]);
                    break;      // goto jump2
                }
                r++;
                continue;        // goto jump1
            }
            while (r > 15) {
                Append(EHUFSI[240],EHUFCO[240]);
                r -= 16;
            }
            /* Encode R & ZZ(k) at this point */
            ssss = Csize(ZZ[k]);
            rs = (r * 16) + ssss;
            Append(EHUFSI[rs],EHUFCO[rs]);
            if (ZZ[k] < 0) ZZ[k]--;
            cde = (ZZ[k]) << (32-ssss);
            Append(ssss,cde);

            r = 0;
            if (k >= 63) break;    // if k<63 goto jump1
        }
        //  jump2:
    }

}
```

C.8 Controlling Thread and Simulation algorithm Class

```
/*
     Master.java

     This is the controlling thread of the simulation.

     This main program, opens the image file, reads the header andsets up the
     header of the compressed file.

     This program the creates the thread groups representing the processors,
     assigns the threads to the thread groups, then performs the multi-processor
     simulation.  Simulating the processor farm paradigm of the parallel
     JPEG algorithm.
*/

import Token;
import java.lang.Thread;
import java.util.Vector;
import putil;
import java.io.*;

// input and output file headers
import Admv3;
import Cmpv3;


public class Master {

    // any Constants
    static final int BlockSize = 8;
    static final String hufftab = "ACtable.dat";

    static final int numProcessors = 8; // Number of processors in simulation
    static final int timeSlice = 100;  //  Time slice that each Processor gets

    /* variables for Huffman encoding procedures */
    static int     EHUFSI[];   // huffman encoding tables
    static int     EHUFCO[];

    // FILE OBJECTS
    static FileInputStream  image;
    static FileOutputStream cimage;

    static Cmpv3  cmpheader = new Cmpv3();  // make an output file header
    static Admv3  admheader = new Admv3();  // make an input file header

    /* variables used for communication between Receive & Send threads */
    static Vector BlockQueue = new Vector();    // raw image block queue
    static Vector dctBlockQueue = new Vector(); // Queue of DCT transformed blocks

    /*-------------------------------------------------------------------------*/
    public static void main(String args[]) {


        int j,k;
        String str;
        char c;

        OpenFiles();
        BuildCodes();
        System.out.println("SCHEDULING ..................");
        Scheduler();
    }

    /* ------------------------------------------------------------------------
    This is the main Scheduling algorithm, the multi-processor system is
    simulated by

    ThreadGroups  simulating processors
    Threads       simulating processes

   A Vector is used to hold the ThreadGroups, thus the Vector has one entry for
  each processor.  Processors are P0 ... Pi  where i is determined by the
  constant numProcessors.

    the Threads allocated are        Send  --> P0
                                     Receive --> P0
```

```
 A copy of Dct to every other processor  Dct --> P1 .. Pn

Each ThreadGroup is then Time Sliced by a number of milliseconds setermined
by the constant  timeSlice

 */

 public static void Scheduler() {
     String str;
     char c;
     Send sender;            // Send thread object
     Receive receiver;       // Receive thread object
     Dct  dct;               // DCT (Worker) thread object
     Vector processorQueue = new Vector();  // Holds all Simulated Processors
     Vector processQueue = new Vector();
     Token token;

     ThreadGroup   schedGroup;  // Thread group containing this scheduler
     Thread        schedThread; // The scheduler thread - this thread

     // general purpose variables
     int         i, j, activeThreads, activeProcessors;
     ThreadGroup processor;
     Thread[]    processList;
     Thread      process;
     int round;
     DataOutputStream fp= null;
     //-----------------------------------------------------------------

     // Initialize scheduler thread and Group
     schedThread = Thread.currentThread();
     schedThread.setName("masterThread");
     schedGroup = schedThread.getThreadGroup();
     schedThread.setPriority(Thread.MAX_PRIORITY);  // priority of scheduler Thread

     // Create the Token
     token = new Token(numProcessors-1);
     token.setTotalBlocks(cmpheader.BlkAcross * cmpheader.BlkDown);

     // now create the simulated Processors
     System.out.println("\nCreating "+numProcessors+" simulated processors...\n");
     for (i = 0; i < numProcessors; i++) {
        // create processor with name P0 .. Pn
         processor = new ThreadGroup("P"+i);
         processor.setMaxPriority(Thread.NORM_PRIORITY); // set its priority
         processorQueue.addElement(processor);   // add to queue of processors
     }


     // now add Threads (processes) to the Thread Groups (simulated Processors)

     // First the processor P0  - Master processor
     processor = (ThreadGroup) processorQueue.elementAt(0); // get processor P0

     // create and add new Send thread
     sender = new Send(image, cmpheader, admheader, BlockSize, BlockQueue,
                       token, processor, "sender");
     // Now Start the new Thread but suspend it so it doesnt get any processor time
     sender.start(); sender.suspend();

     // create and add new Receive thread
     receiver = new Receive(cimage, BlockSize, dctBlockQueue, token,
                            EHUFSI, EHUFCO, processor, "receiver");
     receiver.start(); receiver.suspend();

     // Now all other processors P1 ... Pn - Worker processors
     for (i=1; i<processorQueue.size(); i++) {
         processor = (ThreadGroup) processorQueue.elementAt(i);
         dct = new Dct(BlockSize, BlockQueue, dctBlockQueue, token,
                       processor, "dctTask");
         // Start the new Thread and suspend it so it doesnt get processor time
         dct.start(); dct.suspend();
     }

     // now list all process ThreadGroups
     schedGroup.list();

     System.out.println("Beginning Simulation.......with "+token.numWorkers+
                       " Worker processors");
     try { Thread.sleep(5000); }
     catch (Exception e) { putil.writeln("Error 3");} ;
```

```
    // now schedule all Processors (Thread Groups)
    round = 0;

  //  continue simulating until all processes on all processors have finished
    while (true) {
        activeProcessors = processorQueue.size();
        if (activeProcessors == 0) break;  // we have finished
        i = 0;
        while (i < activeProcessors) {
            processor = (ThreadGroup) processorQueue.elementAt(i);
            //enumerate all active threads in this group
            activeThreads = processor.activeCount();
            // if no active threads then remove from the ThreadGroup list
            if (activeThreads == 0) {
                processorQueue.removeElementAt(i);
                activeProcessors--;
                System.out.println("Processor "+processor.getName()+
                                   " removed from processor queue");
                continue;
            }
            // create a big enough array for the processes and enumerate
            // from the ThreadGroup
            processList = new Thread[activeThreads];

            processor.enumerate(processList);
            //now give each process in this Processor a share of the time slice
            for (j = 0; j < activeThreads; j++) {
                process = processList[j];
                // pause for garbage collector
                try { Thread.sleep(25); }
                catch (Exception e) { System.out.println("Error 0");
                                      System.out.flush();} ;
                process.resume();
                // put control thread to sleep
                try { Thread.sleep(timeSlice/activeThreads); }
                catch (Exception e) { System.out.println("Error 1");
                                      System.out.flush();} ;
                // check for critical section
                while (token.inCritical()) {
                    System.out.println("Critical situation Detected");
                    token.setTime();
                    Thread.yield();
                }
                if (token.isTime()) {
                    token.clearTime();
                    System.out.println("Critical resolved");
                }
                process.suspend();
            }
            i++;   // now go on to the next processor
        }
        round++;

        // pause for garbage collector
        try { Thread.sleep(1000); }
        catch (Exception e) { System.out.println("Error 2");} ;
    }

    try {fp.close();}  catch (IOException e) {};
}

/*---------------------------------------------------------------------*/

static void OpenFiles() {

    ByteArrayInputStream bis;
    ByteArrayOutputStream bos;
    DataInputStream dis;
    DataOutputStream dos;

    int i;
    byte admH[] = new byte[128];   // ADM header itself as a byte array

    System.out.println("Reading Image header !");

    // open the image file
    try {image = new FileInputStream("demo4.adm"); }
    catch (IOException e) { System.out.println("Cannot open image file");
        System.exit(0); };
```

```
    // read the ADM header
    try { image.read(admH); } catch (IOException e) {return;};
    bis = new ByteArrayInputStream(admH);
    dis = new DataInputStream(bis);
    try {
        admheader.Version = reverse(dis.readInt());
        for (i = 0; i < 32; i++)
            { admheader.Description[i] = (char) dis.readByte(); }
        admheader.PixAcross = reverse(dis.readInt());
        admheader.PixDown = reverse(dis.readInt());
        for (i=0; i< 8; i++) admheader.Mode[i] = (char) dis.readByte();
    }
    catch (IOException e) {return;};

    System.out.println("Constructing Compressed File Header !");
    // Now set up the CMP header
    cmpheader.Version = admheader.Version;
    for (i=0; i<admheader.Description.length; i++)
       cmpheader.Description[i] = admheader.Description[i];
    cmpheader.BlkSize = BlockSize;
    cmpheader.BlkAcross = admheader.PixAcross / BlockSize;
    cmpheader.BlkDown   = admheader.PixDown / BlockSize;
    cmpheader.DCBlk1 = 0;
    for (i=0; i<admheader.Mode.length; i++)
       cmpheader.Mode[i] = admheader.Mode[i];

    // open the compressed file
    try {cimage = new FileOutputStream("demo4.cmp"); }
    catch (IOException e) { System.out.println("Cannot open compressed file");
                            System.exit(0); };
    // write cmpheader out to compressed file
    bos = new ByteArrayOutputStream();
    dos = new DataOutputStream(bos);
    try {
        dos.writeInt(cmpheader.Version);
        for (i=0; i<32; i++) dos.write((int) cmpheader.Description[i]);
        dos.writeInt(cmpheader.BlkSize);
        dos.writeInt(cmpheader.BlkAcross);
        dos.writeInt(cmpheader.BlkDown);
        dos.writeInt(cmpheader.DCBlk1);
        for (i=0; i<8; i++) dos.write((int) cmpheader.Mode[i]);
        for (i=0; i<68; i++) dos.write((int) cmpheader.filler[i]);
        // now write the header
        cimage.write(bos.toByteArray()); )
    catch (IOException e) { };

    // close files
    /*      try { image.close();
    cimage.close(); }
    catch (IOException e) {};
    */
}

/*-------------------------------------------------------------------*/

// used to reverse the orser of bytes in an integer -
// Java expects ints stored MSB first
static int reverse(int i) {
    int inta, intb, intc, intd, result;

    inta = (i >> 24);
    intb = (i << 8);  intb = (intb >> 24);
    intc = (i << 16); intc = (intc >> 24);
    intd = (i << 24); intd = (intd >> 24);
    return (intd << 24) + (intc << 16) + (intb << 8) + inta;
}

/*-------------------------------------------------------------------*/
/* Huffman Code Table initialization Routines */

static void BuildCodes() {

    FileInputStream table = null;
    StreamTokenizer tis;
    Reader r;

    int BITS[] = new int[16];
    int  HUFFVAL[], HUFFSIZE[], HUFFCODE[];
    int i, j, k, lastk, code, si, HVTsze=0;

    System.out.println("\n Building Huffman Code Tables\n");
```

```
// open the huffman table file
try { table = new FileInputStream(hufftab); }
catch (IOException e) { System.out.println("Cannot open huffman table file");
    System.exit(0); };

r = new BufferedReader(new InputStreamReader(table));

tis = new StreamTokenizer(r);  // create a token input stream from table
tis.resetSyntax();
tis.wordChars('a', 'z');
tis.wordChars('A', 'Z');
tis.wordChars('0', '9');
tis.wordChars(128 + 32, 255);
tis.whitespaceChars(0, ' ');
tis.eolIsSignificant(false);
tis.lowerCaseMode(false);


/* read the code lengths  (for luminance ) Annex K ISO/IEC 10918-1 */
try {
    for (i=0;i<16;i++) {
        tis.nextToken();
        BITS[i] = putil.hexStrToInt(tis.sval);
    }
}
catch (IOException e) {
    System.out.println("Trouble reading huffman code tables");
    System.exit(0); };

/* Now read the Huffman values associated with the code lengths */
for (i=0;i<16;i++) HVTsze += BITS[i];

HUFFVAL = new int[HVTsze];
try {
    for (i=0; i < HVTsze; i++) {
        tis.nextToken();
        HUFFVAL[i] = putil.hexStrToInt(tis.sval);
    }
}
catch (IOException e) {
    System.out.println("Trouble reading huffman code tables");
    System.exit(0); };

try {table.close(); } catch (IOException e) {};

/* now follow procedures in Annex C to generate the following tables */

/* HUFFSIZE table  - slight changes because BITS indexed from 0*/
HUFFSIZE = new int[HVTsze + 1];
k = 0;  i = 1;  j = 1;
do {
    while (j <= BITS[i-1]) {
        HUFFSIZE[k] = i;
        k++;
        j++;
    }
    i++;
    j = 1;
} while (i <= 16 );
HUFFSIZE[k] = 0;
lastk = k;

/* HUFFCODE table */
HUFFCODE = new int[HVTsze];
k = 0;  code = 0;  si = HUFFSIZE[0];
do {
    do {
        HUFFCODE[k] = code;
        code = code + 1;
        k = k + 1;
    } while (HUFFSIZE[k] == si);
    if (HUFFSIZE[k] == 0) break;
    do {
        code = code << 1;
        si = si + 1;
    } while (HUFFSIZE[k] != si);
} while (HUFFSIZE[k] == si);

/* Now create encoding procedure code tables  EHUFCO & EHUFSI */
EHUFCO = new int[256];
```

```
        EHUFSI = new int[256];
        for (i = 0; i < 256; i++) {
            EHUFCO[i] = 0;
            EHUFSI[i] = 0;
        }

        k = 0;
        do {
            i = HUFFVAL[k];
            EHUFSI[i] = HUFFSIZE[k];
            /* code component - align bits to the MSB end of the 32 bit int */
            EHUFCO[i] = (HUFFCODE[k]) << (32-EHUFSI[i]);
            k++;
        } while (k < lastk);

        /* reallocate some of the memory back */
        HUFFVAL = null;
        HUFFSIZE = null;
        HUFFCODE = null;

    }

    /*--------------------------------------------------------------------*/


}  // end of class definition
```