PARALLEL TECHNIQUES FOR REAL-TIME SIMULATION OF VIDEOPHONE IMAGE COMPRESSION ALGORITHMS

John A. Elliott

A thesis submitted for the degree of Doctor of Philosophy

1992
Abstract

Until recently, videophony and videoconferencing have not been commercially successful due to the lack of standards in the image compression algorithms employed. In 1990, the CCITT released a draft of such an algorithm for videophony, known as recommendation H.261. This algorithm took several years to develop, and already several superior algorithms are known. Tools are required to aid the development stage of algorithm design. The commercial success of videophones relies upon standard algorithms and the availability of a ubiquitous digital switched network. The latter has been provided in the form of the CCITT ISDN standard.

This project investigates parallel techniques for the real-time simulation of videophone image compression algorithms. The simulations are flexible in the sense that the algorithm being simulated can be changed with little effort. Also, they provide a high degree of algorithm visualisation — i.e. video output from any of the algorithm can be easily viewed.

In this thesis a survey of past and present videophone image compression algorithms is presented and the H.261 algorithm is described in detail.

The complexity of current algorithms dictates that they cannot be simulated in real-time using conventional sequential computers. Affordable supercomputer power is now available in the form of MIMD distributed memory parallel computers. Such machines can be used to develop complex image processing algorithms in software.

The use of parallel computers is in its infancy. It is not clear how the processor networks should be interconnected, what languages should be used, how processes should be mapped to processors and how communications should be managed. The first simulations presented are written in occam with a topology independent routing harness, Tiny, and run on a transputer-based Meiko Computing Surface. ‘Compact graph’ processor network topologies are employed to reduce communications overheads. These simulations are compared with related work. In so doing, various topologies are analysed using measures from graph theory.

Amdahl’s Law governs the maximum possible speedup achievable by a parallel system. The simulation results suggest that real-time simulation is not possible using the present system, though an order of magnitude speedup on previous work was gained.

Further simulations are presented using the C programming language with Meiko’s CS Tools parallelisation toolset. Comparisons are drawn between the simulations and the communications harnesses used.
Declaration of originality

This thesis and the work reported herein was composed and originated entirely by the author.

The translation of some of the programs written in occam/Tiny into C/CS Tools described in Chapter 6 was assisted by Christopher Cubiss, a fellow member of the Signal Processing Group, Department of Electrical Engineering, in the University of Edinburgh.

John A. Elliott
Acknowledgements

Thanks are due to many from whom I have benefited over the last three years. Especially:

• Keith Manning and Peter Grant for their continuous support, advice and discussions. Also for always maintaining an open door policy.

• Thanks to all who read through and checked the manuscript—Peter Grant, Tony Walton, Jackie, Edward McDonnell and Graham Sexton. Any remaining errors are mine alone.

• Graham Sexton and Mark Beaumont for industrial supervision. Special thanks to Graham for many insights to the mapping problems addressed in this thesis, providing early drafts of the Reference Model and his general enthusiasm.

• SERC for funding my travel to Utah, British Telecom Labs for allowing me access to the CCITT image sequences and both for providing funding in the form of a Co-operative Award in Science and Engineering (CASE).

• Lyndon Clarke for providing Tiny which drastically affected the direction of this research.
# Contents

List of figures xi  
List of tables xii  
Glossary xiii

## 1 Introduction  
1.1 Emerging communications standards .......................................................... 3  
1.1.1 Narrowband ISDN .............................................................. 4  
1.1.2 Broadband ISDN ........................................................... 5  
1.2 Videophony ................................................................. 6  
1.3 Thesis synopsis ............................................................. 7

## 2 Motion video codecs  
2.1 Introduction ................................................................... 10  
2.2 A brief history of videophony ................................................................. 11  
2.2.1 Early trial systems ............................................................... 11  
2.2.2 European standards .............................................................. 12  
2.2.3 International standards ......................................................... 12  
2.3 Image coding overview ............................................................... 13  
2.3.1 Image quality .................................................................. 14  
2.3.2 Methods of image coding ......................................................... 14  
  Pulse-code-modulation ............................................................... 15  
  Entropy coding: variable length codes ........................................ 16  
  Predictive coding: DPCM ............................................................... 16  
  Frame replenishment ................................................................. 17  
  Contour and run-length coding .................................................... 17  
  Block truncation coding .............................................................. 17  
  Transform coding .................................................................. 18  
  Adaptive algorithms ................................................................. 19  
  TC/DPCM hybrid coding ............................................................. 20  
2.4 Video format ..................................................................... 20  
2.4.1 Colour video signals ............................................................. 21  
2.4.2 Common intermediate format ................................................. 22
## CONTENTS

2.5 CCITT motion video codec standard H.261 .................................. 23
  2.5.1 Overview of the algorithm ............................................... 24
  2.5.2 Reference models ........................................................... 25
  2.5.3 Picture hierarchical structure ........................................... 26
  2.5.4 Interframe DPCM and conditional replenishment ....................... 26
  2.5.5 2-D transform .............................................................. 27
  2.5.6 Classification .............................................................. 28
  2.5.7 Quantisation .............................................................. 29
    Variable thresholding ....................................................... 29
    Scalar quantisation .......................................................... 30
  2.5.8 Variable-length coding ................................................... 31
  2.5.9 Motion compensation ..................................................... 32
  2.5.10 Loop filter .............................................................. 33

2.6 Future motion video codecs ..................................................... 34
  2.6.1 Model-based coding ........................................................ 35
  2.6.2 Fractal coding .............................................................. 37
  2.6.3 Segmentation .............................................................. 37

2.7 Implementation of video codecs ................................................ 38
  2.7.1 Application specific integrated circuits ................................ 38
    Inmos A100 DSP range ......................................................... 39
    Discrete cosine transform .................................................. 39
    Motion compensation .......................................................... 40
  2.7.2 Digital signal processors ............................................... 40
  2.7.3 Flexibility ............................................................... 41

2.8 Summary ................................................................. 41

3 Parallel computer architectures .................................................. 43
  3.1 Introduction ................................................................. 43
  3.2 A brief history of computer architectures ................................ 44
  3.3 Parallel architecture issues ............................................... 45
    3.3.1 SIMD parallelism ......................................................... 46
    3.3.2 MIMD concurrency ........................................................ 48
  3.4 The inmos transputer ........................................................ 50
    3.4.1 The transputer family .................................................. 50
    3.4.2 T800: the floating point transputer ................................ 50
  3.5 Performance measurement ................................................... 52
    3.5.1 T9000: the new generation ............................................. 53
  3.6 The Meiko Computing Surface ............................................... 55
    3.6.1 The supervisor bus ...................................................... 56
    3.6.2 Virtual computing surfaces .......................................... 57
    3.6.3 New directions .......................................................... 58
  3.7 The Edinburgh Concurrent Supercomputer .................................... 58
    3.7.1 Graphics domains ........................................................ 59
  3.8 Parallel architectures for signal processing ............................... 60
CONTENTS

6.5.1 Analysis ............................................ 142
6.5.2 Wiring file production .............................. 144

6.6 Visualisation of intermediate results of the algorithm .... 145
6.6.1 Occam routines for visualisation .................. 145
   Histogram equalisation ............................... 146
   Enhancing transform space ........................... 147
   Image display ......................................... 148
   Colour-space conversion ............................. 148
   Real-time display ................................... 149

6.6.2 Examples of output ................................. 149
   Still frame DCT ....................................... 149
   Motion estimation .................................... 150
   Frame differencing: DPCM ............................ 153
   Cosine transform space ............................... 153
   Low pass filter ....................................... 155
   Transmission buffer overflow ....................... 155

6.7 Performance evaluation ............................... 156
6.7.1 Task buffering at the worker PEs .................. 157
6.7.2 H.261 simulations: occam/Tiny vs. C/CS Tools .... 157
6.7.3 Parallel program efficiency ......................... 158

6.8 Simulations from the viewpoint of the user ............ 159

6.9 Summary and conclusions ............................. 161

7 Summary and conclusions .................................. 162
7.1 Thesis summary ....................................... 163
7.2 Thesis conclusions ................................... 164
   7.2.1 Achievements ................................... 164
   7.2.2 Comparisons with related work ................ 166
   7.2.3 Limitations and suggestions for future work .... 167

References .................................................... 170

A Original publications .................................... 184
### List of figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Simplified block diagram of the H.120 algorithm.</td>
<td>13</td>
</tr>
<tr>
<td>2.2</td>
<td>Quarter common intermediate format frame.</td>
<td>23</td>
</tr>
<tr>
<td>2.3</td>
<td>Block diagram of the H.261 motion video coder.</td>
<td>25</td>
</tr>
<tr>
<td>2.4</td>
<td>Picture hierarchical structure.</td>
<td>26</td>
</tr>
<tr>
<td>2.5</td>
<td>Zig-zag scanning and 2-D variable-length coding of a block.</td>
<td>28</td>
</tr>
<tr>
<td>2.6</td>
<td>Uniform quantiser output for step-size, g.</td>
<td>30</td>
</tr>
<tr>
<td>2.7</td>
<td>Motion estimation between blocks in successive frames with object movement.</td>
<td>32</td>
</tr>
<tr>
<td>2.8</td>
<td>Application of the smoothing filter to a 8 x 8 pixel block. Different convolution masks are applied according to the position of the pixel within the block.</td>
<td>34</td>
</tr>
<tr>
<td>2.9</td>
<td>Conceptual diagram of a model-based coding system.</td>
<td>36</td>
</tr>
<tr>
<td>3.1</td>
<td>The AMT DAP architecture</td>
<td>47</td>
</tr>
<tr>
<td>3.2</td>
<td>Inmos IMS T800 architecture</td>
<td>51</td>
</tr>
<tr>
<td>3.3</td>
<td>Meiko Computing Surface architecture</td>
<td>57</td>
</tr>
<tr>
<td>4.1</td>
<td>Solution hierarchy.</td>
<td>63</td>
</tr>
<tr>
<td>4.2</td>
<td>Decompositions of various grain sizes.</td>
<td>67</td>
</tr>
<tr>
<td>4.3</td>
<td>An example topology and its equivalent graph.</td>
<td>76</td>
</tr>
<tr>
<td>4.4</td>
<td>Conceptual model of the task farm.</td>
<td>82</td>
</tr>
<tr>
<td>4.5</td>
<td>Example of algorithmic decomposition.</td>
<td>83</td>
</tr>
<tr>
<td>5.1</td>
<td>Example of occam layout</td>
<td>91</td>
</tr>
<tr>
<td>5.2</td>
<td>Byte file-reading processes.</td>
<td>96</td>
</tr>
<tr>
<td>5.3</td>
<td>Image reading processes.</td>
<td>97</td>
</tr>
<tr>
<td>5.4</td>
<td>Signal flow graph for the fast DCT algorithm used in the simulations. ( s(.) ) represents the sine and ( c(.) ) the cosine function. Notice that the results, ( Y(k) ), appear in 'bit-reversed' order.</td>
<td>98</td>
</tr>
<tr>
<td>5.5</td>
<td>Classification process.</td>
<td>98</td>
</tr>
<tr>
<td>5.6</td>
<td>Variable length coding processes</td>
<td>100</td>
</tr>
<tr>
<td>5.7</td>
<td>Search order for motion estimation centres</td>
<td>101</td>
</tr>
<tr>
<td>5.8</td>
<td>Motion estimation details (continued over...)</td>
<td>102</td>
</tr>
<tr>
<td>5.9</td>
<td>Motion estimation details continued.</td>
<td>103</td>
</tr>
<tr>
<td>5.9</td>
<td>Low pass filter process</td>
<td>104</td>
</tr>
<tr>
<td>FIGURE</td>
<td>DESCRIPTION</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>5.10</td>
<td>Pie chart of H.261 function execution times.</td>
<td></td>
</tr>
<tr>
<td>5.11</td>
<td>Histogram of H.261 timing profile for load balancing by inspection.</td>
<td></td>
</tr>
<tr>
<td>5.12</td>
<td>The chosen process structure for the simulations.</td>
<td></td>
</tr>
<tr>
<td>5.13</td>
<td>The master process structure.</td>
<td></td>
</tr>
<tr>
<td>5.14</td>
<td>The round robin queue process.</td>
<td></td>
</tr>
<tr>
<td>5.15</td>
<td>Chain topology. 'M' represents the master, 1 to n represent the workers.</td>
<td></td>
</tr>
<tr>
<td>5.16</td>
<td>Tertiary tree topology.</td>
<td></td>
</tr>
<tr>
<td>5.17</td>
<td>Examples of the three graph types for a total of nine PEs.</td>
<td></td>
</tr>
<tr>
<td>5.18</td>
<td>The inter-connection distance matrix with only the Hamiltonian ring entries, where asterisks represent uninitialised values.</td>
<td></td>
</tr>
<tr>
<td>5.19</td>
<td>Pseudo-code for the Floyd-Warshall algorithm.</td>
<td></td>
</tr>
<tr>
<td>5.20</td>
<td>The output of the graph analysis program showing the complete distance matrix for a random graph.</td>
<td></td>
</tr>
<tr>
<td>5.21</td>
<td>The ECS domain wiring file for the above random graph.</td>
<td></td>
</tr>
<tr>
<td>5.22</td>
<td>Functional decomposition of H.261 by Sexton at BTRL.</td>
<td></td>
</tr>
<tr>
<td>6.1</td>
<td>Amdahl's Law.</td>
<td></td>
</tr>
<tr>
<td>6.2</td>
<td>The occam timing process.</td>
<td></td>
</tr>
<tr>
<td>6.3</td>
<td>Chain topology. The master process is represented by 'M', the chain of workers is PEs 1 to n.</td>
<td></td>
</tr>
<tr>
<td>6.4</td>
<td>Process running on a main-node of the chain.</td>
<td></td>
</tr>
<tr>
<td>6.5</td>
<td>Handshaking buffer process.</td>
<td></td>
</tr>
<tr>
<td>6.6</td>
<td>The merging buffer process.</td>
<td></td>
</tr>
<tr>
<td>6.7</td>
<td>The main-node process structure.</td>
<td></td>
</tr>
<tr>
<td>6.8</td>
<td>The forwarding buffer process.</td>
<td></td>
</tr>
<tr>
<td>6.9</td>
<td>The end-node process.</td>
<td></td>
</tr>
<tr>
<td>6.10</td>
<td>Timings for DCT/IDCT simulation on a QCIF frame for chain and random Hamiltonian topologies.</td>
<td></td>
</tr>
<tr>
<td>6.11</td>
<td>Mean inter-processor distance for various processor graphs.</td>
<td></td>
</tr>
<tr>
<td>6.12</td>
<td>Timings for the parallel H.261 simulation (without motion compensation) using a) occam &amp; Tiny and b) 'C' &amp; CS Tools.</td>
<td></td>
</tr>
<tr>
<td>6.13</td>
<td>The effect on speedup of all worker PEs sending results to both the graphics and master PEs.</td>
<td></td>
</tr>
<tr>
<td>6.14</td>
<td>Process to maximise contrast.</td>
<td></td>
</tr>
<tr>
<td>6.15</td>
<td>Two transputer configuration for off-line real-time display. 'M' represents the master transputer, 'G' represents the graphics transputer.</td>
<td></td>
</tr>
<tr>
<td>6.16</td>
<td>Video output showing a complete frame in DCT space (top-right), and the results of progressively setting more of the coefficient values for the higher spatial frequencies to zero (top-left to bottom-right).</td>
<td></td>
</tr>
<tr>
<td>6.17</td>
<td>Video output showing motion estimation. The white square indicates the 'best match' in the search space for the block on the right.</td>
<td></td>
</tr>
<tr>
<td>6.18</td>
<td>Video output showing two sequential frames (bottom); luminance of frame difference (top-left); DCT space of frame difference (top-right).</td>
<td></td>
</tr>
</tbody>
</table>
6.19 Video output showing non-linear enhancement of transform-space enlarged. ........................................ 154
6.20 Video output showing (left) an unprocessed frame and (right) a low pass filtered frame. ........................................ 155
6.21 Video output showing the visual effects due to coarse quantisation when the transmission buffer overflows. ....................... 156
6.22 Efficiencies for H.261 simulations using a) C & CS Tools and b) occam & Tiny. ........................................ 159

7.1 Suggested architecture for improved performance. .................. 168
List of tables

5.1 Breakdown of H.261 function execution times................... 107

6.1 Breakdown of H.261 function execution times for a single macro block 135

6.2 Timings for DCT/IDCT on a QCIF frame with different numbers of tasks buffered...................... 157
The following list defines all the important terms in this document. They are all defined in the text on their first occurrence.

**Achromatic image** A grey-scale image with luminance values but no variation in chrominance values.

**ASI** Application Support Interface.

**ATM** Asynchronous Transfer Mode.

**B-ISDN** Broadband ISDN.

**Block** Part of the H.261 data structure hierarchy.

**BT** British Telecom PLC.

**BTL** British Telecom Labs, the more recent name for BTRL.

**BTRL** British Telecom Research Laboratories.

**CASE** Cooperative Awards in Science and Engineering.

**CCITT** Comité Consultatif International de Télégraphique et Téléphonique (International Telegraphy and Telephony Consultative Committee).

**CEPT** Confederation of European PTTs.

**Chromatic image** A tri-stimulus colour image.

**CM** Connection Machine.

**CPU** Central Processing Unit.

**CS** Computing Surface.

**CIF** Common Intermediate Format.

**Channel** The conceptual means via which processes communicate. Channels are uni-directional.

**CT** Cosine Transform.

**Codec** A device which performs both coding and decoding.

**Communications bound** A parallel program is termed 'communications bound' if the time spent message passing dominates the time spent doing useful processing.

**Computations bound** A parallel program is termed 'computations bound' if the processing power of the processors is the limiting factor on overall execution time.

**DAP** Distributed Array Processor.

**DCT** Discrete Cosine Transform.

**DFT** Discrete Fourier Transform.

**Diameter** A graph theoretic term for the longest of the shortest paths between all pairs of nodes.

**Distance** A graph theoretic term for the number of edges which make up a path.

**DPCM** Differential Pulse Code Modulation.

**DSP** Digital Signal Processing.

**Degree** Graph theoretic term. The degree of a node in a graph is the number of edges with which it is incident.

**ECS** Edinburgh Concurrent Supercomputer.

**Entropy coding** An efficient coding method which encodes frequent events with fewer bits than it does infrequent events.

**EPCC** Edinburgh Parallel Computing Centre.

**ETSI** European Telecommunications Standards Institute.

**FFT** Fast Fourier Transform.

**FLOPS** Floating Point Operations Per Second.
GLOSSARY

FPU Floating Point Unit.
GOB Group Of Blocks. Part of the H.261 data structure hierarchy.
GP MIMD General Purpose MIMD.
H.261 The CCITT recommendation for an international videophone image compression algorithm.
HDTV High Definition Television.
HLL High Level Language.
Huffman coding A specific kind of entropy coding which uses predetermined variable-length words.
HVS Human Visual System.
Hybrid Codec A codec which uses more than one scheme to obtain compression. For example, transform compression and DPCM.
I/O Input/Output.
ISDN Integrated Services Digital Network.
JPEG Joint Photographic Experts Group. The international body seeking to standardise methods of compression of still-frame, continuous-tone, photographic, digitised images.
KBC Knowledge-based coding.
LAN Local Area Network.
Load balancing Distributing the work amongst the available PEs so as to keep them as busy as possible.
Loosely-coupled (of PEs) Memory is not shared but instead, each PE has its own memory.
Mapping problem The task of optimally assigning processes to processors in a parallel processor array.
MB Macro Block. Part of the H.261 data structure hierarchy.
MBC Model-based coding.
MC Motion Compensation.
MCU Master Control Unit.
MFLOPS MegaFLOPS — i.e. millions of floating point operations per second.
MIMD Multiple Instruction stream, Single Data stream.
M²VCS Meiko Multiple Virtual Computing Surfaces.
MSE Mean Square Error.
Multicomputer A loosely-coupled MIMD machine.
OPS Occam Programming System.
OS Operating System.
PCM Pulse Code Modulation.
PE Processing Element.
Pixel Picture element. The smallest element of the display device whose colour can be set independently.
Processor A physical computing element. Conventional serial machines would have one i.e. the CPU whereas parallel machines have many.
PSTN Public Switched Telephone Network.
PTT Post, Telephone and Telegraph organisation.
QCIF Quarter CIF.
RAM Random access memory, i.e. memory which can be both read and written.
RISC Reduced Instruction Set Computer.
RM Reference Model.
SDH Synchronous Digital Hierarchy.
SIMD Single Instruction stream, Multiple Data stream.
TC Transform Compression.
TDS Transputer Development System.
Tightly-coupled (of PEs) Memory is shared via some means of access control.
Through-routing Since most parallel processor arrays are not fully interconnected, it is necessary for processors to forward messages addressed to other processors. This forwarding is called through-routing.
**Topology** The interconnection pattern of an array of processors.

**Variable-length coding** See entropy coding.

**VBR** Variable Bit Rate.

**VLC** Variable-Length Coding.

**VLSI** Very Large Scale Integration.
By sitting in the alcove, and keeping well back, Winston was able to remain outside the range of the telescreen, so far as sight went.

He had set his features into the expression of quiet optimism which it was advisable to wear when facing the telescreen...to wear an improper expression on your face...was a punishable offence. There was even a word for it in Newspeak: facecrime, it was called.

George Orwell (1949)—'Nineteen Eighty-four'.
Chapter 1

Introduction

Video phone data compression algorithms have been developed over the past few years by a laborious process: block diagram design of an algorithm; simulation on computers; design and building of prototype hardware; testing of such prototypes between sites. This design process and its various stages would typically be iterated many times until the final result was satisfactory. As a result, the whole series of events becomes very time consuming. Further, without real-time or near real-time simulation aids it is difficult to assess the relative performance advantages and disadvantages of different algorithms. After several years of development and comparison with different algorithmic approaches [1], the CCITT recently set an international standard for motion video codecs (C0der-DEC0der), recommendation H.261 [2].

The development of motion video coding algorithms can be greatly assisted by real time simulation. A flexible simulation is required which provides visualisation of the results of the various stages of the algorithm. The computational intensity of such a system makes real-time simulation on sequential processors impossible, and so concurrency is needed. In this thesis the software simulation stage is considered.
Until recently simulations were performed on serial mainframe computers, and the vast amount of processing required dictated that run-times of many minutes if not hours were required. This thesis discusses speeding up simulations by the use of concurrency, but it should be borne in mind that the use of concurrent computers is a relatively new field and there really are no standard techniques to adopt.

In this work, the aim was to provide a means of producing simulations of relatively complex videophone image compression algorithms in close to real time. The simulations are flexible in the sense that the algorithm can easily be altered and parameters changed without much effort on the part of the user. A primary aim was also to provide a high degree of algorithm visualisation — i.e. the simulation can provide video output from any stage of the algorithm.

The H.261 algorithm has complexity typical of the present generation of motion video coding algorithms. It exploits both inter-frame (temporal) and intra-frame (spatial) redundancies in the image sequences. This thesis describes the application of a reconfigurable, distributed memory, multiple instruction multiple data (MIMD) computer (or multicomputer), to the simulation of the H.261 algorithm. Fortunately the possibility of videophony through the arrival of ISDN coincides with the availability of parallel computer hardware.

Parallel techniques for speeding up software simulations of video codec algorithms on a reconfigurable MIMD machine with distributed memory (or multicomputer) are discussed. The multicomputer used for this work, the Edinburgh Concurrent Supercomputer, houses over 400 inmos T800 transputer floating point microprocessors. The techniques developed in this thesis have been applied only to the H.261 algorithm. However, they are not algorithm-specific techniques and so are generally applicable to motion video image compression algorithms in general.
1.1 Emerging communications standards

The advantages and disadvantages of standards are well known. In their favour is the fact that they allow the design of compatible interacting devices. However, standards take time to define; by the time they are finally released technology has moved on. In the field of communications the advantages of the former by far outweigh the disadvantages of the latter. There are various standards bodies throughout the world. CCITT is the international communications standards body which only has the power to make recommendations. Within the CCITT are the various PTTs (Post, Telephone and Telegraph organisations) from the individual nations, e.g. British Telecom and Deutsche Bundespost. Within Europe there is the Confederation of European PTTs (CEPT) which is responsible for overseeing the actual implementations of the standards. Recently CEPT was replaced with the European Telecommunications Standards Institute (ETSI). They are concerned with such aspects as videophone terminal design [3].

Existing analogue trunk telephone lines are being upgraded to digital operation. Extensive use is being made of optical fibres each with a capacity upwards of 500 Mbit/s. The emerging ISDN standard [4, 5, 6] recommended by CCITT is characterised by three main features:

- End-to-end digital connectivity
- Multiservice capability (voice, data, video)
- Standard terminal interfaces

Almost twenty years ago CCITT published the following definition:

An ISDN is a network evolved from the telephony integrated digital network (IDN), that provides end-to-end digital connections to support a wide range of services, including voice and non-voice, to which
users have access via a limited number of standard multiple-purpose interfaces. [7, 8]

The standards for ISDN are collectively called the I-series of recommendations which have been produced by CCITT Study Group XVIII [9]. ISDN is evolving from the existing digital telephone networks in stages; first voice and data only at bit rates of 64 kbit/s, later broadband *bulk video services*.

### 1.1.1 Narrowband ISDN

The early architecture is called *basic access* which is configured as $2B + D$, where $B$ and $D$ are channels providing 64 kbit/s and 16 kbit/s respectively. The BT basic-rate implementation, which is just becoming available in Britain, is known as ISDN 2 [8]. Eventually, *primary access* up to several hundred Mbit/s should be possible. This will be implemented as $pB + D$ where $p$ ranges from 1 to 23 in North America and from 1 to 30 in Europe. In this case both $B$ and $D$ channels provide 64 kbit/s. The result is a network that can provide $p \times 64$ kbit/s to any user that is prepared to pay the tariff.

Video-conferencing requires a few megabits per second to encode a standard colour TV signal: specifically, rates of 2 Mbit/s for Europe and 3 ($2 \times 1.5$) Mbit/s for North America are possible choices. Commercial video encoders for colour video-teleconferencing have been available for some time that require bit rates from 2 Mbit/s down to 384 kbit/s [10]. There are high-capacity channels (H channels) which provide multiples of the $B$ rate, *e.g.*:

\[
H_0 = 6B = 384 \text{ kbit/s}
\]

\[
H_{12} = 30B = 1920 \text{ kbit/s}
\]
1.1.2 Broadband ISDN

Narrowband ISDN (N-ISDN) is a vast improvement on the old analogue IDN. However, the basic-rate channel capacities of N-ISDN were still derived from the bandwidth requirements of speech \( p \times 64 \text{ kbit/s} \). Standards are being discussed for broadband ISDN (B-ISDN) \([11]\) which will allow the ISDN standards to expand gracefully into a service which will be able to support mixed media services, high-definition TV and whatever the future demands, by providing gigabits of bandwidth per second \([12, 13]\). It appears that most of these services will be of the variable bit rate (VBR) type. For this reason it has been found most appropriate to define B-ISDN as operating in asynchronous transfer mode (ATM). This means that data is sent in packets which can be interleaved and thus transmission rates are not limit to the multiples of 64 kbit/s used in N-ISDN. The physical layer of B-ISDN will be implemented as an optical fibre network. There is a North American transmission standard for this known as Sonet (synchronous optical network) \([14]\), but the international convention is the synchronous digital hierarchy (SDH).

There are many potential advantages of B-ISDN over N-ISDN. The interleaving of packets will mean that bandwidth can more efficiently be shared between users without having to allocate each individual user with a bandwidth commensurate with the predicted maximum transmission rate. Many more services will be able to be provided such as remote meter reading, transmission of high resolution graphics, full motion video and HDTV (high definition TV). However, since only basic access will be provided to the vast majority of users in the near future, low bit-rate image coding techniques will continue to be important for some time.
1.2 Videophony

Videophony is a relatively new term which is a shortened form of video-telephony. It is used as a name for person-to-person communications using audiovisual services. Videophony differs from videoconferencing in that the former is usually between two individuals using on-demand switched networks; whereas videoconferencing often involves larger groups of people, uses pre-booked leased lines for transmission and is generally of higher quality. At the time when the research recorded in this thesis began, videophony was not available in any form. Since then, three types have emerged [3]:

1. A videophone for the deaf using the present analogue PSTN (Public Switched Telephone Network). Due to the low bandwidth constraint, the image is a mere 64 pixels square and the pixels are restricted to being bi-level in value (black or white). The crude image generated is good enough for transmitting sign-language at a variable rate of 6–12 frame/s.

2. A local or metropolitan high quality videophone using high bandwidth optical fibre for transmission.

3. Videophones for the ISDN network. Crucial issues include:

   • Can just one of the two B channels be used to carry both audio and visual data? Reasonable quality voice signals can be transmitted at a rate of 16 kbit/s leaving 48 kbit/s for the compressed motion video signal.

   • How much will the service cost? Surveys indicate that most users would welcome videophones but only if the costs are comparable to those of telephones.
CHAPTER 1. INTRODUCTION

1.3 Thesis synopsis

The research recorded in this thesis investigates parallel techniques for performing real-time simulations of videophone image compression algorithms on multicomputers. This section explains what is contained in the remaining chapters of this thesis. The book falls into two main divisions. Chapters 2, 3 and 4 orient the reader unfamiliar with the fields of motion video codecs, parallel computer architectures and parallel computer programming respectively. The research methods and results appear in Chapters 5 and 6. The contents of each chapter are given in more detail below:

Videophone image compression algorithms are reviewed from their origins to the present state-of-the-art in Chapter 2. In particular, the international standard, CCITT recommendation H.261 is described in detail. The way in which videophones are currently implemented in hardware and how this may change in the future is discussed.

Chapter 3, Parallel computer architectures, is concerned with the various types of parallel computers available. It is generally agreed that cost effective computers must be parallel if they are to provide the power required for future computational problems. It is not only a matter of economics; fundamental laws of physics dictate that processors cannot operate faster than the speed of light. This limit is being reached by modern, very large scale integration (VLSI) processors and so parallelism must be used if greater processing power is to be gained. The inmos transputer is one such building block which can be used to build multicomputers. The idiosyncrasies of the transputer thus are described, along with the claimed improvements of the awaited next generation device in the transputer family. The University of Edinburgh has a transputer-based supercomputer called the Edin-
burgh Concurrent Supercomputer (ECS) upon which all the parallel simulations described in this thesis were implemented. The architecture of the ECS is described from the viewpoint of the programmer.

Chapter 4, *Parallel computer programs*, examines the problems encountered by programmers of parallel computers. Programming for parallel processors presents many new difficulties as parallel computers provide much higher levels of complexity for the programmers to handle. These include new programming languages, decomposing the problem onto the processors and the difficulties arising from data communication between processors. Few techniques exist for automating the efficient mapping of problem onto the processing elements of parallel machines. Thus potential methods of mapping problems onto the processing elements of the parallel computer are discussed. Topology independent routing harnesses can remove many of the problems encountered in programming a parallel computer. The need for general purpose parallel computers is highlighted. They will provide virtually unlimited processing power to the user whilst hiding the problems of implementation details.

Since transputers were being used as the processors it was natural, at the time, to implement the H.261 simulations in occam\textsuperscript{1}. The novel syntax of occam is explained in Chapter 5 together with the Meiko occam programming system (OPS) which was the programming environment used to develop all the occam programs. Different combinations of routers and programming languages were employed in these simulations. The advantages and disadvantages of each of these are outlined. Related implementations of similar algorithms on multicomputers are compared

---

\textsuperscript{1}Despite having inherited its name from the fourteenth century English philosopher, William of Occam, this language is not conventionally written with an initial capital letter. Additionally, throughout this volume, it shall be written in sans serif font, after the style adopted in the texts of its originators, inmos.
and contrasted with the simulation techniques used in this research.

In Chapter 6 the results of the simulations are presented with emphasis on the ease of use of the techniques employed. Theoretical limits on parallel speedup and efficiency are considered and their impact is examined when attempting to simulate the H.261 algorithm in real time. Performance results are presented for a load-balancing chain and the task farm implementations. Examples of simulation video output are given. The high degree of visibility of the intermediate results of the algorithm afforded by the chosen simulation techniques is considered.

Results and the relative success of the approaches to the simulations are summarised in Chapter 7. Conclusions of the research are discussed with suggestions for extensions to the work.

Appendix A contains details of the published papers on which this thesis is based. The three conference papers are reproduced in full in this appendix.

This thesis will necessarily introduce many terms which may be foreign to the reader. They are all defined when first used. The reader is directed to the thesis Glossary (p. xiii) for quick reference should confusion arise.
Chapter 2

Motion video codecs

2.1 Introduction

Ever since the first experiments in television transmission in the 1920s there has been considerable interest in image coding methods. However it is only in the last twenty years that much progress has been made. The main reasons for this are the discovery of new coding techniques and the availability of cheap, reliable, fast digital electronics.

There is an increasing number of applications for which digital pictures are being used. These include broadcast and relay TV, satellite remote image sensing, videoconferencing, facsimile, remote-piloting of vehicles and image-archiving (e.g. X-rays, engineering drawings, fingerprints.) All of these applications involve transmission and/or storage of images.

This project is concerned with motion video coding for the likes of videoconferencing and videotelephony. Using audio telephones is now second nature to the people of the developed world. However, most of us are unfamiliar with the use of visual telephones. There are several reasons for this outlined below.
A typical audio signal requires a bandwidth of 3.4 kHz whereas a television signal may require 4.3 MHz. Obviously enormous compression is required to prevent the transmission of visual telephony becoming prohibitively expensive. Recently digital coding techniques have advanced apace but progress has been retarded by a lack of standards: users cannot communicate unless they know each other’s compression algorithm. Ideally everyone would use standardised algorithms (Section 2.5). Once a standard has been agreed then codecs can be mass produced and costs begin to fall (see Section 2.7).

As explained in Section 1.1, the communications network we have inherited was designed only with the meagre bandwidth requirements of audio communications in mind. Most forms of image and data transmission require more bandwidth than that currently available. Fortunately the announcement of a standard international digital network comes at just the time when visual telephony is becoming practicable.

All previous attempts at providing a visual telephonic service have failed commercially for want of one or more of the above requirements. In the 1990s, almost all of the problems have been dealt with. It remains to be seen how long it will take for videophony to become ubiquitous. Perhaps other, more subtle (social), problems will emerge to prevent videophones from becoming popular.

2.2 A brief history of videophony

2.2.1 Early trial systems

Experiments in videophony began in early 1960s at Bell Laboratories. Their system was known as Picturephone. These proved useful in defining what problems needed to be solved in the future years. Two trial commercial systems emerged
in the early 1970s called *CONFRAVISION* and *VISIOCONFERENCE* [15]. None of these systems were commercially successful but they made clear the need for less expensive, more convenient videoconferencing/videophone systems. The most glaring inadequacy was the lack of standards which prevented international communications.

### 2.2.2 European standards

In March 1977 there was the first meeting of the European Project on Redundancy Techniques for Visual Telephone Signals [16]. This project was otherwise known as COST211 (Cooperation in Scientific and Technical Research). In 1982, the project concluded with a European standard for monochrome videophony using the PAL TV standard with 625 lines, a frame rate of 25 Hz and an overall bit rate of 2 Mbit/s.

Subsequent to COST211, there have been numerous hardware trials [17] for testing flexible prototypes between countries in Europe by satellite. However, at the same time American and Japanese companies were manufacturing their own incompatible codecs. High demand for communications across the North Atlantic resulted in CCITT recommendations H.120 (Figure 2.1) and H.130. These are the old generation of standards which were based on differential pulse code modulation (DPCM) and *conditional replenishment*—i.e. only parts of the image which have changed significantly from the previously transmitted frame are coded (Section 2.3.2).

### 2.2.3 International standards

As a result of the aforementioned chaos due to the lack of standards, CCITT formed the Specialist Group on Coding for Visual Telephony inside Study Group XV. The
CHAPTER 2. MOTION VIDEO CODECS

Figure 2.1: Simplified block diagram of the H.120 algorithm.

generation of videoconference and videophone codecs which work at bit rates from
2 Mbit/s down to a mere 64 kbit/s. This standard is known as recommendation
H.261 and is considered in detail in Section 2.5.

Two other standards have recently been proposed. These are the Joint Pho-

tographic Experts Group (JPEG) standard for still picture compression and the
Moving Pictures Experts Group (MPEG) standard for full-motion compression on
digital storage media. The MPEG standard algorithm is very similar to the CCITT
H.261 algorithm, being based upon the same functional blocks. The details of how
quantisation and motion estimation are achieved differ, however.

2.3 Image coding overview

The aim of image compression (or coding) is to represent pictures by as few code
symbols as possible for a specified acceptable limit to image degradation. Thus the
classic problem of image transmission is the search for coding methods which can
achieve this first aim while at the same time not being overly sensitive to channel
CHAPTER 2. MOTION VIDEO CODECS

errors.

Many methods of compression exist, some of which depend heavily on the class of image being operated upon. Only colour (chromatic) and grey-scale (achromatic) images will be considered in this section i.e. not bi-level images, but images of the type suitable for videophony. Another consideration is who or what will be the end viewer of the images. It will be assumed that the pictures are to be viewed by humans and so the compression techniques can take advantage of psychovisual traits: pictures usually contain ‘redundant information’ that the observer cannot utilise.

2.3.1 Image quality

If the sink for the transmitted image is human then picture quality must be subjective. There is no good reason to suppose that subjective quality is one-dimensional. If it is multi-dimensional then it cannot be ranked. In practice, however, it has been treated as scalar and 5-step scales devised ranging from ‘excellent’ to ‘unacceptable’. Using this sort of scale many experiments have been conducted but measurement is tedious and difficult. For these reasons objective quality measures have been proposed [18]. The most popular so far is mean-square-error (MSE). A variant of MSE is weighted MSE which takes into account the sensitivity of the eye to various spatial frequencies.

2.3.2 Methods of image coding

There are currently two ways of compressing images: reduction of statistical redundancy and psychovisual redundancy.
Statistical redundancy Of all the possible combinations of pixel intensities in an image, most appear as a jumble of noise. Images that are normally considered worth viewing possess a high degree of both spatial and temporal correlation. That is to say there is a strong dependency between the values of individual pixels.

Psychovisual redundancy The perceptual limitations of human vision can be exploited [19]. These include limitations in amplitude, spatial resolution and temporal acuity. Coders can be designed to selectively discard statistical information without objectionably affecting the perceived image.

There follows a summary of the major image-coding techniques developed thus far.

Pulse-code-modulation

Pulse-code-modulation (PCM) is the simplest of all the coding schemes. It is nothing more than a time discrete, amplitude discrete, representation of the image. The image is sampled, quantised and binary coded. Achromatic images usually use quantisation with 64 to 256 levels per sample, corresponding to 6- or 8-bits-per-sample. To transmit an image by PCM requires $L \times L \times B$ bits/frame, where the image has $L \times L$ pixels and $B$ is the number of bits used to code each pixel. Thus there are $2^B$ possible grey-levels. It has been found that the human visual system (HVS) is only capable of discerning up to 64 grey-levels which indicates that no more than 6 bits are required per sample.

The image is compressed by virtue of the fact that it is sampled from the infinite analogue resolution down to a finite digital resolution and quantised. PCM is considered as the 'uncompressed' image against which the bit rates of the other
Entropy coding: variable length codes

By coding groups of pixels with a code word whose length is inversely proportional to the probability of the group appearing in an image we can hope for substantial transmission-rate reductions. The Huffman coding procedure can provide a minimal average length code for a given set of probabilities. This works well for bi-level images (e.g. facsimile) but for chromatic or achromatic images the number of patterns can be enormous even for small groups of pixels. Thus this form of compression is only used in combination with others such as predictive and transform compression.

Predictive coding: DPCM

Predictive coding, also known as differential pulse code modulation (DPCM), estimates the amplitude of each scanned pixel based on the history of previously scanned pixels. Next the estimate is subtracted from the actual pixel amplitude and the difference is quantised and coded for transmission. At the receiver, the estimate and difference can be summed to form the actual image. Compression is achieved by coarsely quantising the differences. The pixel amplitude predictions can be based upon previous pixels, lines or frames depending on the complexity of the coder.

DPCM has the advantage that it is easy to implement. However, it is vulnerable to channel noise and does not produce as low a mean-square-error image reconstruction as transform coding.
Frame replenishment

For many applications, e.g. the transmission of a person's head and shoulders in video-teleconferencing, the majority of pixels remain virtually unchanged from frame to frame. In a frame replenishment coder, each frame is digitised at 8 bits/pixel. The first frame is stored at the coder and transmitted. Subsequent frames are compared to the stored frame. Pixels which are significantly different are transmitted with their address and are also stored at the coder. Bit rates of 1 bit/pixel (1.5 M bits/s) for monochrome teleconferencing have been reported but the system breaks down under violent motion. The equivalent colour teleconferencing requires 6 M bits/s and colour broadcast TV requires 22 M bits/s [18].

Contour and run-length coding

These are both methods which work well for pictures with only a few regions each having a constant grey-level. Contour coding traces the boundaries between the constant grey-level regions and sending only that information to the receiver. Run-length coding codes the 'runs' of constant grey-level obtained by raster scanning the image and specifying the lengths of the runs.

Block truncation coding

Block truncation coding [20] has a bit rate performance comparable to that of the non-adaptive cosine transform (1.6 bits per pixel) and yet has a much simpler hardware implementation. The image is split into small blocks. For each block the mean and variance of the pixel amplitudes is calculated. Then each pixel is truncated to one bit by thresholding and moment preserving selection of binary levels. The mean, variance and single-bit pixel values are transmitted in place of the block. Generally, the picture quality is not as good as that achieved by
transform coding but block truncation coding performs better with a very noisy channel.

**Transform coding**

Transform coding (TC) has been the subject of much research for the past fifteen years [21, 22, 23, 24, 25, 26]. It is now recognised as the most effective procedure to date. First the picture is transformed into a set of more independent 'coefficients' by an invertible mathematical transformation. The coefficients are then coded and quantised for transmission. At the receiver the received bits are decoded into coefficients to which an inverse transform is applied to recover the pixel intensities.

Some of the compression is achieved through rough quantisation of the coefficients. Also some of the coefficients are discarded (zonal sampling) and variable bit allocation can be used in the representation of the coefficients (zonal coding). This might simply mean retaining, say, only the first $N/2$ of the $N$ coefficients ranked by some means such as information content and subjective quality.

Theoretically the optimum transform would produce uncorrelated coefficients while compacting image energy into as few coefficients as possible. Also it is desirable that the mean-square error in reconstruction of the image after dropping some coefficients is a minimum. There is one transform which satisfies all these criteria called the Karhunen-Loève transform (KLT). However the KLT is not generally used since it is difficult to implement; it requires a priori knowledge of the incoming image data. It has been simulated on computers and used as a means of measuring the performance of other transforms. Many suboptimum transforms have been investigated notably the Fourier (FT), Hadamard (HT), Cosine (CT), Slant, Walsh and Haar transforms. Most recent interest has been in the CT. This is due to the fact that its performance approaches that of the KLT with increasing
transform size and also fast implementations have been discovered based on the fast Fourier transform (FFT).

In this way, bit rates of 2 bits/pixel have been achieved starting from a source of 8 bits/pixel. Experiments with transform size have yielded better results. This technique is called 'block quantisation'. The image is divided into blocks of usually $16 \times 16$ or smaller which are each transformed as separate subpictures. Using this technique, bit rates of around 1 bit/pixel have been achieved [22]. However, these results were for still images. Interframe techniques are required for reconstructing moving images at bit rates of 2 bits/pixel or less.

Transform compression deals well with channel noise. Each pixel amplitude in the reconstructed image is a weighted function of all the transform coefficients. Hence, the magnitude of a single channel error is distributed over all the image pixels, and so is not particularly visible to the human viewer, whereas error concentrated in one area of the image is much more objectionable to the observer. Due to the inherent complexity of transform compression, most of the work has been computer simulations operating on still pictures and not hardware, real-time implementations. However modern parallel computing techniques and generally faster and cheaper hardware are making this possible. The majority of image coding research has been done in this area and it still seems to hold the most promise of all the compression methods to date.

**Adaptive algorithms**

An adaptive algorithm is one which does not operate identically for all images and all image regions. It adapts to the local statistics. We would expect these algorithms to give better results since images do not have constant statistical structure, both from image to image as well as within an image. Thus, for example,
the number of coefficients retained in transform compression can vary from block to block depending on what percentage of the total image energy is required \[23\]. The results of transform compression would indicate that adaptive algorithms must be adopted if they are to make significant progress.

**TC/DPCM hybrid coding**

Both TC and DPCM are based on source decorrelation prior to quantisation. This coding scheme utilises both transform and DPCM compression in an attempt to gain the advantages exhibited by each. A one-dimensional transform is taken along the rows of the image and then the coefficients are encoded in a DPCM technique across the transformed rows. The result is much faster than a full blown two-dimensional transform and has made real time television bandwidth reduction systems possible. In practice, hybrid coders \[27\] combine the advantages of the low hardware complexity of DPCM coders and the high performance of transform coders. The resulting performance lies somewhere between the two. Hybrid coders combining intraframe and interframe coding with DPCM can exploit the inherent properties of video.

### 2.4 Video format

One of the major problems of producing an international videophone codec standard was that there was no international TV signal standard. There are currently three major TV signal standards: NTSC, PAL and SECAM. Each is a composite signal and is incompatible with the other two. For this reason a *common intermediate format* (CIF) was defined by CCITT. All video codecs will be forced to use CIF thus obviating the differences between the European (625/50) and North
2.4.1 Colour video signals

Knowledge of the way the HVS works makes it possible to encode colour images without incurring excessive overheads when compared to processing monochrome images. A three-dimensional colour space can be conceptualised within which every point represents a unique colour [28]. Thus we need three parameters to uniquely represent a colour thus colour signals are termed tri-stimulus. Most common output devices use the red-green-blue (RGB) coordinate system.

The coding of colour images is similar to coding monochrome images. However, it is advantageous to precede the processing step by a transformation to reduce the correlation between the $R$, $G$ and $B$ signals. Historically the technique used has parallels with the evolution of broadcast TV from monochrome to colour. In the early days of TV, the monochrome (or luminance) signal was broadcast with a certain well defined bandwidth allocation. When the system was upgraded to colour, no more bandwidth was required since the chrominance signals were interleaved in the zero regions of the luminance signal. Thus transforms were devised to translate from $RGB$ signals to luminance and two chrominance signals ($YUV$) [29]. The $RGB$ signals are highly correlated and so for a general image the energy is fairly evenly distributed amongst them. By contrast, for $YUV$ images, most of the image energy is concentrated in the luminance signal.

Since the HVS is less sensitive to colour resolution than it is to luminance resolution [26] and because there is less energy in the chrominance signals, an image can be represented in fewer bits than are required for the $RGB$ coordinates description. The trick is to perform a 3-D rotation of the orthogonal $RGB$ axes in colour space. The new axes are $Y$ (luminance), $U$ and $V$ (chrominance). $U$ and
V are sampled at half the frequency of Y. Thus a typical 24-bit RGB pixel (eight bits for each of R, G and B) becomes a 12-bit YUV pixel (8 bits of Y and 4 each for U and V.) Fifty percent compression is automatically achieved in this way for all images.

Since Y is the luminance or monochrome signal, the Y axis lies exactly inbetween the R, G and B axes. This is intuitively correct since an equal increase in R, G and B should result in an increase in luminance. Now, to complete the orthogonal set of new axes, there are many choices for the positions of U and V within a plane in colour space. Several standards have been set, the one used by the CCITT H.261 algorithm calls U and V, CR and CB, respectively. This reflects that they are functions of the difference between B and Y, or R and Y [30] (Section 6.6.1):

\[
C_B = f(Y - B) \\
C_R = f(Y - R)
\]

### 2.4.2 Common intermediate format

Local images will be preprocessed to comply with CIF before they are introduced to a video codec [31]. CIF uses a colour signal comprising of a luminance (Y) and two chrominance (CR, CB) (see Section 2.4.1) and is partially defined in terms of the following parameters (Figure 2.2):

- Pixels per line: Y = 360, CR = 180, CB = 180
- Lines per frame: Y = 288, CR = 144, CB = 144
- Frames per second: 29.97

The two chrominance signals are spatially subsampled taking advantage of the fact that the HVS is less sensitive to chrominance than luminance information, and
also that most of the the energy of images is contained in the luminance signal.

As well as interregional connection, the picture format affects the performance of sub-primary bit rate codecs. Thus the study group determined that in the short term full CIF would be difficult to achieve. It was decided in January 1988 to specify a second, smaller, format called quarter CIF (QCIF): 144 lines \( \times \) 180 pixels \( \times \) 29.97Hz. This format has sufficient resolution for the head and shoulders image of a single person necessary for videophony.

Uncompressed QCIF would require a transmission bandwidth of 9.12 Mbit/s. Thus, assuming an ISDN bearer-channel of 64 kbit/s, we need a compression ratio of 143:1. The amount of processing this requires pushes technology to its limits at the present time and so experiments are concentrating upon QCIF rather than full CIF. Also the frame rate is often subsampled from 30 Hz down to 15, 10 or even 7.5 Hz to assist the codecs in the job of compressing full motion video.
2.5 CCITT motion video codec standard H.261

This section gives a general overview of the particular codec standard that concerns this project. This has been studied by the Specialist Group on Coding for Visual Telephony in Study Group XV of the CCITT [32]. In the following discussion some parameters depend upon the chosen frame and bit rates for operation. Throughout this research a transmission bit rate of 64 kbit/s and a frame rate of 10 Hz have been assumed.

2.5.1 Overview of the algorithm

Generally, H.261 can be described as a hybrid coding algorithm [26, 33, 34]. Several of the techniques described in Section 2.3 are combined. Both inter-frame (temporal redundancy) and intra-frame (spatial redundancy) are exploited, the first by DPCM and the second by discrete cosine transformation of prediction errors. The transform coefficients are quantised and variable length coded before being sent to a buffer from where they are transmitted down the channel [35]. Coding efficiency is also improved by using motion estimation.

Initially inter-frame prediction coding is performed in the pixel domain. The prediction scheme is differential pulse code modulation. This simply means that each pixel in the new frame is assumed to be the same as the pixel in the same location in the previous frame. Thus the prediction errors are generated be simply subtracting the new incoming frame from the previous frame which is kept in a frame store (Figure 2.3). The prediction error is transformed into the spatial frequency-domain where quantisation and classification take place. Motion compensation can optionally be included in the prediction stage (DPCM loop). Generally speaking the coding scheme removes redundancy in the temporal do-
main with (motion compensated) DPCM and redundancy in the spatial domain with the discrete cosine transform (DCT). The modules of the coder are described in more detail below.

The H.261 algorithm operates upon pixel blocks of $8 \times 8$ pixels. For this reason a *significant pixel area* (SPA) is defined which in the case of QCIF (Section 2.4) is $176 \times 144$ pixels. This is achieved by removing two pixels each from the left and right edges. The missing pixels can easily be supplied by the decoder as, say, black to make the output frames conform to CIF dimensions.

### 2.5.2 Reference models

To enable fruitful software simulations of these hardware standards to be carried out at various sites, a reference model has been drawn up detailing certain accuracies and methods which should be employed. The latest version is called *reference model 8* and shall hereafter be referred to as RM8 [36]. This was the source used for details of the H.261 algorithm when the simulations described in this thesis were written.

### 2.5.3 Picture hierarchical structure

In order to facilitate source coding, the picture data is organised in a hierarchy (Figure 2.4). At the lowest level there are $8 \times 8$ blocks of pixels. These blocks are grouped into *macro blocks* (MB) which comprise four luminance and two chrominance blocks. MBs are in turn grouped into *groups of blocks* (GOB) which contain three rows of eleven MBs each. Thus one GOB represents one third of a QCIF picture. The root of the tree is the picture itself which is made up of GOBs: three for QCIF, 12 for CIF.
CHAPTER 2. MOTION VIDEO CODECS

Figure 2.3: Block diagram of the H.261 motion video coder.

Figure 2.4: Picture hierarchical structure.
2.5.4 Interframe DPCM and conditional replenishment

Most of the time the coder runs in inter-mode which means that inter-frame prediction errors are coded. However if there is a scene cut or forced update the original image can be transmitted; this is called intra-mode.

2.5.5 2-D transform

Blocks of size 8 × 8 pixels are transformed into the spatial frequency-domain using the DCT [37]. The DCT is similar to the well known DFT (Discrete Fourier Transform) in that it transforms the input samples into the frequency-domain. However the DFT differs from the DCT in that the latter does not operate upon complex numbers; the data remains wholly real throughout the transformation. The one-dimensional DCT and inverse DCT (IDCT) are as shown in (2.1) and (2.2):

\[ Y_k = c_k \sqrt{\frac{2}{N}} \sum_{n=0}^{N-1} X_n \cos \left\{ (2n + 1) \frac{k\pi}{2N} \right\}; k = 0 \ldots N - 1 \quad \text{(2.1)} \]

where \( c_k = \frac{1}{\sqrt{2}} \) if \( k = 0 \), and 1 otherwise.

\[ X_n = \sqrt{\frac{2}{N}} \sum_{k=0}^{N-1} Y_k c_k \cos \left\{ (2n + 1) \frac{k\pi}{2N} \right\}; n = 0 \ldots N - 1 \quad \text{(2.2)} \]

A two-dimensional block can be transformed by performing one-dimensional transforms on the rows and then on the columns [26]. As with the the DFT there exist many ‘fast’ algorithms for the DCT based upon ‘butterfly’ [38] signal-flow graphs [39, 40, 41, 42]. There are also ‘fast’ algorithms designed specially for performing the 2-D DCT without using a 1-D DCT transform [43].

The size of block was selected taking coding efficiency, picture quality and hardware complexity into account. The accuracy of the transform has been defined
in great detail so as to allow commercial standard VLSI chips to be designed without specifying how they should be implemented.

2.5.6 Classification

For transmission purposes the 2-D blocks must be converted into 1-D arrays. The blocks of transform coefficients can be scanned in various ways. These methods are known as scanning classes which include zig-zag scanning and vertical scanning. The final choice is left to the hardware designer. The selected class becomes a block attribute which is sent to the decoder. It would seem that zig-zag scanning is the most sensible since the majority of high-energy coefficients are of low spatial frequency and also the HVS is most sensitive to changes in low spatial frequency. Therefore the most important coefficients lie in the top-left corner of the transformed block (Figure 2.5). In this way, the most significant coefficients will be towards the beginning of the 1-D array and will be less likely to be lost in the following thresholding and quantisation stages.
2.5.7 Quantisation

The amount of information transmitted is here further reduced by two independent techniques viz variable thresholding and scalar quantisation. It is this stage in the overall algorithm which produces losses and thus determines the overall distortion. Each $8 \times 8$ block of coefficients emerging from the DCT process is scanned to produce a one dimensional data set. These 64 real numbers can be thought of as representing increasing spatial frequencies beginning with the DC coefficient. Since the HVS is more sensitive to changes in low spatial frequencies than high, it is less important to preserve the higher frequency coefficients. This fact is exploited in the thresholding technique which is applied before quantisation.

Variable thresholding

Long strings of zero coefficients can be encoded very efficiently. A strategy to increase the length of zero strings is applied before quantisation and is called variable thresholding. A threshold is applied to the coefficients in turn starting with the DC coefficient. Any coefficients falling short of the threshold are replaced with a zero. At the same time the threshold is increased by one thus making it more likely that subsequent less significant coefficients will be set to zero. In Figure 2.5, the darkly-shaded bars represent coefficients which have fallen short of the threshold and so are set to zero. Hence the threshold does not actually necessarily increase monotonically as shown.

When a coefficient does not fall short of the threshold it is retained and the threshold is reset to its starting value. The resulting set of coefficients is passed on to the quantisation stage.
Scalar quantisation

Each coefficient is processed separately and so this process is termed \textit{scalar} quantisation [44, 45, 46]. Other methods that try to reduce redundancy by quantising data in larger groups are termed \textit{vector} quantisation.

Real numbers emerging from the DCT process must be limited to a finite range by quantisation. A uniform midrise quantiser is employed in RM8 [44]. This means that both the reconstruction and the decision levels are evenly spaced by the step-size, \( g \). The reconstruction levels, \( r_i \), are the mid points between the decision levels, \( d_i \) and \( d_{i+1} \) (Figure 2.6).
Letting \( Q(.) \) represent the quantisation operation, and \( f \) the unquantised data:

\[
Q(f) = \begin{cases} 
0 & \text{if } f = 0 \\
r_i, d_{i-1} < f <= d_i & \text{otherwise}
\end{cases}
\] (2.3)

For each coefficient, \( f \), the quantisation index, \( i \), is determined using (2.3). This is used at the decoder to determine the reconstruction level, \( r_i \).

Initially the step-size, \( g \), of the quantiser is 32. However this value is adjusted at the start of each row of blocks in a GOB — i.e. every 11 blocks — depending on the fullness of the buffer, and ranges from 2 up to 64 in jumps of 2. The buffer size depends upon the frame and bit rates being employed. This allows graceful degradation of image quality when the coder becomes overloaded [47].

### 2.5.8 Variable-length coding

The idea of variable-length coding (or entropy coding) is to assign short codes to common events and longer codes to unlikely events. Thus on average the bit rate is reduced. Huffman coding [48] is a well-known example of variable-length coding which employs a pre-determined table of codes and is known to be optimal. A two-dimensional variable-length code (VLC) is adopted. An event is defined as an ordered pair formed from a non-zero quantisation index and a run-length:

\[
\text{event} = (\text{index}, \text{run-length})
\]

The run-length is the number of zero quantisation indices preceding the current index. Thus the 2-D VLC employed is a combination of entropy coding and run-length coding. It is assumed that the variable thresholding will produce many zero indices and therefore this 2-D VLC should be very efficient.
2.5.9 Motion compensation

This is yet another method of achieving data compression \[49, 50\]. Prediction errors in the DPCM loop are minimised by motion estimation (ME) followed by motion compensation (MC). The DPCM loop described above simply differences macro blocks from the same spatial position in temporally adjacent frames. If, however, by sliding the block up to 15 pixels in any of the directions, north, south, east and west, a better match can be made, then ME has been successful. In this case, the motion vectors of the block are sent as side information. Block matching is only performed on the luminance blocks and the equivalent motion vectors for the chrominance blocks are calculated.

Optimal hardware implementation of the block matching is simply a brute-force match across all possible block positions within 15 pixels. For software simulation purposes (RM8) the search is limited to within 7 pixels and is implemented as a refining algorithm (Figure 2.7) known as a 'logarithmic' search \[49\]. This works
by finding the best match of widely spaced blocks within the search area. The search space is then centred on the best matching block so far and the spacing of the search blocks reduced for the next iteration of the search. This continues until the search blocks are just one pixel apart.

The decision whether to use MC is done on a block by block basis according to an experimentally determined characteristic [36]. Motion estimation and motion compensation are optional parts of H.261 coders whereas all decoders must be able to receive and decode motion vectors.

2.5.10 Loop filter

There is a 2-D digital spatial low-pass filter which is employed after MC and before prediction takes place. The filter is a 121 smoothing filter [19] and is used on both luminance and chrominance blocks and has the effect of smoothing edges introduced by motion compensation. Figure 2.8 shows how it is applied to a block of 8 pixels. Filtering is achieved by convolution with a 3 × 3 kernel (or ‘mask’). However, rather than use information from surrounding blocks in the image frame, the edges of the block are treated differently: corner pixels remain unchanged; remaining edge pixels are convolved with a 1 × 3 kernel.

It can be switched on and off block by block so that computing power is not wasted where the filter is not effective. Experiments have shown that the filter is only really valuable when motion compensation is being used. Thus the filter is only turned on for blocks with non-zero motion vectors.
2.6 Future motion video codecs

The recommendation H.261 took several years to standardise, therefore it almost certainly no longer uses the currently best known techniques. For example it has been proposed that it is a mistake to combine transform coding with motion compensation [51]. The reason for this is that transform coding is computationally intense whereas there may exist better spatial-domain techniques.

Many variations on hybrid coding have been suggested both before and since recommendation H.261 was released in 1990. Vector quantisation has been experimented with as a replacement for the DCT in the DPCM/DCT hybrid coder [52, 53]. Further bit rate reduction can be achieved by skipping frames at the coder and reconstructing them at the decoder by motion compensated frame interpolation [54]. Gains can be made by allowing the transform coder to operate on blocks of varying sizes [55]. Obviously some ideas were too late to be considered for the
standard. One of the main problems of standards is that by the time they are released they no longer use state-of-the-art techniques.

To date the motion video coding algorithms have been based upon the removal of spatial or temporal redundancy in a sequences of images. It is generally agreed that there is little more compression to be achieved using only these statistical techniques. Research into the HVS suggests that we humans actually process images at extremely low bit rates of a few hundred bits per second [19, 56, 57]. The new generation of motion video codecs will further exploit knowledge of the HVS and image understanding. This section describes some of the techniques which are emerging. Attaining extremely low bit rates is of great interest since cost is always directly proportional to bandwidth. As already mentioned, the new international videophone codec standard operates at a minimum of 64 kbit/s which will require one whole ISDN $B$ channel. Lower bit rates may enable both voice and picture to be transmitted down a single $B$ channel [58].

### 2.6.1 Model-based coding

Considerable effort is being directed towards codecs which are not simply based on statistical techniques. Hybrid codecs use a combination of techniques which are all aimed at reducing the correlation in the transmitted data. By observing which moving features of a particular scene are of interest to the observer large reductions in the required bit rates can be achieved. For example in the case of videophony, interesting features might be the eyes, mouth and the orientation of the face. These features are modelled so as to fit a 3-D wire frame [59] representation of the head and shoulders at the source coder. Parameters which capture the changes in these features of interest are derived for each frame of the incoming images. These are *analysed* and coded as the up to date information and sent to the decoder.
At the decoder the same 3-D model of the person's head and shoulders is known. Onto this model are superimposed the update parameters as they are received. In this way an acceptable image of the person talking at the other end can be synthesized using computer graphics rendering techniques [60]. This technique is known variously as model-based coding (MBC), knowledge-based coding (KBC) or analysis-synthesis coding. The bit rates produced are much lower than those of hybrid coders and so MBC could be used for ultra-low bit rate coding or alternatively high-image-quality ISDN-rate coding. A realistic image of a person talking can be transmitted using a few hundred bits per second using MBC [61].

The advantages of MBC do not come without difficulties:

- The extraction of the 3-D model from the incoming 2-D images;
- Locating the relevant features in the image [62, 63];
- Unknown objects entering the scene cannot be modelled.

This field is a very new one. However, the results produced so far look extremely promising [64, 65, 62, 66, 67]. Further possibilities lie in the combination of MBC
with conventional hybrid coding such as that used in H.261 [68] or the use of feature codebooks in place of the 3-D wire model [69].

2.6.2 Fractal coding

At present there is great excitement about the field of fractals [70]. Compression ratios of 10,000:1 have been reported for still frames of natural scenes [71]. The coder works by splitting the image into non-overlapping blocks of two different sizes. A codebook of the larger size blocks is created and functions in a similar way to the codebook used for vector quantisation. A set of affine transformations is employed in the coding process such as rotation, reflection, scaling and translation. To encode each small block of the image a set of these transforms is applied to the codebook of larger blocks. The combination which provides the minimum error from the actual small block is parameterised for reconstruction of the smaller block. This technique can be applied recursively to encode more detail of the image [72].

Using fractal techniques extremely high compression can be achieved [73, 74, 75]. Since the coding is so simple and recursive in nature, decoding can be performed at a very high rate. However, the problem with applying fractals to the area of real-time motion video coding is that, as yet, the encoding process is slow.

2.6.3 Segmentation

One of the problems with transform coding techniques such as the DCT is that edges become blurred at low bit rates due to the high-frequency coefficients being destroyed. Another problem is that the semantic properties of the images are ignored by using fixed blocks whose boundaries do not coincide with those of objects in the scene which results in blocking artifacts. There is a technique called image segmentation [76] which preserves edges regardless of how low the transmission
bit rate is. The idea is to segment the image frame into areas of the same pixel intensity. The region boundaries are coded and transmitted along with the region intensities. More important areas of the image, such as the face, can be identified and coded more accurately [77, 78, 79].

2.7 Implementation of video codecs

Perhaps one of the most significant factors determining the success of videophony is how easily and cheaply the codecs can be implemented. Essentially there are two possible approaches: either the functional blocks can be built from ASICs and discrete electronic components or a combination of powerful digital signal processors and the appropriate software can be used. Each method has its own attractive attributes: digital signal processors are more flexible due to the software and require less development time; ASICs are cheaper in the long run [80].

Towards the end of 1990, several companies announced chip sets for image and video compression. Most conform to one of the three standards mentioned in Section 2.2.3.

2.7.1 Application specific integrated circuits

State-of-the-art high density, low power CMOS VLSI provides just the technology required for producing ASICs for videocodes. Functional blocks of the H.261 algorithm are ideal candidates for implementing in silicon — e.g. DCT, MC, VLC.

British Telecom are producing an implementation of the H.261 algorithm which is marketed under the name VC2100 [61]. It is a full implementation and includes space for extra cards to perform such functions as encryption for privacy. In 1987 their video codec for the old $p \times 384$ kbit/s bit rate standard required four boards
of semi-custom components. The new $p \times 64 \text{ kbit/s}$ bit rate codec requires less than one board using 1 $\mu$m ‘sea-of-gates’ VLSI technology [81].

**Inmos A100 DSP range**

Inmos is currently producing a range of digital signal processing (DSP) devices designed for applications such as radar, sonar, communications, video-phones, robot vision and high definition television (HDTV). These include the IMS 100 Cascadable Signal Processor, the IMS A110 Image and Signal Processor and the IMS A121 2-D Discrete Cosine Transform Processor.

**Discrete cosine transform**

The availability of real-time DCT chips will have impact on any future designs that will require efficient image compression. Other than videophones, these will include ISDN terminals, video discs, CD-ROMS, HDTV, digital video tape recorders (VTR) and still picture compression such as facsimile.

New proposals for VLSI implementation of the DCT are appearing regularly [82, 83]. Interestingly enough, most of the implementations are not based upon the classic fast algorithms used for software implementation. This seems to be due to the fact that the reordering of data required is too difficult on chip compared with a pipelined implementation of a straightforward matrix multiplication.

The first company to make a commercial real-time 2-D DCT integrated circuit was SGS-THOMSON with their STV3200. It is a dedicated DSP chip that performs forward and inverse DCT at pixel rates of 13.5 MHz. There are seven different possible input block sizes ranging from $4 \times 4$ to $16 \times 16$.

A faster DCT processor is the inmos A121. This device is a 2-D DCT processor specially designed for the image compression market. It is not coincidence that
the specifications of the chip fit exactly those proposed by the CCITT for video codecs for the ISDN network. The A121 processes fixed blocks of $8 \times 8$ pixels and performs four functions: DCT, inverse DCT (IDCT), 2-D linear filter or matrix transposition. All these functions are performed by simple matrix multiplication and transposition since this maps more easily onto silicon than any of the many proposed 'fast' algorithms. The device is fast enough for a single device to be multiplexed to perform all the required functions. Also the A121 contains post-addition and pre-subtraction to facilitate the addition and subtraction of previous frames whilst reducing component count. The maximum clocking rate is 20 MHz, which means that a block is transformed in 3.2 $\mu$s.

Motion compensation

Possibly the most computationally intense part of present-day image coding algorithms is the motion compensation (MC). It is expected that in the near future companies such as inmos will produce chips dedicated to performing MC in real time. Many papers are appearing on the subject containing suggestions for new efficient ways of implementing MC in silicon [84, 85, 86, 87, 88, 89].

2.7.2 Digital signal processors

Compression algorithms such as H.261 require tremendous DSP power. Currently the implementation of such functions as transform coding and motion compensation in real time is beyond the capabilities of programmable DSP (Digital Signal Processing) chips such as the Texas TMS32000 and Motorola 56000 families [90, 91]. It would be possible to build codecs using multiple DSPs in parallel. However the design effort would probably make other approaches more desirable [92]. The very latest member of the Texas Instruments TMS320 family,
the TMS320C40 [93], is a very powerful DSP device with on-chip communications links. It is designed to be programmed at a high level in such languages as C, C++, Ada, and Fortran, and incorporated into multiple processor architectures. Such a device would be ideal for the implementation of video codec algorithms.

The new generation of programmable DSP devices are powerful enough to handle either the encoding or decoding part of the H.261 algorithm. In this way a single PCB prototype videophone was built by British Telecom [61] (Section 2.7.1).

2.7.3 Flexibility

Only necessary details of the H.261 algorithm are actually specified in the CCITT standard. Other specifications which do not actually affect the communication between codecs at different sites are left open for innovation. For example it is not specified how the DCT should be implemented, only how accurate the results should be. Hopefully this should result in greater competition in the market place and so cheaper and better codecs being available.

2.8 Summary

Motion video coders have been considered in the context of today’s international standards. To begin with a brief history of motion video codecs was given with particular reference to the CCITT standards.

Next the problem of incompatible video formats and the proposed solution were explained. The main body of the chapter was devoted to introducing the H.261 video source coder algorithm which was the subject of simulation in the work recorded in this thesis.

In the final two sections the future codec algorithms and the problems of im-
implementing codecs in hardware were considered.
Chapter 3

Parallel computer architectures

3.1 Introduction

This chapter focuses on parallel computer architectures. Parallel architectures are changing the economics of high powered computing through the economy of scale provided by VLSI: it is cheaper to design and produce many identical processors rather than a single powerful one. The idea that processors operating in parallel can result in faster operation than a single processor of the same complexity is an obvious one, and was thought of many years ago. However, it is only now that the technology is available to make parallel computers practicable. There is a limit to how fast one processor can be made to run; internal signals are bounded by the speed of light. Integrated circuits have reached the level at which quantum effects are beginning to prevent their correct operation. Therefore parallel processing will ultimately be the way forward in designing faster computing engines. The remaining bound is the communication bandwidth between processors.

The following section traces briefly the history of computer architectures from early bit serial machines to the highly complex modern concurrent supercomput-
ers. Given that technological improvements cannot be controlled by the computer architect, he is left simply to find techniques to make more effective use of existing technology. According to Ibbett [94], almost all of the techniques can be grouped into two categories: storage hierarchies and concurrency. That which Ibbett terms 'high-level concurrency' is of interest here, as opposed to the 'low-level concurrency' found within individual processors. Section 3.3 describes the different types of parallel architectures available together with their memory organisation.

In Section 3.4, the inmos transputer family is described. This 'computer-on-a-chip' is the processor found in the parallel computer used for this project, which is described in Section 3.6. Sections 3.5 and 3.8 discuss architectures for performance measurement and signal processing respectively. Finally a summary is presented.

### 3.2 A brief history of computer architectures

Originally computers were bit-serial. An early commercial example of such a machine is the UNIVAC 1. Gradually it was realised that introducing various forms of parallelism would provide increases in throughput. First, groups of bits were processed at the same time. Later came the concept of the pipeline: sequential processing was divided into 'stages' which could be overlapped in time so that once the 'pipe' was full, results were produced at the rate of the slowest stage. This technique is extensively used in supercomputers such as the CRAY-1, and in high speed signal processors.

Next came vector supercomputers. These were still essentially serial machines but they provided special hardware for executing operations on vectors in parallel. Despite these changes the programmer experienced no great hardship. It was discovered that it was fairly easy to write compilers that could automatically vectorise
programs simply by determining which tasks were independent.

It has been realised since the time of Charles Babbage that the ultimate performance would be achieved by having multiple processors working on the data at the same time. However it was not practical to lash together machines that filled whole computing halls. The advent of the VLSI technological revolution, over 100 years later, provided the necessary scale and reliability in the form of the microprocessor. For a more detailed history the reader is referred to Hockney and Jesshope's book [95]. A more up to date survey of available parallel computing systems has recently been made by Trew and Wilson [96].

Current technology presents us with a complete computer on a chip; central processing unit (CPU), floating point unit, memory and input/output (I/O) communications links can be integrated onto a single die. The development of such chips is very expensive but this is counteracted to some extent by the fact that they are being manufactured in bulk to be used as the processing elements (PEs) in truly parallel machines.

The optimum architecture depends on the application. Thus it seems to be a definite advantage for the user to be able to specify the machine topology to suit their particular problem.

3.3 Parallel architecture issues

The design of parallel computers gives rise to a whole new set of architectural problems. These include:

- Should the memory be shared or distributed amongst the processing elements;
- How powerful should the PEs be;
• Can the data I/O be performed quickly enough;

• What should the PE interconnection network topology be;

• What kind of control strategy ought to be employed.

The further problem of how the machine should be programmed will be tackled in Chapter 4, Parallel computer programs. All the above problems are intimately related; a decision on one cannot easily be made without considering its affect on the others.

Flynn [97] classified architectures by their control strategy for data and instruction streams:

**SISD:** Single instruction, single data stream — i.e. serial machines.

**SIMD:** Single instruction, multiple data streams.

**MIMD:** Multiple instruction, multiple data streams.

This has since become standard nomenclature. Only parallel machines — i.e. computers with multiple data streams — are considered in this thesis.

### 3.3.1 SIMD parallelism

Due to the relatively large number of PEs used and their low complexity, SIMD machines are sometimes termed ‘fine-grain architectures’. Typically a master control unit (MCU) is employed to distribute instructions to multiple identical processors. The term ‘parallelism’, rather than ‘concurrency’, is used here to indicate that events occur *simultaneously* and *synchronously*. A single instruction is broadcast to all processors which then execute the instruction in *lockstep* on local data. In this way, good balancing of the load amongst the PEs is guaranteed (see Section 4.2.1).
which can be one of the main factors leading to inefficiencies with MIMD machines. The topology of SIMD machines is usually fixed and regular. The term 'massive parallelism' is often used when talking about SIMD architectures. This reflects the usually large number of richly connected low complexity (or fine grain) PEs. Example SIMD machines are the Connection Machine [98] with up to 65,536 PEs and the AMT Distributed Array Processor (DAP) with either 1024 or 4096 PEs [96, 99] shown in Figure 3.1.

Originally the DAP was designed in 1970 at ICL, UK, and as such was one of the first commercial parallel computers. Now, however it is manufactured by Active Memory Technology, Inc., in the USA. The PEs are one bit processors arranged in a square grid (32 × 32 or 64 × 64). The MCU is connected to the program memory, the PE array, and the host. The host might be a workstation or a mainframe on which the user can debug their programs. The MCU takes instructions from the program memory and broadcasts them to the PEs. A fuller
description of the DAP range of machines is given by Trew and Wilson [96].

Another example of an SIMD computer is Thinking Machines Corporation's Connection Machine (CM). The CM comes in two sizes containing either 16,384 or 32,768 PEs. The latest CM, known as the CM-200, has a peak performance of 8 GFLOPS.

The major constraint of SIMD machines is that all the PEs must execute the same instruction. They do not perform well if different algorithms need to be applied to different data. The future generation of CMs, known as CM-5, promises to be a hybrid design of SIMD and MIMD and so will be a much more flexible architecture. The major advantage of SIMD computers is that they are very similar to sequential machines to program; 'for' loops are simply replaced with 'for-all' loops which execute in parallel. The single instruction stream allows the number of PEs to be varied without affecting the program.

3.3.2 MIMD concurrency

By contrast, the PEs in MIMD machines are more complex (and hence more expensive) and fewer. Thus the term 'medium-grain architecture' is applied to MIMD computers. This is the most general type of architecture. MIMD machines are asynchronous, hence the term 'concurrency' instead of 'parallelism'. Multiple PEs execute independent instructions, and the PEs are highly complex (coarse grain). Examples of MIMD machines are the transputer-based computers now available, such as the Meiko Computing Surface (see Section 3.6) or the ESPRIT\(^1\) project Supernode [100].

There are two categories of MIMD architectures.

\(^1\)ESPRIT: European Strategic Programmes for Research and development in Information Technology.
Loosely coupled MIMD machines have memory distributed amongst the processing elements. Each PE can only access its own local memory. All non-local data must be obtained through communication links. Hence these loosely-coupled MIMD computers are sometimes called ‘message passing’ machines. They were developed in the 1980s to satisfy demand for an architecture that will ‘scale’ — i.e. the number of nodes can be increased or decreased easily. The PE network topology of such machines is often configured as a ring, mesh, tree or hypercube. Some recent ones, such as the Meiko Computing Surface (see Section 3.6), are logically reconfigurable.

Tightly coupled MIMD machines by contrast employ shared memory. They tend not to be as favoured as message passing machines since they suffer from problems with memory access collisions and data security. Additionally, they do not scale as easily since adding more PEs just aggravates these memory problems. The theoretical advantages of shared memory can be provided on distributed memory machines by software systems such as Linda (see Section 4.2.3).

Obviously there are many possible architectures made from the various combinations of the features described. Interestingly enough though, Roberts has proposed that just two are selected for future research, thus simplifying software development [100]. The architectures he chooses are those of the AMT DAP and the Meiko Computing Surface².

²By coincidence, the University of Edinburgh currently has two of each of these machines.
3.4 The inmos transputer

3.4.1 The transputer family

Inmos have produced a range of 16/32 bit microprocessors [101] which can either be used as single powerful processors or combined into modular systems. The transputer is a ‘computer-on-a-chip’ — i.e. a single device consisting of a CPU and enough program memory to hold a complete small program. It combines a central processing unit, four bi-directional serial communications links, and memory in a single device. A reduced instruction set computer (RISC) [102, 103] design philosophy was employed which resulted in the CPU requiring only 30% of the silicon area. This left the remaining room for memory, memory interface units and the four serial links. The T414, the basic integer processor transputer model, was released in 1985. The floating point version, the T800, appeared in 1987 (see Figure 3.2).

Context switching\(^3\) is extremely fast. This is important for executing processes in parallel. There are 4 kbytes of on-chip memory and up to 4 Gbytes of external memory can be addressed. Serial communication between transputers is provided by four 20 Mbit/s links. These links are autonomous resulting in 80 Mbit/s bandwidth at the same time as CPU and floating point unit (FPU) operations are being performed.

3.4.2 T800: the floating point transputer

The T800, in addition, boasts an FPU on the same piece of silicon making for extremely fast execution times. It executes concurrently with the CPU.

\(^3\)The actions performed by the CPU when changing from executing one process to executing another — e.g. preserving registers and the stack frame.
Figure 3.2: Inmos IMS T800 architecture
The Queen's Award for Technological Achievement 1990 was conferred jointly upon Oxford University Computing Laboratory and inmos Ltd, for the development of formal methods in the specification and design of microprocessors [104]. It has been estimated that a year of development time was saved by the use of formal methods in the design of the T800 FPU. The usual nonexhaustive testing techniques involve comparing the output of the new FPU with the commercial designs of competitors. Formal methods even allowed some errors to be discovered in the FPU designs of competitors.

This FPU was simply added on to the integer T414 design previously released. The T414 design was rather rushed to meet the release date of the awaited first transputer, whereas more time was spent on the design of the T800's FPU. A rather surprising side-effect was that the floating point multiplication and divide have faster execution times than their integer counterparts. A convenient consequence of this is that programmers do not necessarily have to recode their algorithms in integer arithmetic to gain ultimate speed.

3.5 Performance measurement

Manufacturers' measures of performance have always been of doubtful use to users of their machines. Usually a maximum attainable number of FLOPS (floating-point operations per second) is quoted rather than a typical sustainable average. Another measure, MIPS (millions of instructions per second), is also difficult to interpret, especially now that many processors are of RISC design. RISC design employ a small number of instructions which execute quickly. Operations which take a single instruction on a conventional CPU may require several on a RISC CPU. Therefore RISC CPUs generally perform a higher number of MIPS than con-
Conventional CPUs do. Inmos claim peak performance of 4.3 MFLOPS and 30 MIPS for the IMS T800. Results reported for the Edinburgh Concurrent Supercomputer (Section 3.7) claim between 0.6 and 1 MFLOPS actually sustainable per node [105] depending on which programming language is used.

Both of these measures are even more confusing when they are applied to parallel machines. This is due to the difficulties of writing an efficient parallel program. Rarely can the full potential of the hardware be exploited due to the difficulty of understanding the intimate link between algorithm, hardware and programming language. One thing that is certain is that the cost per MFLOP of a parallel supercomputer made from mass-produced VLSI PEs is much less than that of a vector supercomputer.

3.5.1 T9000: the new generation

It has become apparent that the T800 has some severe limitations. These will be described in the rest of this thesis. They arise due to the fact that the T800 only partially embodies the Communicating Sequential Processes model (see Section 4.2.2). Ideally, an arbitrary number of processes can be created and interconnected by any number of channels. In reality, channels between processes on distinct processors must be placed onto hardware links of which there are only four on each transputer. Thus channels must be software multiplexed onto links. If processors which are not connected need to communicate, then software protocols must be organised to route messages through other processors. This can result in highly inefficient programs. The next generation of transputers, called the T9000, addresses this and other problems.

Several hardware enhancements are incorporated for faster operation and to provide support for better operating systems. The goals were an order of mag-
nitude increase in both processor and communication speed. The most obvious way this will be achieved is by using a 1.0 μm three-level metal CMOS fabrication process that will enable the processor clock speed to be doubled to 50 MHz. Predictions of 100 MIPS and 20 MFLOPS peak are being made by inmos. Perhaps greater than 10 MFLOPS sustained performance can be realistically expected. Multiplexing hardware is provided to enable the hardware to provide arbitrary numbers of virtual channels on the hard links. These links will still only number four, but will be five times faster than those of the T800, providing 100 Mbit/s full duplex. The solution inmos propose to the communications problem involves a high performance routing chip, C104, which, working with the hardware multiplexers of the T9000, will provide a packet-switching network.

The C104 packet-switch exchange chip is essentially a 32 x 32 crossbar switch of transputer links together with some logic circuits but without a CPU. It can either connect transputers together or else connect to other C104s to make larger networks. These chips will provide the virtual channels in a way which is transparent to the programmer. Messages will be sent in packets of 32 bytes rather than on byte-by-byte basis which reduces the number of message start-up and acknowledge overheads. A new routing algorithm is to be used called 'interval labelling' which is deadlock free, fast and scalable. More details can be found in the preview articles [106, 107, 108]. Other chips will be provided for linking the new generation transputers to the old, and also to other peripheral device and bus standards.

By providing this packet-switched network at the hardware level, the communications are effectively separated from the programming. There will be no need for the programmer to physically place his channels on links and processes on processors. Both of these actions will be provided by the compiler resulting in topology independence and hence greater portability (see Section 4.2.3).
Though no T9000 devices will be available for some time (perhaps late-1992), simulations show that for a hypercube network, the average message delay for 16,384 nodes is barely double that for 64 nodes. This indicates that the T9000 and the C104 will together produce a step in the direction of the truly general purpose parallel computer (see Section 3.9). It is also quite likely that this chip-pair may be used as the basis for future multimedia workstations which will complement B-ISDN (Section 1.1). These are single workstations which will provide the user access to video and audio I/O as well as access to peripherals, local area networks and conventional terminal services.

3.6 The Meiko Computing Surface

All of the parallel simulations for this project were carried out using a T800-based machine called a Meiko Computing Surface. Meiko Ltd. was formed by previous employees of inmos in 1985. They designed a distributed memory multi-processor computer based on the transputer and called it the Computing Surface (CS) [109]. It is a modular system built to the specifications of the customer by plugging combinations of expansion cards into a backplane.

In terms of computing power it is in the supercomputer league. However, its main attribute is its adaptability. The classic problem of 'optimal topology' is avoided by making the CS electronically reconfigurable. Thus the topology of the processors can be adapted to any application by means of a software reconfiguration tool. At present the topology is statically configured before the software is loaded onto the transputers. Research is underway to develop dynamically reconfigurable machines [110]. It will be possible for the topology to change at run-time according to the needs of the various stages of the program.
By selecting from the elements available, the user can make up the machine which best suits his requirements. The elements include:

**Local host:** Controls the supervisor bus

**Compute:** The number-crunching power

**Mass store:** 8 Mbytes RAM plus disc access

**Display:** Video RAM and RGB video output

**Data port:** High speed data link for external devices

**Frame grabber:** Connects video camera to data port bus

The separate elements are plugged into a card cage. A populated cage is called a *module*. In turn, modules can be connected together, thus harnessing even more power.

### 3.6.1 The supervisor bus

There is a central monitoring system which is known as the *supervisor bus*. This is a low data rate bus which is globally shared. It is not part of the application since it is shared by all users and hence is fairly slow. Rather it is used to provide:

- A route for debugging messages;
- Software and hardware failure detection and reporting;
- Electronic reconfiguration.

It is essential to have a route for debugging messages separate from the transputer links in case communications fail due to a communications related fault or bug. Each element contains a single transputer. The electronic configuration of
these elements is provided by the supervisor bus. The user must specify how they are linked together using an electronic reconfiguration tool.

3.6.2 Virtual computing surfaces

A number of virtual computing surfaces, called *domains*, are derived from the CS by means of system software called the Meiko multiple virtual computing surfaces (M²VCS). Each user is also given the illusion that he has his own CS and supervisor bus. M²VCS links the domains to the filestore and the terminals (see Figure 3.3). Typically a user will use a small domain with just a host seat and a single processor for developing his multi-processor software. Once the program is debugged, it can be moved to a larger domain with many processors in order to execute concurrently. Thus the large domains are more efficiently shared. In the future it is hoped that domains can be allocated dynamically from a pool of PEs according to the needs of users as they arrive.
3.6.3 New directions

In the 1990s, Meiko offers a new range of CSs [111]. The computing surface network (CSN) is a software layer which allows the software reconfiguration of the transputer link network. This has been developed into an interface between different processors. Other high performance processors like the Intel i860 and the Cypress Technologies SPARC have been integrated into the CS architecture [96, 108]. Thus it has become possible for the industry standard UNIX operating system to be ported to the CS, allowing users to program in C. Further software called CS Tools (see Section 4.2.3) has been developed to sit above the CSN and make the CS easy to use regardless of its configuration. The result is a resource that the user can tailor according to his vector or scalar requirements. Meiko plans to incorporate new processors as they become available.

3.7 The Edinburgh Concurrent Supercomputer

The Edinburgh Concurrent Supercomputer (ECS) is a Meiko Computing Surface. Originally, the ECS aimed to house 1024 T800 when complete. At present (November 1991) there are over 400 distributed amongst the single user domains. Domains vary in size from the smallest, with simply a local host transputer, up to the largest, with a local host with 131 general purpose T800. Domain sizes have been changed according to user demand and this current configuration seems fairly stable.

The user of the ECS has a choice of operating systems (OSs): the occam

---

4It now seems that the ECS will not expand much further due to the fact that the T800 technology is being superseded and Edinburgh University has acquired the new 'Grand Challenge' machine. This is a 64 node Meiko CS which can provide a peak performance of 5 gigaFLOPS (GFLOPS). Each node contains one Intel i860 processor with 16 Mbytes of RAM and two T800s just for communication.
programming system and more recently, MeikOS. MeikOS is based upon AT&T System 5.3 and Berkley 4.3 UNIX and so is the favoured familiar environment of most users.

### 3.7.1 Graphics domains

Of particular interest to this project are the *graphics domains*, i.e. domains which contain a display element. The display element comprises:

- a single T800 processor;
- 2 Mbytes video dual ported RAM;
- 4 Mbytes program RAM;
- a 200 Mbyte/s pixel bus.

There are three different sizes of ECS graphics domain. Each has a single display element and a local host plus a varying number of T800s: 5, 17 and 65. Note that these numbers are all some power of two plus one. This is because of the commonly used control scheme of one master processor and then some power of two configuration of worker processors.

For most applications, one display element per graphics domain is quite adequate. This was not found to be the case, however, during this work. A detailed explanation of this is given in Section 5.4.2. It possible to place more than one display element within a single domain. Each would provide graphical services for its own section of the display. However, the graphics domains are in high demand and so reconfiguring the ECS domain structure so as to loose a graphics domain was not considered justifiable.

---

5 Meiko's version of inmos' transputer development system (TDS)
3.8 Parallel architectures for signal processing

The T800 has proved most useful in embedded systems; most of the sales have been in this area. Let us consider the suitability of the T800 for signal processing. Studies have shown that the T800 compares very favourably with other available processors in the field of digital signal processing (DSP) [112, 113]. Obviously, it will not perform as quickly as dedicated DSP chips but they are more flexible and versatile. A most impressive result is that the T800 can perform at 30–80% of the speed of a VAX 8650 on floating point operation intensive applications, and yet it only costs a fraction of the price.

Undoubtedly, specifically tailored architectures will always out-perform general ones. For example the Princeton Engine, which the US NIST\(^6\) is about to acquire, has gigabytes of I/O bandwidth and 1024 PEs which can each process a pixel in parallel. It will be ideally suited to the image processing applications for which it is intended. However, the cost of such systems is prohibitive.

3.9 General purpose parallel architectures

Over the 1980s, a wide variety of parallel computers have been introduced [96]. The major obstacle to the exploitation of the potential benefits offered has been the cost of developing software for them. This in turn is due to the lack of standard development environments. Without these, most parallel applications, such as the one described in this thesis, have been developed from scratch. It is clear that there is a need for a general purpose parallel machine as proposed by Valiant [114]. A new ESPRIT project was begun in 1990 called GP MIMD. It is aimed at developing a standard parallel architecture and applications support interface (ASI). A family

\(^6\)United States Department of Commerce National Institute of Standards and Technology
of open architectures is being designed, based upon the T9000 processor, with performances ranging from workstation up to supercomputer.

3.10 Summary

An overview of parallel computer architectures has been given. Specific difficulties have been alluded to and their affect on the two basic categories, SIMD and MIMD, discussed. The inmos transputer and the Meiko Computing Surface which were used throughout this work were described in some detail. The next generation of transputers was considered and how they will shape the parallel computers of the future. Clearly it will shortly become much easier to write software for parallel computers.

Finally, architectures specifically designed for DSP were considered and in so doing, the T800 was compared with other available processors.

This thesis will now cover the use of T800 arrays to simulate the performance of motion image coders, emphasising the processing speed which can be achieved with arrays of different sizes.
Chapter 4

Parallel computer programs

4.1 Introduction

Computer software solutions can be viewed at different levels of abstraction and complexity: conceptual, algorithmic, implementation and physical [115]. Figure 4.1 shows how the levels relate. The objective of this chapter is to consider the implementation level. However, the physical level must also be partly considered, i.e. how the processors are connected and how quickly their interconnecting links can operate. Concurrent computers are economically very attractive because of their small cost relative to the performance they can achieve. An extremely powerful multicomputer (described in Chapter 3) is available for use. The problem of how to harness that power and optimise processing speed is examined in this chapter.

There are many new aspects the programmer of parallel machines must consider in order to write efficient programs. These include communications overheads, processor idling due to synchronisation with other processors, and balancing the load between the processors. The following section describes in detail the difficulties
Figure 4.1: Solution hierarchy.
encountered\textsuperscript{1}. Section 4.3 presents currently employed methods of exploiting parallelism. General purpose parallel computers are discussed in Section 4.4. Section 4.5 is a summary of the chapter.

4.2 Difficulties of programming parallel computers

Parallel programs present a new dimension of difficulty to the programmer. The situation was succinctly summarised by McGraw and Axelrod:

The behaviour of even quite short parallel programs can be astonishingly complex. The fact that a program functions correctly once, or even one hundred times, with some particular set of values, is no guarantee that it will not fail tomorrow with the same input. \cite{116}

Much of the vast body of knowledge and tools amassed over the first forty years of computing history no longer applies. For example a clever serial sorting algorithm might prove impossible to implement concurrently. In addition to usual requirements of sequential programming, the programmer has to organise: data distribution to the PEs; distribution of the calculations amongst the PEs; communications between PEs; PE network connectivity. As a result, it is common for the programmer to spend more time managing the parallel resources than actually solving the problem. The ratio between the performance of a good and a bad parallel program is considerably larger than that of serial programs. It has risen from typically 2 or 3:1 to perhaps 10:1 or more \cite{95}.

An implementation whose execution speed is dominated by communications overheads is termed \textit{communications bound}. One whose execution speed is pri-

\textsuperscript{1}Only parallel aspect of programming will be considered; it will be assumed that all sequential code is optimally implemented.
manly determined by the processing power of the PEs is termed *computations bound*.

### 4.2.1 Load balancing

For maximum efficiency optimal use must be made of the resources available. There is little use in having many processors if only a few of them are doing useful work at any given time. The work should be distributed amongst them so as to keep all of them busy all of the time if possible. This is called *load balancing*. Several schemes are proposed to ensure good load balancing. These schemes can be classified as static, dynamic and adaptive [117]. The specific details of load balancing are made clearer in Chapter 5 where the implementations are described.

### 4.2.2 Communications

Probably the most important of the new aspects to consider is communication between PEs. PEs in distributed memory machines have distinct address spaces and hence must exchange information by communication. Hoare developed a model for this called Communicating Sequential Processes (CSP) [118] upon which the *occam* programming language is based. In this model the processes which constitute a program communicate via *channels*. The channel between two processes is only used for communications between those processes. Each act of communication forces the processes involved to synchronise, perhaps preventing one from doing useful processing while it waits for the other to be ready to communicate. Processes are not allowed to share data even if they reside on the same PE. This can lead to unnecessary copying of data and hence inefficient programs.

The transputer is also closely based on the CSP model. However, since the number of links is limited, it may not be possible for processes to be directly
linked via channels if they do not reside on directly connected PEs. Instead, communication protocols must be devised and implemented by the programmer. When the number of communicating processes becomes large, the handling of channels can become a daunting task. For these practical reasons a better communication model is one which addresses messages by destination process rather than channel name [119]. Tiny is a message passing system or software router which supports such a model of communication (see Section 4.2.2). Another solution which shortly will be available is the use of hardware routing chips [120].

Implementation grain size

Selecting the size of the portion of the problem which is allocated to each processor is critical to the efficiency of any parallel program. These portions are known as grains, and solutions are said to be of fine or coarse granularity if the grain size is small or large, respectively (Figure 4.2). Intuitively, one expects fine grained concurrency to deliver faster results whilst using more processors. However, for message passing machines, there is a point at which the additional communication outweighs the expected benefits. A small grain size can lead to better load balancing since hopefully the more computationally intense grains will be processed by distinct processors. The selection of an appropriate grain size is both application and machine dependent.

Image processing may fall into any of the above grain-size categories. For example, a fractal may be generated by assigning a single pixel to each PE; an image spatial transform might be achieved by distributing blocks of between 10 and 100 pixels to the PEs; for edge detection, it may be more appropriate to

\[\text{In practice an educated guess can be made and then empirical experiments will provide the best result.}\]
process larger segments of the image on each processor. The most appropriate grain size must be determined for each particular application.

**Topology independent message router**

A very efficient message router, known as *Tiny* [121], was developed at the University of Edinburgh on the ECS. *Tiny* functions on any processor array topology allowing the user to experiment easily with various transputer configurations and process mappings. Message routing is optimised for small messages so as to allow high granularity to be exploited in applications, thus leading to better speedup and load balancing. *Tiny* is supplied as a compiled *occam* library. User processes are coupled to *Tiny* by means of an *occam* harness. It can currently be used to link processes written in *occam*, Fortran, or C.

The important features of a good message router are:

- Communication times grow slowly with the number of processors, \( n \), *i.e.* \( O(\log n) \);

- The routing is parallel — *i.e.* alternative routes and adaptive strategies are
used under conditions of high network loading;

- Non-local communications are fast so that the program can be ported to different topologies and still run efficiently;

- Communications are not forced to be synchronous so that unnecessary delays are removed.

_Through-routing time_ is that time 'wasted' by a PE when it has to stop doing its own useful work and organise for a message to be forwarded. By using a combination of occam, C and transputer assembler, together with much low level expertise, the through-routing time has been reduced to an astonishing 20 μs.

_Tiny_ provides the illusion of asynchronous global connectivity. Thus any processes can communicate via the router without having to decide which route to take. Since the connection is asynchronous, processes do not waste time waiting for end to end connectivity. The message is simply handed to the router by a process, after which the process can continue to work leaving the task of delivery to the router.

_Tiny_ has a process running on each processor. When it starts to run, it first explores the topology and builds routing tables. The result of this is that the router process on every node knows the possible routes to all user processes. Thus we have topology independence (see Section 4.2.4); neither the router nor the programmer needs to know the topology before the program is loaded and executed. Each user process has a unique identification number known as its process ID. When a process wants to send a message to another process it needs to know only the destination process ID.

There are three routing strategies provided by _Tiny_.

- Adaptive routing takes the shortest route, taking into account the congestion
of the links. Thus messages may not arrive in the order they were sent, but the average communications delay may be decreased.

- Sequential routing uses a single shortest route so that message order is preserved. This can be useful where deterministic program behaviour is required, or for debugging purposes.

- Broadcast routing uses a tree route to all PEs so as to avoid unnecessary message duplication.

Technological bandwidth limitations

Ideally, the programmer would like a completely connected network with an infinite number of processors (see Section 4.4). Any application could be run on just a small part of such a network. Unfortunately the number of connections increases as the square of the number of processors. Thus VLSI sets technological limits on the actual machine architectures available. Currently communications technology is lagging behind processing technology. Parallel programs would nearly always benefit from higher link bandwidths if they were available. The result is that the time to exchange data between PEs dominates the actual calculation time. Thus, for a communications bound application, an increase in the number of PEs might actually result in an increase in execution time. This problem is partially removed by the transputer which has autonomous communications link processors which can run concurrently with the CPU, allowing communications and processing to be overlapped to some degree.
4.2.3 Language support for parallelism

Many standard techniques have evolved to aid the writing of software for conventional sequential von Neumann machines. Sequential languages have developed to such a level that they can almost completely hide the machine hardware from the programmer. They provide abstract data types and optimised libraries so that the programmer rarely has to leave the conceptional level of problem solving to deal with machine peculiarities. Indeed, this is the way things should be; there is no need for each and every programmer to understand how the target machine works in order to use it.

By marked contrast, programming environments for parallel machines are at a very early stage of development. Currently there are three kinds of programming language support available to the parallel programmer:

1. Those such as occam and Ada which contain truly parallel constructs;

2. Sequential languages with parallel extensions such as various parallel versions of Fortran, Pascal and C;

3. Compiled parallel message passing libraries, such as Tiny, which are independent of the programming language being used.

Each has its associated advantages and disadvantages. The first is not very popular outside academic circles because it entails learning a new philosophy of program development, and a new language structure which is often considerably larger than that of conventional sequential languages\(^3\). Also, existing programs must be rewritten in the new language and new programs cannot be shared with colleagues.

\(^3\)Occam, with its clean semantics, is an exception to this trend, developed as a reaction to the complexities of parallel programming
Language extensions have been more popular with the scientific community. Scientists seem happier to develop their programs in tried and tested sequential ways and then add in the 'parallel bits' when they are sure everything works. However, the lack of standards means that code written in this way is necessarily hardware dependent.

When parallel computers first appeared it was assumed that new parallel programming languages would have to be adopted. What is now emerging as the most popular choice of all is the third option, message passing libraries. Provided that appropriate compilers exist, the programmer can use their preferred sequential language and reap all the benefits that familiarity provides in shortening the learning curve and development times. The result is greater portability since for a new machine only the parallel libraries need to be changed.

Parallel programming aids

Present parallel debuggers are of little use to the programmer. They often provide a lot of low level information that few programmers find useful or comprehensible. Experiments are being performed with graphical debugging [122] and monitoring [123, 124] tools which will allow the programmer to 'see' what is happening 'inside' parallel programs. Formal methods are not currently available to all programmers; pencil and paper diagrams are not always practical for concurrent systems.

Increasingly, steps are being taken to hide the hardware details from programmers. They should not be concerned with, for example, how the memory of the machine is managed — whether it is distributed or shared. Distributed memory machines scale well whereas shared memory machines are more natural to use. Linda, developed at Yale University, is a model which allows the sharing of data
structures regardless of whether the machine is tightly or loosely coupled. It contains an associative memory model known as ‘tuple-space’. A dedicated, loosely coupled, *Linda* machine has been proposed to implement *Linda* efficiently [125].

One approach sometimes used is to develop parallel programs on sequential machines using a simulator. These simulators include details of the target PEs and machine, and can easily provide very high level debugging facilities since it is actually just a sequential program. Examples of simulators used for transputer networks are *Simscript II* [126] and *Transim* [127]. An additional advantage of such development systems is that users are not restricted to the number of transputers actually at their disposal. Another product, *Gecko*, which allows screen visualisation of the static structure and dynamic behaviour of programs, is marketed with *Transim*.

Parallel program development toolkits such as CS Tools (Communicating Sequential Tools) [128] from Meiko, aim to provide a cross development environment which is machine independent. CS Tools contains compilers, configuration systems and high level communications facilities. Towards the end of this project, simulations were written using the C programming language supported by the CS Tools communications library (see Chapter 5).

**Program portability**

Since the hardware technology has been available to manufacture parallel supercomputers, industry has been slow to respond. The difficulty of software portability can be pinpointed as the main reason. The investment of man hours in writing parallel programs at present does not seem worth while. Thousands of lines of serial Fortran programs have been written, representing hundreds of man years’ effort.
Compilers for standard languages such as Fortran, C and Pascal on parallel machines are emerging. However the vast differences in machine architectures is making portability a real problem. Parallel computer manufacturers are appearing and disappearing at an alarming rate [96]. Industrial progress in producing parallel software is bound to be slow until standards are defined.

Automatic parallel compilation

Vector supercomputers did not present a problem to users since vectorising compilers hid the hardware inovations. Unfortunately, parallelising programs for parallel supercomputers is a much more complicated affair.

Ideally large chunks of user code can be processed sequentially and asynchronously in a secure manner. The compiler would have to identify these sections of code which really amounts to understanding the application problem. This is considerably harder than merely searching through a piece of high level language program and picking out the vector operations. In addition to this the target machine may not have a fixed topology. The compiler would then have the job of deciding on the best topology and mapping of processes to processors.

Automatic compilation of parallel programs is an active area of research. Low level exploitation of implicit parallelism has been used on single statements with such techniques as loop distribution and tree height reduction [129]. However, the real hope lies in the detection of implicit parallelism on a higher level, i.e. on the level of the algorithm itself.

Tools are emerging which can take existing serial programs and parallelise them by dataflow analysis techniques. These, however, still rely on human interaction to achieve the best results. The portability problem mentioned above may be partially solved by these techniques. It is unlikely that totally automatic parallelisation will
ever be possible.

4.2.4 Processor network topology

The pattern of interconnectivity of the PE network in a parallel computer is termed the PE network topology. There is no 'best' topology for a general parallel computer; what works well for one application could be totally unsuited to the next. For example, a problem such as convolution of an image could be implemented well on a two-dimensional grid topology since only communication with nearest neighbours is necessary. However, if shared data structures are employed, such as a global hash table, a more compact topology would seem appropriate. Thus it seems sensible that the programmer is able to select the PE topology.

Topology dependence

Usually the links between PEs are too few to allow direct passing of messages. Often they must be routed through other PEs. This means that a process at each node must recognise whether arriving messages are to be forwarded and if so in which direction.

Not only does this make programming extremely difficult and tedious but also it makes each program extremely topology dependent; the programs cannot easily be ported to machines with different topologies. Rather than use a good topology for a particular problem, the programmer might end up choosing one for which the communications are easier to handle.
4.3 Techniques for achieving a parallel solution

Most scientific and engineering problems have some intrinsic degree of parallelism — e.g. the 'conveyor belt' of a factory simulation, or the 3-D projection and rendering of a set of medical 2-D image 'slices'. Thus it might be expected that programmers would naturally select a parallel algorithm, rather than devise a sequential algorithm to solve their problems. In most applications, however, this has not proved to be the case. Spotting the inherent parallelism in a problem is not always easy.

How should the application best be divided amongst multiple processors? All of the considerations outlined in Section 4.2 must be taken into account. The few available techniques are described in this section. First, as an aside, elements of graph theory, which will greatly assist in tackling the rest of the problem, are considered.

4.3.1 Graph theory

The essential communications characteristics of a processor network can be modelled using techniques from graph theory. There has been considerable interest in this area in the past, due to its application to telecommunications networks and distributed computer systems. If the reader is a newcomer to the field, a readable introduction is given by Aho et al. [130]. Aspects relevant to this thesis are briefly introduced below.

PEs are represented by nodes or vertices, their interconnecting links by edges. The degree of a node is the number of edges with which it is incident. Thus, for the purposes of this project, the maximum degree is four because the PEs (transputers) have only four links. Figure 4.3 shows an example topology and its
equivalent graph. More formally, the graph can be denoted $G = (V, E)$ where $V$ is the set of vertices and $E$ is the set of edges, such that edge $(x, y) \in E$ denotes that vertices $x, y \in V$ are connected.

Various terms are borrowed from the grammar of graph theory:

**Distance:** It can be shown empirically that the time, $T$, for a message of fixed size, $w$, to travel between two PEs in a network is directly proportional to the number of links traversed called the *distance*, $l$:

$$T(l, w) \propto l \quad (4.1)$$

**Bottle-neck:** A link in the topology which carries a disproportional amount of messages. Its performance determines the overall performance.

**Mean inter-PE distance:** The average of the shortest paths between all pairs of PEs in the topology.

**Diameter:** The largest of the shortest paths between all pairs of PEs in the topology.
Scalability: If a topological scheme can be employed for varying numbers of PEs, \( n \), and the resulting topologies supply performance proportional to (or almost proportional to) \( n \), then the scheme is said to scale well.

Narrowness: The worst ratio of communications to computation power. An indication of whether certain links will have to do a lot of through-routing for other PEs.

The terms topology and graph will be used interchangeably throughout the rest of this thesis.

Graph theory is a useful tool for describing networks. It has often been employed in strategies for network interconnection (e.g. [131]). Extensive use of these techniques is made in Chapter 5 where the implementations under consideration are presented.

4.3.2 The mapping problem

Let it be assumed that the problem in hand has been analysed in the conventional way and several modules have been produced which will represent processes in the program. Next, these modules need to be assigned to PEs in such a way as to make the overall program efficiency as high as possible. This assignment involves placing application processes on a given processor array and, if necessary, placing their communications channels on the links between processors. Determining this assignment is known as the \textit{mapping problem} [132]. At present this is a non-trivial problem which must be considered \textit{each time} a new parallel program is written.

Both the PE topology and the application program can be thought of as graphs. In solving the mapping problem, the aim is to match these two graphs as closely as possible. In this way, unnecessary communications are minimised and computa-
tional load is balanced. In the days of fixed topology machines this was the whole of the mapping problem. Nowadays, topology optimisation may be considered separately from the process mapping optimisation (see Section 4.3.4). Mapping and evaluation may be performed for each of the topologies under consideration. It is usually assumed that the mapping is static. However, there are instances in which a dynamic approach has been adopted [133] since communications patterns may change throughout the execution cycle. In addition this reflects the fact that parallelism has been shown to be distributed extremely unevenly throughout scientific applications [134].

4.3.3 Mapping models

Models have been formulated in an attempt to solve the mapping problem e.g. [135, 136, 137]. It is assumed that the overall execution time is to be minimised. Each process is labelled with an execution time or cost of computation. A process graph is constructed where the nodes are processes and the edges represent communication between process. The edges are labelled with the volume of the communication they represent.

For a given mapping, the computational load of a PE is the sum of the computation cost of its assigned processes. Inter-PE communications are summed in a similar manner. The mapping problem is now reduced to the task of finding a function which maps the process graph to the PE graph so as to minimise some cost. The cost which is to be minimised is decided by the programmer and might be, for example, total inter-PE communication.

More subtle variations on the model described above which include such aspects as the phase of communications have been proposed, but none of the proposed models capture all aspects of the mapping problem [138]. For example, process
idle time whilst waiting to receive messages is completely ignored since there are no precedence relationships captured in the model. Worse still, even if it is decided to apply a mapping model the problem has been shown to be equivalent to the unsolved 'graph isomorphism' problem from combinational mathematics [132]. This problem has been classified as \textit{np}-complete [139]. This means that there is no solution that will complete in time proportional to some polynomial of the problem size; the difficulty of solving it rises exponentially with \( n \), the number of processors. Thus programmers are forced to rely upon heuristically optimised solutions — \textit{i.e.} ones which are good but are not guaranteed to be the best [140, 141].

### 4.3.4 Optimisation of the mapping

Having chosen a mapping it would be convenient if optimisation techniques could be applied to improve the mapping or at least determine how good the mapping is. Finding the true optimum mapping or topology is known to be difficult to solve for large numbers of PEs as discussed in the previous section. Iterative improvement schemes become trapped in local minima and so are not very useful (\textit{e.g.} [136]). Two other techniques have been applied. These are simulated annealing and evolutionary or genetic algorithms.

**Simulated annealing**

Simulated annealing [142, 143] is analogous of a well known effect observed when matter changes form. If a liquid is rapidly cooled, eventually crystals form containing many imperfections. A perfect crystal can be thought of as having minimum energy. Crystals containing imperfections have higher energy-states. In general, the more slowly the liquid is cooled, the more perfect the resulting crystals.

When simulated annealing is used as an optimisation technique, a 'temperature'
and ‘cooling interval’ are used along with a function to be minimised. A search loop is entered in which the function is allowed to both increase and decrease, hopefully allowing local minima to be avoided. As the ‘temperature’ is decreased, the probability of allowing the function to increase is gradually reduced to zero. When the ‘temperature’ is zero, the optimisation is complete.

This method does not guarantee to produce the global minimum but usually finds very good ones. The main problem with the technique is it takes a long time.

**Evolutionary algorithms**

An intuitively appealing set of heuristic solutions to the mapping problem fall under the collective title of evolutionary or genetic algorithms [144]. The basic idea is to mimic the famous theory of natural selection first proposed by Charles Darwin in 1859 [145].

First any naïve but valid solution is selected. Small random changes are made to the original solution and in this way a ‘population’ of solutions is generated. The random changes provide the mutants observed in natural populations.

Next a breeding loop is entered in which the fittest survive. Pairs are selected from the population and used to give birth to new solutions which have characteristics of both parents. Using some kind of cost function, each solution is tested for fitness and only the better cost solutions are kept. This loop is iterated until a stable ‘best’ solution emerges.

At this stage, one local minimum cost solution is known. But how are we to know that it is not an evolutionary dead end? To avoid this possible scenario, the equivalent of an environmental disaster can be applied. A random number of mutations are made and then the breeding loop entered again. If the same solution occurs then it is assumed that it is the best the algorithm can provide.
Evolutionary methods have been used both for topology [146] and process mapping [147] optimisation.

### 4.3.5 Stereotyped methodologies

The mapping models discussed in Section 4.3.3 do not match reality very closely and are also difficult to apply. Instead programmers have come to use stereotyped solutions and which they later try to optimise using heuristics [140, 141]. Experience has shown that if we map the data space instead of the algorithm greater efficiency can be achieved, and expensive and complex solutions — e.g. [148] can be avoided.

Strategies for mapping a problem onto an array of parallel processors are in their infancy. To date three general categories of problem have emerged, each going by at least two popular names:

- Task farm or event parallelism;
- Geometric or data parallelism;
- Algorithmic or data flow parallelism.

Each of these types of parallel decomposition is briefly described below.

**Task farm**

The task farm is perhaps the most widely used of all parallel programming paradigms. The user has a serial program that needs to be run many times with different data sets. The only communication is the distribution of data and the collection of results. A task farm consists of a 'distributor' (or 'master') processor which farms out tasks to a pool of many identical 'worker' processors (Figure 4.4). The worker
CHAPTER 4. PARALLEL COMPUTER PROGRAMS

processor network topology can be chosen by the programmer, *e.g.* pipeline, tree or random network.

With this scenario load balancing is automatic; when a worker has completed its task it returns its result and receives another task. In this way the workers are kept busy all the time. Also communications can often be almost completely overlapped with computation to provide nearly 100% efficiency.

Examples of applications ideal for task farming are ray tracing of images [28] and calculating the Mandelbrot set [70]. In both these examples the colour of each pixel can be calculated independently of all others.

**Geometric parallelism**

Sometimes called *data parallelism*, this method relies on the data being divided into subsets which are in some sense 'close' to each other. The algorithm to be performed must only involve operations on local or nearby data. In a similar way to a task farm, each node runs the same program but in this case boundary data must be exchanged. This brings the nodes into synchronisation and thus the
MIMD machine is operating like an SIMD machine.

Examples include fluid flow simulation [149] and solving partial differential equations by finite difference methods.

**Algorithmic parallelism**

The situation here is comparable with a factory production line. All the data flows through a network of PEs typically controlled by the first which has all the data. Little memory is needed by the 'worker' PEs since the data flows through and is not usually stored.

This kind of parallelism requires irregular topologies to fit the algorithm being implemented. Thus scaling the number of PEs to achieve better performance is difficult since it is not obvious where to add them. Figure 4.5 shows an example in
which another problem arises. The summing process represented by ‘+’ requires five channels. Possible solutions are to multiplex the channels in software onto the four transputer links, or to use two transputers to create a ‘supernode’ with more available links. The latter approach is illustrated in Figure 4.5. However, it may be difficult to justify using extra processors in this way if resources are scarce. Notice also how many of the links go unused resulting in wasted bandwidth. There can be a further problem if different configurations are required for different parts of the algorithm or if many more channels are required than links are available. Algorithmic parallelism involves the simplest conversion from algorithm to mapping. However, good load balancing can be very difficult to achieve unless the functions themselves can be further partitioned into blocks of similar complexity. Typical efficiencies using this method are 50–60%. Perhaps for these reasons, examples of the use of algorithmic parallelism are few [150].

4.4 General purpose parallel computers

At the end of Chapter 3, the need for general purpose parallel computer architectures was discussed. Perhaps the main conclusion of this chapter is the need for general purpose parallel computers from a software environment perspective. People will not invest many man years of program development time until it becomes easier to write parallel programs which are portable between machines with different architectures. The GP MIMD project ASI mentioned in Chapter 3 aims to solve this problem. The commercial future of parallel computers depends upon the availability of general parallel machines in the same sense that all serial computers are based upon the von Neumann architecture. VLSI technology and routing chips are almost at the stage where such machines can be built. Hopefully this
will increase the portability of parallel programs.

### 4.5 Summary

In this chapter the general difficulties involved in writing programs for parallel computers have been considered. The biggest difficulty is that all the aspects considered (load balancing, asynchronous communications, bandwidth technology, parallel languages) are interrelated; one cannot be considered without thinking of its effect on the others. Until automatic parallelisation becomes possible, efficient implementations will continue to require a good understanding of the communications patterns of program.

In the second half of the chapter techniques for engineering a parallel solution to a problem were discussed. Some kind of topology independent router is essential in order to achieve high efficiencies, be it implemented in software or hardware.

There is a need for a general purpose parallel machine. Only then will the programmer be able to ignore individual machine nuances and achieve large scale portability of his programs. Until that time the learning curve for newcomers to parallel programming will be long.
Chapter 5

Video coder simulations

5.1 Introduction

In this chapter the actual techniques employed for simulating the H.261 algorithm are revealed. They are related to the aims of flexibility and visibility laid out in Chapter 1. It is necessary to differentiate between the conceptual level of the algorithm and the implementation level of the program. Terminology in this field is not standardised. Therefore, in this thesis, the algorithm will be said to be broken down into tasks whereas the program will be said to comprise processes.

Inevitably this chapter will include some low level details of the machine for which the simulations were written, as explained in Chapter 4, Parallel computer programs and Chapter 3, Parallel computer architectures. The approach adopted was to first develop the programs on a single PE and then experiment with the process mapping onto multiprocessor networks. Section 5.3 introduces the occam programming language, which is ideal for this style of program development.

In Section 5.2 the H.261 algorithm is broken down into tasks and Section 5.4 details the uniprocessor implementation. The timing profile thus derived, naturally
dictates the suitable choices of mapping to be used. This selection process is
discussed in Section 5.5. Having settled upon an appropriate mapping strategy,
Section 5.6 addresses the question of good PE network topologies and how to
generate and analyse them.

In late 1989, when this work began, the only practical choice of language for
writing complex programs on the ECS was occam. During the course of the project,
other languages became available and their effect on this project is discussed in
Section 5.8. Section 5.9 describes related parallel implementations and compares
them with the work of this thesis.

5.2 Problem analysis

The previous chapter made it clear that existing mapping models are not suffi-
ciently well developed to be useful. Thus the strategy adopted to parallelise H.261
was to analyse the algorithm by inspection. It was necessary to decide how the
algorithm could be partitioned in such a way as to provide good load balancing
whilst being flexible enough to allow any stage of the algorithm to be displayed if
required. A further consideration was that the mapping should minimise commu-
nication so that overall efficiency could be maximised.

The computational load of an algorithm must be known before the load can be
distributed across the PEs. Therefore, the first step was to program a uniprocessor
implementation in occam and to obtain the sequential timing profile without the
added complications of communication overheads.
5.3 Occam: an overview

One example of the truly parallel languages mentioned previously (Section 4.2.3) is occam. Work in this thesis may be of interest to those outwith the field of parallel computing so it will not be assumed that the reader is familiar with this relatively new programming language. This section introduces occam in enough detail to enable the reader with some experience of computer high-level languages (HLL) such as Fortran, Pascal or C. Those wishing to know more are referred to the excellent texts by Wexler [151] and Jones and Goldsmith [152].

The occam language was designed using the philosophy of William of Occam's razor, \textit{Entia non sunt multiplicanda praeter necessitatem,} which can be interpreted as: 'keep entities as simple as possible'. It can be argued that this is especially sensible in the parallel programming environment where it is already very difficult to prove that programs are correct. This explains why occam provides so few of the high level features that programmers are used to relying upon. Professor C.A.R. Hoare, director of the Programming Research Group at Oxford University, has expressed his concern over the possible dangers to mankind of systems programmed in complex languages:

\begin{quote}
Programmers are always surrounded by complexity; it cannot avoided. If our basic tool, the language in which we design and code our programs, is also complicated, the language becomes part of the problem rather than part of its solution. \cite{153}
\end{quote}

Examples of potentially hazardous systems include the US 'Star Wars' project and aeroplane and spaceship control systems.

A new philosophy of program development is provided, based upon Hoare's CSP model \cite{118}. A program is thought of as a process or description of the activities required. A process can be made up of other processes. Occam provides
for parallel thinking, which should result in naturally parallel problems being easier to program. Processes of the program are forced to be independent so that the classic problems of shared data are removed. Programming is transformed into the problem of making independent processes co-operate by explicit communications. In the CSP model, parallel computing is described using synchronised communication; a process cannot communicate until all its partners are ready to do so.

The current version of occam is properly known as occam 2\(^1\). This is an expanded version of the original occam created in the light of experience of the first few years of usage. Occam 2 is upwardly compatible with occam and includes data types and communications protocols.

### 5.3.1 OPS notation

Description of the processes written for the simulations will necessarily include some small fragments of actual occam text. For these the inmos occam programming system (OPS) notation for folds is used. An example is given in Figure 5.1. Whenever three dots appear at the beginning of a line, they indicate a fold which contains more lines of occam and perhaps more folds. This nested style of programming is very convenient since it hides unnecessary detail and encourages top-down programming style. Folds can be opened to view their contents and in this case, the beginning is marked with three opening braces, \{\{\{ and the end with three closing braces, \}\}\}. Unlike most modern HHLs, the layout of occam is not free format — i.e. indentation is part of the language and not simply part of the style of the programmer or convention. Instead the indentation is used to group processes

---

\(^1\)All occam programs for this project were written using occam 2 and so the shorter term 'occam' shall be used throughout this thesis and taken to mean 'occam 2'.
together. Blank lines have no effect and so can be used liberally to make programs more legible. Long lines can be broken only where it is obvious that the line is incomplete (see for example the INT declarations in Figure 5.1, in which the first line ends with a comma).

### 5.3.2 Occam semantics

The following examples of occam semantics will be best understood by referring to Figure 5.1 which contains a fictional example of occam text. Reserved words are always written in capital letters, e.g. VAL, SEQ, PAR, etc. Comments are preceded by the double dash, --, and last to the end of the line. There is no assumed precedence between the operators; explicit parentheses are required — e.g. addition is not associative.

Processes can be grouped into lists which are executed either sequentially or in parallel. The former uses the keyword SEQ, the latter uses PAR. Processes within a PAR are disjoint — i.e. they cannot share variables other than by communicating via channels. Variables can be declared at almost any point and 'attached' to a process by means of a colon. A variable is only in scope inside the process to which it is attached.

The conditional process is slightly different from the ‘if...then...else...’ construct which is found in most other common HHLs. The keyword IF introduces a list of processes with boolean guards. The first process whose guard is found to be TRUE is executed, and all others are ignored. At least one guard must be TRUE. Thus it becomes necessary to have an empty process, named SKIP, which does nothing. This is used in the case where it is desired to do nothing if all of the guards evaluate to FALSE.

Communication is provided by channels between concurrent processes. Com-
--
-- This is a comment
--

INT [tune.len] melody, folk.melody, -- Declarations of variables [tune.len][3] chords:
CHAN OF ANY jazz.to.folk, -- and channels
   folk.to.jazz:

PAR
   {{{
   Jazz music
   TIMER time:
   SEQ
   {{{
   Set the period
   IF
      today >= bebop.date
      ... Charlie Parker
      today >= swing.date
      ... Benny Goodman
      TRUE
      SKIP
   }}}
   -- the SEQ continues at
   -- this level of indentation.
   harmonise (melody, chords) -- Calling a procedure.
   newtune := reharmonise (chords) -- Calling a function.
   time ? AFTER delay*ticks.per.sec -- Wait for "delay" seconds
   jazz.to.folk ! melody[0] -- Send note to folk process
   folk.to.jazz ? folk.melody[0] -- Receive note from folk process
   }]
   -- the PAR continues at
   -- this level of indentation.

... Classical music -- More OPS Folds hiding
... Folk music -- other processes in the PAR

Figure 5.1: Example of occam layout
munication is synchronised; when both the input process, ?, and the output process, !, are ready, the values are copied from the output process to the input process. All programs written for this project (except for those to implement the balanced chain in Section 6.4) avoided the use of channels by employing the software routing harness, Tiny (Section 4.2.2). However some channels were explicitly used to allow concurrent processes to communicate on the same processor so as to enable parallel thinking to be maintained, e.g. the image file-reading processes (Section 5.4.1) and the 'round robin' slave queue (Section 5.5.1).

An unusual feature of occam is the provision of timers. In usage they look a little like channels. However they can only be read from and never written to. The returned result is the current value of the transputer on-chip clock. Timers have two main uses:

- Timing pieces of program;
- Causing delays of specific duration.

The second use is achieved by use of the reserved word, AFTER, which causes execution to halt until the specified number of processor clock ticks have elapsed. Timers are especially useful for real time applications.

5.3.3 Optimising occam programs

This section is devoted to the problems of optimising occam programs for shortest execution time. Various programming technical terms will necessarily be used since it is assumed that this section is not of interest to non-programmers. Experience with sequential languages has led to a standard body of program optimisation techniques such as the use of pointers in preference to array indices. Usually some assumptions can be made about how the compiler will translate the source code
into object code in order to gain further optimisation. For example it is usually assumed that procedure calls are implemented using a stack for local variables. Therefore large parameters such as arrays should be passed by reference, rather than by value, so as to minimise the time taken to set up the procedure call.

Occam is a static language—i.e. the compiler needs to know everything about the program before it can produce the executable code. This allows the compiler to perform various checks on the program to ensure its correctness which can be a useful feature. However, it also means that other useful features such as recursion, compound data types (e.g. C structures or Pascal records) and dynamic memory allocation cannot be allowed.

The occam compiler provides very different features, which means that the accepted rules for the optimisation of programs have to be relearned. For example, the transputer is designed to perform context switching extremely quickly. Thus it is not necessary to avoid running many parallel processes on the same processor. Thus the programmer is encouraged to think in parallel even when processes will actually execute on the same single processor.

An inmos technical note written by Atkin [154] was the only available document which describes how to write efficient programs in occam. Items considered include:

- Declaring frequently used variables so that they reside in on-chip memory;
- Using TIMES instead of * for integer multiplication by small constants;
- Buffering links and decoupling communication from computation;
- Giving high priority to processes using links;
- Using abbreviations.

Atkin's guidelines were used when writing the simulations in occam.
5.4 Uniprocessor implementation

Code was initially developed on a minimal network of two transputers; one for the simulations and the other for displaying the graphical output\(^2\). In this way, complications due to communications were removed. The uniprocessor timings are needed both to calculate the efficiencies of larger processor networks, and to determine a good choice of mapping. As previously discussed (Section 5.3), occam programs are written as processes connected by channels, regardless of which processor the processes will execute on. This uniformity of occam makes it very easy to write processes without needing to know whether they are running on the same or different processors.

5.4.1 Details of H.261 occam processes

The individual processes from which the simulations were built are described in this section. Throughout this description it is necessary to include some small fragments of occam processes by way of illustration. Unless the occam syntax is of particular interest, it is not shown and the process is only described in words.

**Image reading**

The images were held on disc as files of bytes representing pixel luminance and chrominance intensities. It was necessary to read in these files and convert the bytes into numbers of REAL32 format for the algorithm to operate upon. This was achieved by two processes running in parallel on the same processor. The first, `byteReader` (Figure 5.2), reads the files and sends the resulting stream of REAL32s

---

\(^2\)It is a feature of the ECS graphics domains that the graphics processor is never the processor connected to the host (and thence the filestore). Thus two is the smallest number of transputers we could employ since access to both the filestore and the display was required.
down a channel. The second, `imageReader` (Figure 5.3), receives these REAL32s and stores them in separate arrays for luminance and chrominance.

**Block differencing (DPCM)**

The MB from the 'previous' frame is subtracted from the MB from the 'current' frame on an element by element basis.

**Discrete cosine transform**

A 'fast' algorithm is used to implement the DCT process which operates upon blocks (8 x 8 pixels). Figure 5.4 shows the flow graph for the one-dimensional forward fast DCT (c.f. Equation (2.1)). It is a four pass algorithm. All loops were expanded to achieve fastest possible execution times. The two-dimensional DCT is performed by applying the one-dimensional DCT first to the rows of the block and then to the columns of the resulting block. A MB is transformed by calling the two-dimensional DCT procedure on each of the six blocks of the MB. The inverse DCT (IDCT) process uses the same flow graph implemented in reverse.

**Classification**

It was decided to implement the zig-zag scanning classification. A two-dimensional block, `block`, is zig-zag scanned into the one-dimensional array, `zblock`. Figure 5.5 shows the details.

**Quantisation**

The quantiser index for a DCT coefficient is calculated by simple division dependent upon the current quantiser step size, $g$. 
PROC byteReader (VAL BYTE file, INT nbytes,
CHAN OF ANY toFs, fromFs,
CHAN OF REAL32 toUser, CHAN OF ANY debug)

-- Opens image file, reads BYTES but supplies a stream
-- of REAL32s to client (User).

{{{
declarations
INT32 file.id:
BOOL opened.ok, closed.ok:
INT nrecords:
}}}
SEQ
... open file
IF
opened.ok
INT record.size, given:
SEQ
... Calculate number of records to read
... Check for part record at end
{{{
read records and send REALs to user
given := 0
[fileSys.maxDataBytes]BYTE buffer:
INT len:
SEQ i = 0 FOR nrecords
SEQ
readRecord (toFs, fromFs, file.id, len, buffer)
IF
(len > 0) AND (given < nbytes)
INT i:
SEQ
i := 0
WHILE (given < nbytes) AND
(i < fileSys.maxDataBytes)
SEQ -- Send REALs to User
toUser ! REAL32 ROUND (INT buffer[i])
i, given := i + 1, given + 1
}}}
TRUE
SKIP
... close the file

Figure 5.2: Byte file-reading processes.
PROC imageReader(REAL32 Y, U, V,
   CHAN OF REAL32 from.reader,
   CHAN OF ANY screen)

{{{
   VAL head.size IS (2048):
   [head.size]REAL32 trash:
   BOOL read.chrom.row:
}}}

SEQ
read.chrom.row := TRUE -- Read first chrom row,
   -- then every other one
SEQ row = 0 FOR Y.SIZE -- Rows
SEQ
   SEQ col = 0 FOR X.SIZE -- Luminance row
      from.reader ? Y[row][col]
   .IF
      read.chrom.row
         SEQ -- Chrominance is subsampled
            SEQ col = 0 FOR X.SIZE/2 -- V row
            SEQ
               from.reader ? V[row/2][col] -- Get chrom byte
            SEQ col = 0 FOR X.SIZE/2 -- U row
            SEQ
               from.reader ? U[row/2][col] -- Get chrom byte
         TRUE
            SKIP
      read.chrom.row := NOT read.chrom.row

Figure 5.3: Image reading processes.
CHAPTER 5. VIDEO CODER SIMULATIONS

Figure 5.4: Signal flow graph for the fast DCT algorithm used in the simulations. $s(.)$ represents the sine and $c(.)$ the cosine function. Notice that the results, $Y(k)$, appear in 'bit-reversed' order.

PROC zig.zag.scan (CHAN OF ANY debug, VAL [] []REAL32 block, []REAL32 zblock)

... x scan path table
... y scan path table

SEQ i = 0 FOR BLOCK.SIZE
SEQ j = 0 FOR BLOCK.SIZE
  zblock[((i*BLOCK.SIZE) + j) :=
    block[x[((i*BLOCK.SIZE)+j)][y[((i*BLOCK.SIZE)+j)]

Figure 5.5: Classification process.
Variable thresholding

The current quantiser step size, $g$, defines the initial threshold level. The aim is to increase the number of zero coefficients. If a coefficient falls short of the threshold then it is zeroed and the threshold is increased. Otherwise the threshold is reset to its initial value.

Variable length coding

The block being processed has been zig-zag scanned and quantised and therefore is now a one-dimensional array. Next it is necessary to go through it and pick out the events (level, run-length). Using the process $word\_len$ (Figure 5.6) the number of bits required to VLC each event, and thus the block, can be calculated. The end of block marker (EOB) requires a further two bits.

Variable length codes are assigned to the more likely events and fixed length codes of length 20 are assigned to the unlikely events. The actual codes generated by the H.261 algorithm are not of interest, only their lengths. A look up table indexed by level and run-length is used to avoid any delay incurred calculating the codes.

Motion estimation

This is the most complicated and computationally intense part of the whole H.261 algorithm. Before the search begins, DPCM without MC is calculated to determine whether ME and MC are required. Assuming a maximum displacement of 7 pixels the step algorithm iterates in three stages to find the resulting minimum absolute error. Note that this is a compromise of minimising the result with maximising execution speed; the optimum result could only be guaranteed by a brute force search over all possible positions. The order of the search is defined as shown
PROC word.len (VAL INT run, level, INT length)
    ... word.len.table
SEQ
    ... make level positive
    ... find word length for this EVENT
:

PROC VLC.block(VAL []INT iblock, INT bits, VAL INT x, y)

    INT run, level:
    INT p: -- pointer to zig-zag scanned array
SEQ
    run, p := 0, 0 -- initialise counters
    bits := 2 -- allow for EOB at the beginning!
    WHILE p < (BLOCK.SIZE*BLOCK.SIZE)
        SEQ
            IF iblock[p] = 0  -- found a zero
                run := run + 1 -- increment run length
            TRUE  -- otherwise
                INT length:
                SEQ
                    level := iblock[p] -- get the end of run level
                    word.len(run, level, length) -- get word length
                    bits := bits + length -- total for this block
                    run := 0 -- reinitialise run length
            p := p + 1 -- move pointer along

Figure 5.6: Variable length coding processes
in Figure 5.7, where the squares represent the displacement of the ME search centres relative to the offset of each iteration. The centre square represents no displacement. Figure 5.8 shows the details of the ME process. The tables updx and upd y are used to determine the size of the ‘jumps’ at each stage of the search. On the first pass, the jumps are four pixels, on the second, two pixels, and finally one pixel.

Motion vectors for a MB are produced by estimating the motion between the current frame and the previous frame. From these vectors a motion-compensated frame prediction for all three bands \((Y, C_R, C_B)\) is produced; the motion vectors are halved in magnitude when applied to the chrominance bands to allow for the spatial sub-sampling.

**Motion compensation**

Having made an estimation of the motion between frames the inter-frame block differencing is simply performed with the blocks displaced so as to compensate for the motion. This is easily achieved using the motion vectors calculated.

**Loop filter**

This filter works like most two-dimensional filters which convolve a \(3 \times 3\) kernel with the image [46]. However, in this case the edges are treated differently so as to save time:
SEQ

... make the search area for motion estimation

-- motion estimation
... constants

-- updx, updy the search position tables
VAL []INT updx IS [0,-4,0,4,-4,4,-4,0,4, 
0,-2,0,2,-2,2,-2,0,2,
0,-1,0,1,-1,1,-1,0,1]:

VAL []INT updy IS [0,4,4,0,0,-4,-4,-4,
0,2,2,0,0,-2,-2,-2,
0,1,1,0,0,-1,-1,-1]:

-- updx and updy contain the x and y coordinates of the
-- search patterns defined in the appendix A of the
-- RM8 "3-step algorithm"
... vars declared in here

SEQ

... init x and y frame co-ords
SEQ

... Difference with previous frame luminance MB

--

-- Three stage algorithm below here
--

... initialisations for the search

Figure 5.8: Motion estimation details (continued over...)
SEQ stage = 0 FOR 3

SEQ

... Transfer min vectors to prev iteration vector
SEQ loop.off = upd.off FOR 9 -- starts at 0,
-- then 9, then 18

SEQ

... set up temporary offset co-ordinates
... set up previous frame co-ordinates

-- calc total error
IF

... check search area not outside frame limits
... Sum frame difference
SEQ

... initialise totals
SEQ s = 0 FOR MACRO.BLOCK.SIZE -- rows
SEQ

... compute y offset into search area
SEQ p = 0 FOR MACRO.BLOCK.SIZE
SEQ

... compute x offset into search area
... take difference
... sum the squared error
... accumulate current.diff
... build me.diff.macro.lum

... If a better match, remember the vectors
TRUE

... say search went out with frame

upd.off := upd.off + 9 -- update offset for search
-- position tables for next pass
-- end of 3 stage loop

Figure 5.8: Motion estimation details continued.
PROC filter.block(REAL32 out, VAL real32 in)
INT x, y:
SEQ
... copy corners
... top and bottom rows
... left and right edges
-- centre of block
SEQ x = 1 FOR 6
SEQ y = 1 FOR 6
VAL a IS in[x-1][y-1]:          -- Abbreviations
VAL b IS in[x][y-1] * (2.0(REAL32)):  -- for array element
VAL c IS in[x+1][y-1]:          -- access efficiency.
VAL d IS in[x-1][y] * (2.0(REAL32)):
VAL e IS in[x][y] * (4.0(REAL32)):
VAL f IS in[x+1][y] * (2.0(REAL32)):
VAL g IS in[x-1][y+1]:
VAL h IS in[x][y+1] * (2.0(REAL32)):
VAL i IS in[x+1][y+1]:
SEQ
out[x][y] :=
(((a+b)+(c+d))+((f+g)+(h+i)))+e)/(16.0(REAL32))

Figure 5.9: Low pass filter process

- The corners are left unchanged, and are simply copied;
- The remaining edge pixels are filtered with a one-dimensional kernel.

The effect is to smooth any sharp edges and so hide any artifacts introduced by the motion compensation. These artifacts occur when DPCM is performed with MBs displaced so that the edges of one MB line up with the inner parts of the other MB. This Occam procedure makes effective use of abbreviations to produce an efficient program by reducing the range checking code. Figure 5.9 shows the details of the main part of the block filter process.
5.4.2 Input and output processes

As already explained, the images entering the H.261 algorithm have to be in a format with one luminance and two chrominance planes. The values within these planes are naturally stored as sequences of bytes. However, some parts of H.261 such as the DCT process, are more naturally implemented using floating point operations. On conventional microprocessors the obvious choice for ultimate speed would be to program the whole algorithm using integer arithmetic. However, since the T800’s FPU multiply operation is fast, the choice becomes less obvious. It was decided to convert the incoming images from bytes into floating point values before they were processed. Floating point format was retained throughout, until the results reached the display. In this way, excessive overheads due to numerous type conversions were avoided. The graphics program is forced to use byte values since these are required by the video-RAM.

I/O problems

It was deduced during the early stages of this project that, although close to real-time\(^3\) processing could be achieved, the displaying of results in real-time was not be possible. Section 6.6.1 describes the details of experiments conducted to achieve real-time display of images. At British Telecom Labs (BTL) hardware boxes are used to do the colour-space (YUV to RGB) conversions extremely quickly. When using the BTL Computing Surface, the extra hardware is simply inserted between the computer and the video-RAM. It was decided that further investigation into methods of achieving real-time speeds in software on the ECS could not be warranted. If necessary, the output frames could be written to disk and then later

---

\(^3\)‘Real-time’ is taken to mean 10 QCIF frame/s.
viewed in real time using the experimental configuration described in Section 6.6.1.

Input to the simulations was also problematical. Initially it was hoped that input from a camera could be used in conjunction with the Meiko frame grabber board. However, the camera associated with the frame grabber on the ECS was a low quality monochrome device and so was unsuitable. The alternative was to use standard test sequences supplied by the CCITT. These are stored as computer files on disc. Since accessing the filestore involves using the shared supervisor bus on the ECS and disc access times are slow, there was no hope of reading images from disc in real time. The solution adopted was to read a sequence of frames into the RAM of the master processor before simulations began. There were 4 Mbytes of RAM available. These were shared between program and data. It was determined that there was room for 10 to 20 QCIF frames depending upon the size of the simulation master process being used.

Client/server model for display requests

Client/server models are often used to provide a clean interface between a resource and its users. The basic idea is that two libraries are provided: the client library which deals with requests which use the resource, and the server library which provides access to the resource so as to satisfy the request. A client and a server process run in parallel; the user talks to the client and the client talks to the server. The graphics primitives were designed to work in this manner, based upon Meiko's gfx software [155]. Gfx is a high level graphics library providing all the usual functions such as line drawing, polygon filling, etc. Most of the functions were not required for this project and therefore a cut-down version tailored for

\footnote{This has the added advantage that results can be compared with those produced at other sites.}
### Table 5.1: Breakdown of H.261 function execution times.

<table>
<thead>
<tr>
<th>Function</th>
<th>% Time</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DPCM</td>
<td>1.3</td>
<td>1.9</td>
</tr>
<tr>
<td>DCT</td>
<td>20.0</td>
<td>30.0</td>
</tr>
<tr>
<td>Scanning</td>
<td>5.5</td>
<td>8.3</td>
</tr>
<tr>
<td>Quantisation</td>
<td>1.1</td>
<td>1.7</td>
</tr>
<tr>
<td>Inv. quantisation</td>
<td>1.0</td>
<td>1.4</td>
</tr>
<tr>
<td>Inv. scanning</td>
<td>2.2</td>
<td>4.9</td>
</tr>
<tr>
<td>Inv. DCT</td>
<td>19.0</td>
<td>29.0</td>
</tr>
<tr>
<td>Inv. DPCM</td>
<td>1.2</td>
<td>1.8</td>
</tr>
<tr>
<td>VLC</td>
<td>1.0</td>
<td>1.1</td>
</tr>
<tr>
<td>Motion estimation</td>
<td>47.0</td>
<td>70.0</td>
</tr>
<tr>
<td>Total</td>
<td>100.0</td>
<td>150.0</td>
</tr>
</tbody>
</table>

Table 5.1: Breakdown of H.261 function execution times.

speed was derived.

It was relatively easy to write higher level display processes, once the graphics primitives such as draw block, clear screen, and draw row were available. These are described in Section 6.6.1.

### 5.4.3 Timing profile for the occam implementation

In Section 5.2 it was found necessary to know the timing profile of the processes simulating the algorithm in order to determine a good mapping of those processes. Table 5.1 shows the proportion of time taken to calculate the various functional blocks of H.261 using the occam processes. These values are averages for standard test image sequences. Motion estimation consumes far more computational power (47%) than any of the other functional blocks. A pie chart of these results is given in Figure 5.10 for easier visualisation.
5.5 **Multiprocessor implementations**

In the light of experience of previous attempts at implementing H.261 in real-time [156], it was known that the fastest possible techniques would have to be exploited, perhaps at the expense of good programming style. In this section the timing profile derived above is analysed with a view to choosing a well balanced parallel processor mapping. It was seen in Chapter 4 that theoretical mapping models in their current state of development are of little practical use and they were therefore rejected in favour of better understood stereotyped mapping paradigms.

5.5.1 **Choice of mapping strategy**

From the timing profile presented above (Table 5.1 and Figure 5.10) it is obvious that the processes are extremely unbalanced in their processing requirements. Since nearly 50% of the time is spent on the motion estimation, it seemed that an algorithmic decomposition could only be split into *two* portions, and hence only utilise *two* PEs, if good load balancing were to be maintained (Figure 5.11).
Instead it was decided to use a hybrid of the task farm and algorithmic decomposition. The usual master/worker relationship found in any task farm was employed. However, there was also an additional graphics process which resided on the unique graphics PE and handled all display requests. Thus, an implementation with three distinct occam processes was opted for, as shown in Figure 5.12:

- A master process which distributes the MBs and receives the results. This process is also responsible for the updating of various data structures, such as the transmission buffer and the frame store, as the simulation results return. It must run on the transputer which has one of its links connected to the host so that it can access the file store. Also it is the only process with access to the keyboard and screen and therefore has to be the one which handles user instructions such as whether the display is on or off.

- A worker process which receives MBs from the master and processes them. The processed data is returned to the master and any required results can be sent to the graphics to be displayed. This worker process is duplicated $n - 1$ times, where $n$ is the total number of PEs used, not including the special graphics PE.
A graphics process supplies a client/server interface to the graphics board facilities. This was described in Section 5.4.2. It writes directly to video RAM and provides a minimum of functionality so as to maximise performance.

In the worker process, incoming data flows uninterrupted through the processing blocks: frame differencing; discrete cosine transform; quantiser; inverse quantiser; inverse DCT; variable length coder. Thus the worker processes can perform these functions on MBs in sequence, before returning the result to the master process. This avoids any excessive communications that might have been encountered with a pipelined approach (see Section 5.9.1).

As processed MBs arrive back from the workers, the master maintains the frame store and the transmission buffer accordingly. The quantiser step size is also updated by the master process as the simulation progresses.

The master process (Figure 5.13) initially reads a sequence of frames via the host into its local RAM. A 'round robin' [129] queue of worker IDs is maintained to ensure that the first available worker is always known. Figure 5.14 shows the
... Read in image sequence

PAR

... Send and receive MBs with slaves and time it

{ {{ initialise queue

SEQ

{ {{ Queue them all once

SEQ i = first.slave FOR num.slaves

SEQ

to.q ! i -- slave ID to queue

})}

... and then again for a job in hand

}}

Queue (to.q, q.ack, q.req) -- The slave queue itself

Figure 5.13: The master process structure

complete Queue process. Notice the use of channels to enable the queue process to run in parallel with the master process. Initially all workers are enqueued twice by sending the slave IDs down the to.q channel. This ensures that all workers receive a job to work on, as well as a second job which waits in a buffer on the worker transputer (see Section 5.5.2). When a slave ID is required, a request is sent down the channel, q.req. When the Queue process receives the request, it dequeues the first slave ID, and returns it via the channel, q.ack.

In this way it was extremely easy to build up the simulation step by step. For example the DCT routines were developed in the following way: The master process simply split a frame into MBs and sent these to the workers. The worker process would perform the DCT on the MB and send its result to the graphics for a visual check and to the master for numerical verification.
PROC Queue (CHAN OF INT to.q, q.ack, CHAN OF SIGNAL q.req)
INT head, tail, free:
[Qsize]INT Q:
SEQ
  free, head, tail := Qsize, 0, 0 -- initialise pointers
  WHILE TRUE -- run forever
    ALT
      free > 0 & to.q ? Q[tail] -- enqueue
        free, tail := free -1, (tail + 1) REM Qsize
      free < Qsize & q.req ? CASE signal -- dequeue
        SEQ
          q.ack ! Q[head] -- send first free slave ID
          -- on acknowledge channel.
          -- Then update queue pointers
          free, head := free + 1, (head + 1) REM Qsize

Figure 5.14: The round robin queue process

5.5.2 Selecting the grain size

For our particular problem the frame of data is already broken down into a hierarchical structure. From this, the smallest element upon which all the tasks of the H.261 algorithm can be performed independently is the MB. The MB was chosen as the grain size. Any smaller grain would not be practical to operate upon independently (e.g. MC operates on MBs, not blocks); any larger would not utilise the number of PEs available and allow good load balancing. There are 99 MBs in a single image frame; the largest ECS domains presently contain around 100 PEs.

Communications were overlapped with computation by ensuring that each processor worker was initially sent two MBs. On receiving them, the first was processed whilst the second was buffered. On completing the processing of the first MB, the worker sent the results to the master and immediately started work on the waiting MB. On receiving a result from a worker, the master sends that worker another
CHAPTER 5. VIDEO CODER SIMULATIONS

MB to process.

Experiments with more than one MB buffered at the worker showed that communications were already maximally overlapped with computation, for the chosen grain size (see Section 6.7.1).

5.6 Processor network topologies used

It was shown in Chapter 4 that the definition of a good PE network topology is extremely application dependent. Having chosen the implementation strategy, the problem of finding the 'best' topology could be addressed. The terms 'network topology' and 'graph' are be used interchangeably throughout the following discussion.

Having all of the main H.261 functions contained in each worker process meant that it was easy to experiment with various topologies with little reprogramming. For the chosen mapping strategy the communications pattern is easily derived. The master process needs to communicate with all the worker processes; all the worker processes must communicate with both the master and the graphics processes. Thus the 'best' topology would seem to be one in which the master and graphics processors are as close as possible, on average, to all the workers. Since the workers do not need to communicate with each other, their distance is not considered important. For small numbers of PEs (i.e. less than eight), one topology which uses all available links was found to perform as well as the next (see Section 5.6.2). The transputer has four links and, therefore, up to five PEs can be fully interconnected. Communications overheads are not observed until the number of PEs becomes significantly larger. In this discussion larger networks are assumed.
5.6.1 Definitions

Since the message length is fixed by the chosen granularity, communications can only be minimised by minimising the number of links involved in passing the messages between PEs. To this end such measures of topology as diameter and mean inter-PE distance are of interest [157].

5.6.2 Topology choice

Given that the chosen parallelisation strategy was the processor farm, the next problem was that of deciding how to interconnect the farm of workers. Processor farms have often been implemented as chains (or load balancing pipes), triple chains (or parallel pipes), and tertiary trees [158]. It can be shown that none of these is particularly suited to the current application. The following discussion explains how compact graph [157] topologies came to be favoured. Throughout the discussion it is assumed that, one master PE schedules tasks, \( n \) worker PEs process them and one graphics PE displays their results.

Hypercube topologies

The hypercube (or binary \( k \)-cube) has been a very popular choice of topology for parallel machines of \textit{fixed} topology. The reason for this popularity is the high interconnectivity provided for a small number of connections at each node. For larger transputer-based machines, however, it is limiting. With four links at each node, at most sixteen nodes can be interconnected in this way. Thus, for this project, hypercubes are not suitable. Additionally, analysis shows that smaller diameters and average inter-node distances can be obtained from other topologies (see later in this section).
CHAPTER 5. VIDEO CODER SIMULATIONS

Figure 5.15: Chain topology. 'M' represents the master, 1 to n represent the workers.

**Chain and tree topologies**

Chains were used to implement a multifunction pipeline in the work preceding this project, which was carried out at BTRL by Sexton (see Section 5.9.1). Figure 5.15 shows a typical chain topology. The squares represent transputers connected by their bidirectional links. The master distributes tasks to the numbered workers on demand. A point in favour of chain topologies is that the routing algorithm is extremely easy to write. However, now that topology independent routing harnesses are available this is now less significant.

The chain is not a good topology since two out of the four links are unused. This may be expressed as a high narrowness [157]. The narrowness of a graph is the worst ratio of processing power to communications power. In this case there are n workers whose messages must all travel to the master via the single link connecting the master to the chain. For the chain topology, the narrowness is calculated as:

$$N_{\text{chain}} = \frac{n}{1} = n = O(n)$$

where $N$ is the narrowness, and $n$ the number of worker processors.
Another problem is that the mean master-worker distance for the chain, \( \mu_{m-w_{\text{chain}}} \), is large. It can be shown to be of order \( n \):

\[
\mu_{m-w_{\text{chain}}} = \frac{n(n+1)}{2} \cdot \frac{1}{n} = \frac{1}{2} (n + 1) = O(n)
\]  

(5.2)

Minimising \( \mu \) is desirable because:

- fewer PEs are disturbed by through-routing messages;

- more bandwidth is left free;

- message source-to-destination time is reduced.

A better topology for link utilisation is the tertiary tree (Figure 5.16). The narrowness of the master's links to the subtrees is captured in (5.3).

\[
N_{\text{tree}} = \frac{n}{3} = O(n)
\]  

(5.3)

By comparing (5.1) with (5.3) we see that the tree topology is not as narrow as the chain. This is because more of the links are employed. However, at the leaves of the tree, still only one of the four available links is used; there is still wasted communications bandwidth. The PEs are closer to the master in a tree topology than in the chain, as shown by (5.4) [109].

\[
\mu_{m-w_{\text{tree}}} \leq 2(\log_2(n + 1) - 1) = O(\log n)
\]  

(5.4)

It is not obvious where to put the graphics PE with which all the workers must communicate. The problem with the tree for this application is that the general mean inter-PE distance is large due to the leaves on separate branches being relatively distant. Thus the graphics PE cannot be close to all the worker PEs.
Link loading on both the tree and the chain is unbalanced, severely so for the chain. Assume a message is sent from the master to every worker in each unit time-step. Then $n$ messages pass through the first link, $n - 1$ through the second, and so on down to the last link which only carries one message. As $n$ grows, the links near the master become bottlenecks and the bandwidth of the ones farthest away is underutilised, i.e. again the chain is seen to scale badly. Generally, link loading is higher than it needs to be, since for both topologies not all links are utilised and there are no alternate routes between PEs that a router might take advantage of during times of bandwidth saturation.

**Compact graphs**

In contrast to the chain and tree topologies, a family of PE graphs known as *compact graphs* [157], performs very close to theoretical bounds for all the measures discussed so far. Additionally the graphics PE can be included anywhere in the
network without introducing undue degradation [159]. The presence of alternative paths between PEs allows an intelligent router to maintain good performance despite heavy communications loading.

Recent research [157] has shown that regular topologies are rarely optimal. Furthermore, certain irregular topologies are shown to perform very closely to theoretical minima with respect to such measures as diameter, mean inter-processor distance and worst through-routing time. Due to this 'closeness' of the nodes in these graphs they are termed compact graphs. Three different kinds of compact graphs were considered in this work (see Figure 5.17). Each was developed from a number of PEs connected in a ring. This was necessary so that a 'boot path' to every PE was guaranteed, through which programs could be loaded. This is known as a Hamiltonian boot path. Graph generation and optimisation are considered later in Section 5.7. A brief description of how each type is formed is given below:
Chordal rings These regular graphs are formed by connecting chords across a ring of processors [160]. Trial and error can be used on the chord length to generate optimum graphs of this kind. Chordal rings have been popular in the past because their regular structure makes routing algorithms easier to implement [161].

'Greedy' graphs These graphs are produced by iteratively placing links so as to locally optimise the partial solution. Thus a 'greedy' graph might be generated by progressively connecting links which will most reduce the diameter of the graph. The resulting graph may not have minimal diameter for the number of processors but it will have a low diameter. 'Greedy' algorithms [139] are often used in optimisation problems.

Random graphs A random graph is easily configured by first wiring the PEs in a ring and then randomly connecting up remaining pairs of links. The only constraint on the graphs is that they contain no self-links. Such a linking scheme may cause bottle-necks to appear. Several random graphs were generated and the one with the lowest mean inter-PE distance was selected.

5.7 Programs for graph manipulation

It was necessary to write programs to manipulate the processor graphs so that good PE topologies could be sought. All the graph programs were written in C. They all have a certain amount in common. An $N \times N$ distance matrix is used to store the distances between the $N$ nodes. This distance matrix is symmetrical about its main diagonal but it was stored in full so as to save time in subsequent calculations. The distance matrix is initialised so as to be empty except for a Hamiltonian ring
Figure 5.18: The inter-connection distance matrix with only the Hamiltonian ring entries, where asterisks represent uninitialised values.

connecting all $N$ nodes. This is because all the topologies needed a boot path to all PEs so that programs could be loaded from the host (Figure 5.18).

### 5.7.1 Graph generation programs

The application graphs used for these simulations were constrained by several consideration:

- All nodes are of degree four since transputers have four links.

- The master processor is linked to the host machine and therefore has only three remaining links for connection in networks being created.

- The graphics processor is unique and should therefore be carefully positioned in the network.

This explains why the mean inter-PE distance measure is important rather than just the mean master-to-worker distance. Three programs called mkcrd, mkrnd and mkgdy were written to generate chordal rings, random and ‘greedy’ graphs respectively. Figure 5.17 shows examples of these three types of compact graphs for eight-worker networks.
Chordal rings

Various different chordal rings can be generated for a given number of nodes, \( N \), each with a different chord length. From the resulting graphs, the 'best' chordal ring graph was selected and retained. Here the word 'best' is used in the same sense as explained in Section 5.6. Chord lengths in the range two to \( N/2 \) were tested. A chord length of one would simply produce a doubly linked ring, and by symmetry, a chord greater than \( N/2 \) is equivalent to a chord less than \( N/2 \) by the same amount.

Thus after generating a chordal ring of a given chord length, it was necessary to calculate the inter-node distances and the mean inter-node distances. If the mean inter-node distance was the smallest so far, then it was recorded along with the chord length which was used to generate it. In this way, the 'best' chord length was determined.

Random graphs

Two tables were used in the random graph generation program, one for nodes which had a link free to connect to and the other for nodes which had a link from which to connect. Thus all nodes except the master appeared in both tables since they have two free links after being connected in a ring. The master, being connected to the host, appeared in only one of the tables. An element from each of these tables was selected using a random number generator and the nodes contained therein were connected. The table sizes were reduced by removing the elements just used and moving all elements above down by one index. The only restriction placed on this choice of nodes was that a node was not allowed to link to itself. If a self link were selected, then it was discarded and another random pair was selected instead. In this way, random graphs were generated very quickly. The ease and speed of
creation of random graphs is a chief point in their favour.

‘Greedy’ graphs

The program to generate greedy graphs was written recursively. Initially, the nodes are connected in a ring. All inter-node distances are calculated and, as a side effect, the diameter, $D$, is known. Next, all pairs of nodes at distance $D$ are linked if it is possible. While there are links left, the program recurses connecting all pairs of nodes at distance $D - 1$. In this way, the diameter only has to be calculated once at the beginning. It is interesting to note that the ‘greedy’ graphs algorithm generation actually produced chordal rings, but not of the same chord length as those generated by the chordal ring generation algorithm (Figure 5.17).

5.7.2 Graph analysis program

A program was written to calculate the distances between all nodes of a processor graph. Once a graph is generated, the node inter-connection distance matrix contains only distances of one, representing a direct link, or zero, representing the distance a node is from itself. From this information, a distance matrix of distances between all processors is easily deduced. In graph theory this is termed the ‘all-pairs shortest path’ problem, for the solution of which there are several algorithms (e.g. Floyd-Warshall described by Cormen et al. [139] and reproduced in psuedo-code in Figure 5.19). The solution adopted in this work was similar but simpler, since all edges in the graphs used were unweighted (equal communications load since all worker PEs perform identical tasks) and undirected (bi-directional links). Thus the distance matrix of shortest paths is symmetric and therefore only half of its values needs to be computed. Additionally, the shortest path from a node to any other must begin through the node’s immediate neighbours. Thus the
program Floyd-Warshall
begin
    initialise d; \{ The distance matrix for the \( N \) nodes\}
    for \( x := 1 \) to \( N \) do
        for \( u := 1 \) to \( N \) do
            for \( v := 1 \) to \( N \) do
                \( d[u,v] := \min(d[u,v], d[u,x] + d[x,v]) \)
        end.
end.

Figure 5.19: Pseudo-code for the Floyd-Warshall algorithm.

An iterative loop is entered which calculates all distances in the distance matrix of length two. Once these are all located and entered into the distance matrix, distances of length three can be found. In this way, increasing distances are calculated until all entries in the distance matrix are completed.

The matrix of inter-node distances can be practically displayed for small \( N \) only (e.g. Figure 5.20), assuming that the output device has limited line length. The mean inter-node distance is calculated using half the elements of the distance matrix due to its symmetry. Also the distance matrix can be used to easily deduce how many of the available links have been used, simply by counting the ones in each row of the distance matrix.

5.7.3 ECS wiring file generation

In Chapter 3, it was explained that the topology of ECS domains is defined by the user by means of a wiring file. Once the node inter-connection distance matrix is completed by one of the graph generation programs, a wiring file for an ECS domain can be easily generated (Figure 5.21). The first number is the number of lines which follow in the file. The next line represents the host link (processor 0
Average distance between processors is 1.281250
Max dist between processors is 2

1 0 1 1 2 2 2 1
2 1 0 1 2 2 1 2
3 1 1 0 1 2 2 2 2
4 2 2 1 0 1 2 1 1
5 2 2 2 1 0 1 1 1
6 2 1 2 2 1 0 1 2
7 2 2 2 1 1 1 0 1
8 1 2 2 1 1 2 1 0

1 2 3 4 5 6 7 8

Figure 5.20: The output of the graph analysis program showing the complete distance matrix for a random graph.

link 1 → processor 1 link 0) since processor 0 is taken to be the host. The next eight lines are the Hamiltonian ring and the rest of the file represents the randomly connected remaining links. The graphics processor is represented by -1.

5.8 Choice of languages

Initially the simulations were written entirely in occam [151] under OPS. The sequential processes themselves, however, were far more easily written in a more semantically rich language such as C which allows the use of pointers, recursion, explicit memory allocation and has a heritage of useful libraries. Development in C is further aided by the existence of familiar environments with powerful debugging tools (see Section 4.2.3). It was found that in translating processes from occam into C, up to twice the sequential execution time speedup could generally be achieved. A similar reduction in program development time was observed. For these reasons all program development is now done in the C programming language.
Figure 5.21: The ECS domain wiring file for the above random graph.

The third phase in the development of the simulations will be to move back to using Tiny. Instead of OPS and occam, the 3L development system is now used to tie the C sequential processes together. In this way the best communications speed from Tiny and the most efficiently written code from C is achieved (see Section 6.7.2).

5.9 Other related implementations

There is little work directly related to this project. However, the work by Sexton et al. (Section 5.9.1), which preceded the present project, used a similar Meiko Computing Surface and the aims were broadly similar. This section discusses Sexton’s work as well as other more broadly related work.
5.9.1 Functional decomposition using chains

This project started at British Telecom Research Laboratories (BTRL) as a serial simulation of the draft software simulation specification for H.261, known as the reference model (RM) [36]. The programs were written on a VAX 750. Parallel image processing problems have been traditionally tackled as a multi-staged pipeline [136]. This was the approach used by Sexton [156] at BTRL in work immediately preceding this project. Each stage was implemented as a load-balancing chain. A speedup of 20 times over the previous serial implementation on a VAX 750 was achieved.

However, fundamental problems prevented the simulation from reaching real-time speeds. Figure 5.22 shows the data flow through the pipelined processor graph. Load balancing is a problem if multi-stage pipelines are used; considerable effort must be made to ensure that each stage of the pipeline is equally computationally intense [162]. Multi-stage pipelines have a communications problem. If each load balancing chain is the correct length then, on average, each job travels half way down the chain and back up again. This usually makes for excessive communications overheads. The load-balancing chain topology is analysed in Section 6.4.

The original sequential implementation at BTRL was coded in C on a VAX 750 and ran at approximately 15 minutes per QCIF frame. Sexton’s parallelisation of the C code with a hand written occam harness on a 14-transputer pipelined topology produced speeds of less than 10 seconds per QCIF frame.
5.9.2 Fine grained customised functional mapping

A fine grained functional decomposition has been implemented using transputers by Ichikawa and Shamura [148]. This uses 100 transputers in a hybrid topology tailored specifically to the requirements of this algorithm. Different parts of the topology are tailored to meet the functional requirements of various parts of the algorithm. The result is a massive amount of non-reusable engineering which would far better have been achieved in hardware, since changes in the algorithm would almost certainly require changes in the topology.

5.9.3 Local area network variation on H.261

Rudberg et al. [163] have implemented a variation of recommendation H.261 for a local area network (LAN). Their variation on H.261 does not include variable length coding of the DCT coefficients, and only intra-frame coding is implemented. In this way the most computationally intense part of the algorithm, viz. motion compensation, is ignored. Their mapping strategy is essentially the same as the
farming model adopted in this thesis (published prior to Rudberg as [164]), whereby all the H.261 functions are implemented on each transputer. By contrast however, they do not use a message passing system and all their programs are written in occam. The published results are for a tree topology using 16 transputers. They report execution speeds of less than two QCIF frame/s for their reduced algorithm.

Their work is not easily compared with the work contained in this thesis. Firstly the algorithm being implemented is quite different and secondly, their aim is simply to provide a LAN videophone service. Whereas the aim of the work described in this thesis is to provide a flexible implementation of the complete CCITT recommendation H.261 algorithm, which allows the effects of the stages of the algorithm to be clearly visible.

5.9.4 JPEG algorithm on transputers

In Section 2.2.3 the JPEG algorithm was discussed. The JPEG standard image compression algorithm is very similar to recommendation H.261. Recently an implementation of this algorithm using transputers was reported [165]. The programs were written in C but details of the mapping are unavailable.

5.10 Summary

A system has been described that can be used for the evaluation and development of motion video codec algorithms. As an example, a simulation of the current international standard algorithm for motion video coding, CCITT recommendation H.261, was described. The strategy described using a topology independent routing harness and compact processor graphs is flexible and conceptually easy to understand. In addition, since the processor graphs are easily generated, this
solution can be applied with virtually any of number of transputers.

Difficulties in simulating a complex image compression algorithm in real-time have been discussed. The evolution of the implementation of the simulations from the use of occam with the Tiny packet routing harness, through C with Meiko's CS Tools, and ultimately 3L C with Tiny was described. A comparison of the occam execution times with those achieved using C indicate that the occam coding is far from optimised. Using C compilers it is much clearer how programs should be organised to achieve high efficiencies.

Finally some simulations carried out by others were reviewed and compared with this project.
Chapter 6

Simulation results

6.1 Introduction

Chapter 5 describes the techniques used in carrying out the videophone codec simulations. In this chapter, the results of the simulations carried out on the ECS are presented. Section 6.2 introduces the notion of determining the execution times of both parallel and sequential pieces of program. Amdahl's law is defined and its impact upon the potential parallel speedup on MIMD machines is outlined. Section 6.3 presents and compares the timing results achieved for the individual H.261 functions using the languages occam and C.

In order to verify the theoretical prediction made in Section 5.6.2, an implementation was carried out using the load balancing chain configuration of transputers. These experiments are described in Section 6.4. Section 5.7 gives details of how the chosen compact graph topologies were generated and analysed so as to minimise inter-PE communication.

One of the principal aims of this work was to investigate how algorithms could be simulated so that intermediate results were easily visualised. Video output from
the H.261 simulations are shown in Section 6.6.

6.2 Timing the simulations

Since the aim of this work is to use parallelism to simulate the H.261 algorithm in real time, the actual timing of the various parts of the simulations was very important. For most events, average timings were calculated.

6.2.1 Amdahl's law

In the field of Computer Science there is a well known equation governing the speedup of parallel programs, called Amdahl's Law [166]. It covers the situation which arises whenever the total time of some activity is the sum of a fast process and a slow process. For MIMD machines:

- A program is said to have parallelisation degree $k$ if it can be divided into $k$ parts which can all run in parallel.

- Let $T_k$ be the execution time of a program with parallelisation degree $k$ on $k$ PEs.

- Let $f$ be the portion of the MIMD program ($0 \leq f \leq 1$) that cannot be executed with parallelisation degree $k$, but only sequentially (degree 1).

In all the following equations in this section it is assumed that the degree of parallelisation of the program, $k$, is large enough for the number of processors, $N$, to be exploited. The time, $T_N$ for the MIMD program to complete on a machine with $N$ processors is given by:

$$T_N \geq fT_1 + (1 - f)\frac{T_1}{N}$$  \hspace{1cm} (6.1)
where $T_1$ is the time for the program to complete on a single processor. Parallel speedup is given by:

$$\text{Speedup}_N \overset{\text{def}}{=} \frac{T_1}{T_N} \leq \frac{N}{1 + f(N - 1)}$$  \hspace{1cm} (6.2)

Amdahl's law states that the parallel speedup cannot exceed the speedup which is dictated by the sequential part of the program. For example, if $N = 100$, the program has parallelisation degree 100 to match $N$, and 1% of the program cannot be parallelised ($f = 0.01$):

$$\text{Speedup}_{100} \leq \frac{100}{1 + \frac{100 - 1}{100}} = 50$$  \hspace{1cm} (6.3)

Despite using 100 PEs on a problem with the same degree of parallelism, the 1% that cannot be parallelised has drastically reduced the actual maximum possible speedup from the expected 100 down to 50: a 50% inefficiency. Figure 6.1 shows speedup plotted against the number of PEs for this situation. The curve asymptotically approaches 100 as explained below.

A further generalisation can be made for large numbers of PEs. In the limit, as the number of processors approaches infinity [109]:

$$\lim_{N \to \infty} S_N(f) = \frac{N}{1 + f(N + 1)} = \frac{1}{f}$$  \hspace{1cm} (6.4)

This demonstrates the constricting effect that the sequential part of the program can have. If 1% of the program cannot be parallelised, then there is a maximum speedup of 100, since:

$$S_\infty(0.001) = \frac{1}{0.001} = 100$$  \hspace{1cm} (6.5)
For a program which is 10% sequential, the maximum speedup is reduced to 10. It should be noted that this is an upper bound: all of these figures are calculated without taking into account the effects due to communication; the actual speedup observed will be smaller.

6.2.2 Timing in occam

As explained in Section 5.3, occam provides a special means for accessing a real-time clock called a TIMER [152]. A timer is like a CHAN OF INT which can only be read. It returns a number which is the current value of the transputer on-chip clock. The clock is simply an integer which increments continually at the approximately fixed rate of ticksPerSec\(^1\) which is defined in the system libraries. Thus a program fragment can be timed by reading the transputer timer at the beginning and again at the end. The two values are then subtracted using modulus arithmetic and

\(^1\)ticksPerSec is defined as 15625
PROC show secs (CHAN OF ANY debug, VAL BYTE string, INT now, then)

... vars
PROC ticks to secs (VAL INT ticks, REAL32 secs)
SEQ
secs := (REAL32 ROUND ticks)/(REAL32 ROUND ticksPerSecond)

SEQ
ticks to secs(now MINUS then, time)
... show time and message

Figure 6.2: The occam timing process.

the result is divided by ticksPerSec. In order to time parallel processes, it was necessary to first enclose them in a SEQ process. Figure 6.2 shows how this was packaged into a useful occam process which was used to calculate the execution times for all the occam programs used in the results in this chapter. The timer speeds vary slightly between transputers but this was not important since many transputers were being used and so the results were automatically averaged.

6.2.3 Timing in C

The timing in C is also performed on the master processor by counting clock ‘ticks’. A C library function, unsigned ticks(void), similar to an occam TIMER returns an unsigned integer. This integer is the value of a counter which is incremented every 64 μs. It is not a standard C or UNIX function, but one specifically written by Meiko Ltd. for the transputer.
6.3 Uniprocessor timings for H.261

In Chapter 5 (Table 5.1) a timing profile for the H.261 function implemented in occam was presented. The values in the table were obtained using TIMERS as explained above. This Table is extended in Table 6.1 to include the timing from the simulations written in the C programming language.

The total time to process a single MB using the occam processes is approximately 150 ms. Thus, even without any communication delays, the fastest processing rate possible would be less than seven frame/s if all 99 MBs in the frame were processed in parallel. Therefore, real-time simulation of H.261 is not possible using this combination of machine and programming language. For this reason, as well as the increased ease of development, further simulations were carried out using the C programming language as discussed in Section 5.8.

<table>
<thead>
<tr>
<th>Function</th>
<th>Absolute time (ms)</th>
<th>occam</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>DPCM</td>
<td>1.9</td>
<td>3.2</td>
<td></td>
</tr>
<tr>
<td>DCT</td>
<td>30.0</td>
<td>12.0</td>
<td></td>
</tr>
<tr>
<td>Scanning</td>
<td>8.3</td>
<td>0.4</td>
<td></td>
</tr>
<tr>
<td>Quantisation</td>
<td>1.7</td>
<td>0.4</td>
<td></td>
</tr>
<tr>
<td>Inv. quantisation</td>
<td>1.4</td>
<td>0.4</td>
<td></td>
</tr>
<tr>
<td>Inv. scanning</td>
<td>2.2</td>
<td>0.5</td>
<td></td>
</tr>
<tr>
<td>Inv. DCT</td>
<td>29.0</td>
<td>11.4</td>
<td></td>
</tr>
<tr>
<td>Inv. DPCM</td>
<td>1.8</td>
<td>3.1</td>
<td></td>
</tr>
<tr>
<td>VLC</td>
<td>1.1</td>
<td>0.5</td>
<td></td>
</tr>
<tr>
<td>Low pass filter</td>
<td>7.3</td>
<td>9.6</td>
<td></td>
</tr>
<tr>
<td>Motion estimation</td>
<td>70.0</td>
<td>39.0</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>157.3</td>
<td>81.5</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.1: Breakdown of H.261 function execution times for a single macro block
CHAPTER 6. SIMULATION RESULTS

6.3.1 Maximum theoretical speedup

Using Amdahl's Law (Section 6.2.1), it is possible to predict the maximum possible speedup for a given parallel program. In these simulations, the master process has the task of creating, distributing (or farming) and collating MBs. It is this part of the parallel program which is intrinsically sequential. Even if an infinite number of PEs were used for the other parts, this part (of parallelisation degree one) dictates the maximum speedup. From these simulations it was discovered that the sequential part of the programs amounted to a proportion of about 2–3%. Thus, from (6.4), an upper bound on speedup is:

\[
\text{Maximum speedup} = \frac{1}{0.03} = 33 \tag{6.6}
\]

This would dictate a maximum frame rate of around 4 Hz. Thus, real-time processing (10 Hz) is impossible using this system even without communication overheads.

6.4 Load balancing chain implementation

When Sexton (Section 5.9.1) reported his implementation of recommendation H.261, he did not include actual specific execution times. Obviously, it was not possible to exactly reconstruct Sexton's simulations, since his programs and enough specific details were not available. In order to make a valid comparison it was decided to try to reconstruct a similar kind of setup from which results could be extracted. In one sense this was in fact a fairer comparison than comparing with Sexton's original results, since the same occam processes were used in both cases. Thus it is the differences in communications strategy and processor topology which are being highlighted.
Figure 6.3: Chain topology. The master process is represented by 'M', the chain of workers is PEs 1 to n.

The concept of a chain (or load balancing pipe [167]) is a simple one. The PEs are connected in a line (or chain) using only two of their four bi-directional links as shown in Figure 6.3. Since a routing harness was not being used, it was necessary to explicitly buffer the links. Special buffer processes were used to achieve this which ensured that communication could take place concurrently with computation. The master was almost identical to the master described in Section 5.5. Each node of the chain ran one of two distinct processes, termed main-node and end-node, running on the main (1 to n - 1) and end (n) PEs of the chain respectively. These are broadly similar in function and contain the worker process discussed in Section 5.5. The main-node, being the most complex, is described first.

Figure 6.4 shows the processes running on the main-nodes of the chain. In the middle is the worker process which performs the functions from H.261 on the incoming MBs. The MBs are received by a splitting buffer which determines whether the worker process is busy. If not then it sends the MB to the worker process via a handshaking buffer (Figure 6.5). Otherwise, the MB is sent to the next PE down the pipe via another handshaking
CHAPTER 6. SIMULATION RESULTS

Figure 6.4: Process running on a main-node of the chain.

PROC HandshakeBuf (CHAN OF ANY in, out, handshake) [buffer.size]INT msg:

WHILE TRUE
SEQ
  handshake ! 0 -- dummy message
  in ? msg     -- buffer message
  out ! msg   -- forward message

Figure 6.5: Handshaking buffer process.
PROC MergeBuf (CHAN OF ANY left, right, out)
[buffer.size]INT msg :

WHILE TRUE
SEQ
ALT -- wait for either left or right
left ? msg
SKIP
right ? msg
SKIP
out ! msg

Figure 6.6: The merging buffer process.

SEQ PRI PAR
{{{
buffers run at PRI for speed
PAR
... SplitBuf (t.in, t.thro, t.to.buff, t.hand, buff.hand)
... HandshakeBuf (t.thro, t.out, t.hand)
... HandshakeBuf (t.to.buff, buff.to.w, buff.hand)
... FwdBuf (r.in, r.thro)
... MergeBuf (w.to.r, r.thro, r.out)
}}}
Worker (buff.to.w, w.to.r)

Figure 6.7: The main-node process structure.

buffer. The handshaking buffers are there simply to ensure that the MBs are not sent anywhere before they can be processed. Processed MBs are returned from the worker process to a merging buffer (Figure 6.6). The merging buffer forwards processed MBs both from its own worker and those coming back up the chain from lower processors.

The buffer processes are run at high priority so as to ensure that none of the PEs further down the chain are deprived of data (Figure 6.7).

The end-node is distinct in that it is the last PE in the chain and therefore is not
required to forward messages to and from PEs further down the pipe. Hence the process structure is greatly simplified and only three forwarding buffers (Figure 6.8) are necessary (Figure 6.9); two to buffer the links and the third to store a second task for the worker.

In order to verify that the chain topology performs as poorly as predicted, the DCT/inverse DCT coding loop was implemented on a balanced chain. Figure 6.10 shows the timing results for the chain topology compared to a random graph which confirms the theoretical prediction. The disparity is apparent for chains involving more than two worker PEs. Above 16 workers the simulation using the chain topology slows down, whereas the simulation using the random graph topology continues to make gains. Thus the chain topology was not considered further
in this study. The chain has an optimum length which is application specific. In general, shorter chains perform better than longer ones since the amount of communication is reduced.

A better idea would be to organise the PEs into parallel chains, three of which could be attached to the three available links on the master PE. For a given number of PEs, these parallel chains are shorter than a single chain and they operate concurrently. However, each chain still inherits the deficiencies of chains in general and so, for medium to large numbers of PEs, compact graph topologies were preferred.
6.5 Graph generation and analysis

Fortunately the generation and analysis of PE graphs was easily automated. Conventional sequential C programs were written to provide a toolset for graph generation, analysis and to produce wiring files for the ECS (Section 5.7).

6.5.1 Analysis

Having generated the processor graphs according to these various schemes, it was of interest to see how the mean inter-node distances varied with the number of nodes. Figure 6.11 shows this result plotted with the number of processors on a logarithmic scale. It is clear that the 'greedy' graph scales the best, although the random graphs scale almost as well. Since the plots appear to be almost linear for both 'greedy' and random graphs, it can be deduced that their rate of increase is $O(\log n)$, where $n$ is the number of processors. This is verified by Prior et al. [157]. The chordal graph plot increases much more quickly, and it is therefore concluded that chordal rings do not scale well. Browne and Hodgson [161] give the chordal ring mean inter-PE distance as being at best $O(\sqrt{n})$. Since the advent of topology independent routers, the programmer is no longer concerned with keeping the processor topology simple and regular. This result indicates that regular graphs do not scale as well as irregular ones. Figure 6.12 shows how the H.261 simulation execution times for a QCIF frame varied for these three topologies, for up to 64 worker PEs. The variation is small, as would be expected from the results shown in Figure 6.11; for small numbers of transputers (i.e. less than twenty) the mean inter-PE distances are almost indiscernible.

The random graphs were optimised for low inter-PE distances and no account was taken of potential communication bottle-necks. This can be seen in Figure 6.12
CHAPTER 6. SIMULATION RESULTS

Figure 6.11: Mean inter-processor distance for various processor graphs

Figure 6.12: Timings for the parallel H.261 simulation (without motion compensation) using a) occam & Tiny and b) ‘C’ & CS Tools.
by the fact that the execution time for the random topology increases when moving from 16 up to 33 workers, and then the execution time decreases again for 64 workers. Hence this indicates that the 33 worker random graph is not a good one and may contain bottle-necks. The symmetry of chordal rings guarantees no bottle-necks, provided that the computation is evenly distributed. It is also unlikely that the 'greedy' graphs will contain bottle-necks, since at each stage the link placement strategy aims to minimise the graph diameter; when each link is placed, the diameter must shift to another area of the graph which is not as well connected. In this way the communications bandwidth is evenly distributed throughout the graph.

6.5.2 Wiring file production

The wiring files used to semi-automate the wiring of the ECS domains are of a simple fixed format. The first line in the file is an integer which is the number of wiring lines to follow. All processors are numbered starting from 0 (the host) and upwards, in integer steps. If a graphics processor is present in the domain, it may be referred to either as the last PE in the list or \(-1\). This has the advantage that the same-wiring file may be used for different sized graphics domains. The following lines all specify two links to be connected, \(e.g.\ 0 \ 1 \ 1 \ 0\) indicates that link one of processor zero (the host) should be connected to link zero of processor one. Thus once the table of PE interconnections is created by one of the other programs above, the wiring file is easily generated.
6.6 Visualisation of intermediate results of the algorithm

One of the primary aims of these simulations was that the user should be able to view the output from any stage of the algorithm. This was one of the main reasons for choosing the compact graph topologies in which the master and all of the worker PEs are as 'close' to the graphics PE as possible. Also this is why the mean inter-PE distance was minimised in the graph optimisations. Unfortunately the ECS graphics domains contain a single graphics PE to service all graphics requests. Thus a severe bottle-neck is present. Figure 6.13 shows how displaying graphical output prevents the simulations benefiting from additional worker PEs. This is due to a combination of two factors:

1. Since the graphics part of the program increases the sequential part of the program significantly Amdahl's Law dictates a dramatic reduction in the maximum possible speedup.

2. The number of packets being communicated is increased by approximately a third which leads to much higher chances of link contention and through-routing delays.

The solution adopted was to allow the user of the simulation to turn the graphics on or off as desired. If required, the results could be stored to disk and displayed off-line in real time (Section 6.6.1).

6.6.1 Occam routines for visualisation

This section describes the various processes which were written to aid the visualisation of image results. These include colour-space conversion, histogram
CHAPTER 6. SIMULATION RESULTS

Figure 6.13: The effect on speedup of all worker PEs sending results to both the graphics and master PEs.

equalisation and off-line real-time display.

Histogram equalisation

The luminance histogram of typical linearly quantised images is usually very unevenly skewed. The majority of pixels are below the average value. The result is dull, flat-looking images in which detail is difficult to see. The image contrast can be enhanced by a technique known as histogram equalisation [45]. This was achieved by finding the range of luminance values and then stretching this range so that it used the total range available. The process structure is shown in Figure 6.14. The process was written in such a way as to work for any size input array. For the simulations following recommendation H.261 strictly, the image contrast enhancement was not employed since it would change the pixel values and hence invalidate the simulation results. However, it was found useful for clearer viewing
CHAPTER 6. SIMULATION RESULTS

-- Contrast is a function of luminance (Y) only.
-- Here we take Y and make its values use the full range, 0 .. 255

PROC real.high.contrast (CHAN OF ANY debug, [] REAL32 Y)
  ...
  SEQ
      ...  How big is it?
      ...  Find min and max pel values
      ...  Calculate the range used, hence mult factor required
      ...  highten the contrast of [] Y

Figure 6.14: Process to maximise contrast

of such luminance images as produced by the DCT, DPCM and motion estimation.

Enhancing transform space

Coefficients in transform space can be both positive and negative. For the 8 x 8 DCT, the possible range of values is -2047 .. 2048 and is continuous. Thus before they can be displayed as pixels, they must be made positive and scaled to the display range, 0 .. 255. A typical block in transform space contains a relatively large DC term; the rest of the coefficients are much smaller in size. In order to see detail in these transformed blocks, it was found useful to perform some kind of non-linear scaling on the coefficient values. The required effect was to relatively reduce the larger values and increase the smaller values. The technique of taking the square root of all pixel values before performing histogram equalisation was found to give acceptable results. Experiments taking the fourth root instead of the square root were also conducted. Both were successful in providing different levels of enhancement.
CHAPTER 6. SIMULATION RESULTS

Image display

Both master and the worker processors needed to be able to display images; the workers only processed macro blocks whereas the master dealt with complete QCIF frames. Sometimes the images were colour, other times they were monochrome. Further, it was necessary to be able to display images in which colour has no real significance. For example an image in cosine transform space does not benefit from having its luminance and chrominace values combined — the result is just a mess of colour. Thus there was a need for processes which could display both these sizes of image at various magnifications and in either monochrome or full colour. To this end various processes were written for displaying MBs (colour and monochrome) and QCIF frames (colour and monochrome). These were used many times for displaying the results from all stages of the algorithm simulation (Section 6.6.2).

Colour-space conversion

The routine to perform conversion from $YUV$ colour-space to $RGB$ was used a great deal (Section 2.4.1). Thus it was well worth optimising. Basically, it is a 3-dimensional transformation achieved by means of matrix multiplication:

$$
\begin{bmatrix}
R \\
G \\
B
\end{bmatrix} =
\begin{bmatrix}
1.000 & 0.000 & 1.400 \\
1.000 & -0.344 & -0.713 \\
1.000 & 1.772 & 0.000
\end{bmatrix}
\begin{bmatrix}
Y \\
U \\
V
\end{bmatrix}
$$

(6.7)

In order to ensure maximum speed of execution, all loops were removed. In addition the redundant multiplications by zero were avoided. Unfortunately, once the value of each pixel was calculated, it was required in order to check whether clipping was necessary to keep the result within the byte range, 0 . . . 255. This was unavoidable
since numbers outside this range would cause a software error on the graphics processor. This routine was called by all processes that display YUV colour images on the RGB graphics board.

Real-time display

From a close examination of the timing results of the display processes, it was determined that the main problem was having to perform the YUV to RGB colour conversion on every pixel before it could be packaged with others and sent to the graphics processor. A simple arrangement of two transputers was constructed using just a master and a graphics processor (Figure 6.15). The image files were converted from YUV format to RGB format off-line. Next the RGB files were loaded into the RAM of the master processor. From there they were sent to the graphics processor using all available link bandwidth. It was found that in this manner, the displaying of colour QCIF frames at 10 frame/s was just possible. This was without any other processing in parallel and without any delays due to larger communication distances, link contention and through-routing. Thus, the aim of displaying real-time simulation results on the ECS was abandoned. It was known to be possible to use this two PE configuration to display results off-line if desired. The software to achieve this was written.

6.6.2 Examples of output

In this section, examples of the output of the H.261 simulations are presented.

Still frame DCT

Figure 6.16 shows, at three times magnification, the absolute luminance values of a frame in DCT space. The surrounding frames display the effect of continually
halving the number of DCT coefficients which are retained per block, before performing the inverse DCT. All 64 coefficients are retained per block in the top-left frame; only the single DC coefficient for each block is retained in the bottom-right frame. This indicates how the picture degrades when less information is transmitted. When DPCM is introduced to the coding loop, the degradation due to the loss of transform coefficients is dramatically reduced. This is because only the 'prediction errors' are being transformed instead of the whole pixel values. However, this 'still-frame' output is an interesting demonstration of the operation of the DCT.

Motion estimation

Successful motion estimation for a MB is demonstrated in Figure 6.17. On the left is the search area from the previous frame, while on the right is the MB from the current frame. The white square in the search area moves vertically and horizontally until the 'best match' is found — i.e. the position at which the MB difference is the smallest. Using this enlarged output, the 'jumps' of the motion estimation search were clearly observed for individual macro blocks.
Figure 6.16: Video output showing a complete frame in DCT space (top-right), and the results of progressively setting more of the coefficient values for the higher spatial frequencies to zero (top-left to bottom-right).
Figure 6.17: Video output showing motion estimation. The white square indicates the 'best match' in the search space for the block on the right.
Figure 6.18: Video output showing two sequential frames (bottom); luminance of frame difference (top-left); DCT space of frame difference (top-right).

Frame differencing: DPCM

The result of DPCM coding is displayed in Figure 6.18. Two successive image frames are displayed at the bottom. At the top left is the luminance of the absolute frame difference; light areas indicate large differences due to motion. The frame difference in the transform domain appears in the top right of the Figure.

Cosine transform space

The effect of enhancing an image in transform space (Section 6.6.1) is shown in Figure 6.19. The non-linear function allows detail to be seen in the high frequency coefficients which would otherwise appear as all black.
Figure 6.19: Video output showing non-linear enhancement of transform-space enlarged.
Figure 6.20: Video output showing (left) an unprocessed frame and (right) a low pass filtered frame.

**Low pass filter**

The effect of applying the low pass filter (Section 5.4.1) to a still frame is shown in Figure 6.20. The edges are blurred but no visible blocking effects are introduced. The blurring is undesirable, but H.261 only applies the filter to MBs which have been motion compensated. Also when the pictures are moving and DPCM is employed, the blurring is much less noticeable.

**Transmission buffer overflow**

Without ME, as the simulation progresses excessive motion can cause the transmission buffer to become full and information to be lost. The visual result of this is seen in Figure 6.21. As the subject moves to the right, the quantisation becomes coarser and hence less accurate; many more of the small DPCM coefficients are
Figure 6.21: Video output showing the visual effects due to coarse quantisation when the transmission buffer overflows.

set to zero in an attempt to compensate for the full transmission buffer. In this case, the rather bizarre effect is that the bright-intensity face moves but the low intensity hair remains left behind as part of the background.

6.7 Performance evaluation

The simulation results have been described in some detail above. This section discusses how well they performed, highlighting areas where there is room for improvement.
### Table 6.2: Timings for DCT/IDCT on a QCIF frame with different numbers of tasks buffered.

<table>
<thead>
<tr>
<th>Workers</th>
<th>Zero tasks buffered (s)</th>
<th>1 or 2 tasks buffered (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6.25</td>
<td>5.99</td>
</tr>
<tr>
<td>2</td>
<td>3.16</td>
<td>3.02</td>
</tr>
<tr>
<td>4</td>
<td>1.64</td>
<td>1.52</td>
</tr>
<tr>
<td>8</td>
<td>0.86</td>
<td>0.80</td>
</tr>
<tr>
<td>16</td>
<td>0.47</td>
<td>0.44</td>
</tr>
</tbody>
</table>

6.7.1 Task buffering at the worker PEs

The idea of buffering tasks at the worker processes was described in Section 5.5.2. The effect of buffering zero, one or two tasks at the worker processes is shown in Table 6.2. It is clear that buffering more than one task at the worker PE does not result in greater efficiency. It is concluded that the communications are already maximally overlapped with the computation. However, buffering a single task at the worker does result in an increase in efficiency. This occurs because the transputer is able to use its links and CPU concurrently. Thus, it receives the second task while processing the first task. This is buffered while the first task is processed. Once the first task is processed and passed back to the routing harness, the CPU can begin to process the buffered task without any delay waiting for the next task to arrive.

6.7.2 H.261 simulations: occam/Tiny vs. C/CS Tools

This section compares the results obtained when using occam/Tiny with those obtained using C/CS Tools. Figure 6.12 shows plots of execution time for the H.261 algorithm (without ME) against increasing numbers of transputers. Results are presented for simulations using occam with the Tiny routing harness and C
with CS Tools. These simulations used random Hamiltonian processor topologies but, as explained in Section 5.7.1, the performance of 'greedy' graphs and chordal rings is virtually identical for these numbers of PEs. Though the C serial programs are considerably faster than those written in occam, Tiny is more efficient than CS Tools. The CS Tools simulation seems to saturate at about four workers, whereas gains continue to be made using up to 64 workers with Tiny. Using Tiny, approximately two frames can be processed in one second by 64 worker PEs. This is a fifth of real-time. It seems that the selected grain size is probably better matched to the operation of Tiny than that of CS Tools. The simulation appears to be communications bound at this point but we may be observing the effects of Amdahl's Law. Clearly, adding more PEs is not benefiting this simulation.

### 6.7.3 Parallel program efficiency

The efficiency of a parallel program can be defined as follows

\[
\text{Efficiency} \overset{\text{def}}{=} \frac{T_{\text{calc}}}{T_{\text{calc}} + T_{\text{comm}}} \leq 1
\]  

(6.8)

where \(T_{\text{comm}}\) is the non-overlapped communication time and \(T_{\text{calc}}\) is the total computation time, reflecting that any inefficiency is due to the communication overheads. Thus it is desirable to minimise non-overlapped communication and maximise the overlap of communication with computation as provided by the transputer. Section 5.5 described how the selected implementations achieved these aims. The equation is difficult to use in this form since \(T_{\text{comm}}\) is not easily determined. An equivalent definition of efficiency which does not require communication and computation times to be calculated is
Figure 6.22: Efficiencies for H.261 simulations using a) C & CS Tools and b) occam & Tiny.

\[ \text{Efficiency} \overset{\text{def}}{=} \frac{T_1}{NT_N} \equiv \frac{\text{Speedup}_N}{N} \leq 1 \quad (6.9) \]

where \( T_N \) is the total execution time for \( N \) processors and \( T_1 \) is the execution time for a single processor. Speedup is as defined in Section 6.2.1. Figure 6.22 shows efficiency curves for the simulation timings shown in Figure 6.12. It is clear that Tiny is more efficient than CS Tools. The former maintains 50% efficiency for up to 30 worker PEs, whereas the latter can only manage that for around 10 workers PEs.

### 6.8 Simulations from the viewpoint of the user

Several problems were encountered in implementing these simulations. These included displaying results in real time (Section 6.6.1) and, for example, how the
simulation parameters such as transmission buffer state can be traced by the user. This section describes how the simulations appear to the user and how the above problems were dealt with.

The standard sequences of images used consisted of up to 38 frames, each sampled at a frame-rate of 10 Hz. It was found desirable that the user should be able to select at which frame to begin the simulations and how many frames should be processed before the simulations terminated. This is of importance since some parts of the sequences contain more motion than others and therefore put greater demands on, for example, the motion compensation and the transmission buffer. Once the user selects the range of frames, the appropriate image files are read from disk into the RAM of the master PE.

Feedback to the user of information such as transmission buffer overflow was greatly facilitated by the debug channel provided by the Computing Surface. The master processor is the only one connected to the host and so is able to talk directly to the user. However, the Computing Surface supervisor bus (Section 3.6.1) allows all processors to send messages to the user's console. Thus it was straightforward to report on the various parameters of the algorithm simulation at run-time, rather than having to dump them to a file to be examined later off-line.

At the beginning of the execution of the simulation, the user is asked for various pieces of information:

- Which image files should be used as input?
- Are execution times required?
- Should the graphical results be displayed?
- Should debugging information be reported?
These factors are important because they are interconnected and they affect the speed of the simulations. For example, there is little point in taking timing results whilst displaying graphical result or reporting debugging information, since the latter two slow down the simulation considerably. The user is asked for the information in this way so that flags can be set up in the program. Subsequently re-compilation of the program (which is tediously slow) is avoided when, for example, debugging information is required. These flags are sent to the worker processors along with their MB data to be processed. The MB is packaged up with the flags and other information such as where the MB is positioned in the frame, and sent to the worker as one contiguous message. On receiving the messages, the workers unpack and interpret the flags thus distributing them globally across the processes of the parallel program.

6.9 Summary and conclusions

This chapter presents the results gained from the various techniques which were investigated in this thesis. Firstly timing is discussed and the H.261 sequential function timings achieved in occam and C are presented. The load-balancing chain implementation is described and its disadvantages are illustrated by means of the results it yielded.

Next, graph analysis is discussed in detail. The H.261 simulation results using these compact graph PE topologies are presented, compared and contrasted in terms of speedup and efficiency. Also the results of the occam and Tiny scheme are compared to those achieved using C and CS Tools.

Finally, simulation visualisation and usability were considered.
Chapter 7

Summary and conclusions

The advent of ISDN has made it possible for many more communications applications to be implemented using wideband switched digital networks. These include the transmission of voice, data and images. Despite the vast improvement in bandwidth and reliability that ISDN provides over the old PSTN\textsuperscript{1}, complex and computationally intensive algorithms are required to compress moving colour images sufficiently so as to allow their transmission at real-time rates.

Designing these algorithms and developing them into a standard presently takes many years; by the time that they are standardised, they cannot be up-to-date. A recent example of such an algorithm is the current international standard for videophone image compression, CCITT recommendation H.261. The design process would be greatly aided if it were possible to perform software simulations of an algorithm in real time. The computational complexity demands that parallel computers are used to perform such simulations. The work in this thesis has been to develop flexible techniques for simulating H.261 which can be applied generally to the problem of real-time simulation on multicomputers.

\footnote{PSTN: public services telephone network \textemdash i.e. the present analogue telephone network}
The following section summarises the techniques used in this work to simulate the H.261 algorithm. Conclusions derived from the research are given, and finally suggestions for future directions of the work in this thesis are presented.

7.1 Thesis summary

Chapter 2 provided an overview of the history, state-of-the-art, and future video-phone image compression algorithms. The current international standard, CCITT recommendation H.261, was described in detail.

The next two chapters laid out the necessary background to programming parallel computers. In Chapter 3, parallel computer architectures were considered. The differences between MIMD and SIMD machines were discussed. In particular the architecture of the ECS was detailed and the processing element upon which it is based, the T800 transputer, was described. This led on to Chapter 4 which itemised some of the difficulties of programming a parallel computer. These included the problems of communication between PEs, load balancing and PE network interconnection topology. Mapping models are in their infancy and are currently difficult to apply to problems. Instead programmers tend to use stereotyped mapping paradigms such as event parallelism, data parallelism and data-flow parallelism. These were described and in the light of the problems encountered, the need for general purpose parallel computers was emphasised. There is a clear requirement for automatic parallelisation of programs before parallel machines can be fully exploited by the scientific and non-scientific communities. Parallel software toolkits such as CS Tools are now emerging but the use of automatic parallelising compilers is a long way off.

The novel research was presented in Chapters 5 and 6. Chapter 5 contained
the description of the simulation techniques developed and employed. The results of the simulations were given in Chapter 6.

7.2 Thesis conclusions

7.2.1 Achievements

Due to the advent of topology independent message routers, programmers are no longer tied to simple processor topologies with simple message passing protocols. Nor do they have to worry about rewriting communications software for each new application. Instead a highly optimised software routing harness can be written once and for all. The router can be topology independent and hence the programmer is free to experiment with any processor network topologies. The next phase will be to relinquish the handling of communications completely to a hardware packet switching network.

The results presented in this thesis demonstrate that although close-to-real-time simulation can be achieved using 20 or more transputers the execution speed of the simulations is bounded by communications overheads and Amdahl’s Law. However, using compact graph topologies it was found possible to maintain 50% efficiency for up to 60 worker transputers.

Perhaps the most important discovery is that for small numbers of transputers ($\leq 20$), provided all available link bandwidth is used, one topology performs as well as any other. It is only when larger numbers of PEs are utilised that the differing scaling properties come into play. In the past this fact has been hidden for two main reasons:

- Efficient topology independent routing harnesses have not been available and
so programmers have restricted themselves to simple topologies such as chains and trees which do not use all the available link bandwidth;

- Most transputer research has been conducted using either small numbers of transputers, or else larger numbers in a fixed topology (see, for example, the conference proceedings of Transputer Applications '91 [168]). The work presented here has concentrated on parallelism with larger numbers of PEs.

All three of the compact topologies experimented with are very easy to generate for any number of PEs; they provide a practical means of interconnecting processors having four interprocessor links and are incrementally extensible. For large numbers of PEs both greedy and random graphs perform well. The chordal ring does not scale well, and since routing algorithms are no longer a problem there is no reason to favour regular processor topologies. Irregular graphs have the lowest inter-node distances and scale well, but for up to 100 PEs, all the graphs perform virtually as well as each other. Thus it can be concluded that, for machines containing numbers of PEs of this order, topology is no longer such a burning issue and efforts should be concentrated on automating the mapping of the problem to the PEs.

Though the simulations were all performed on the H.261 algorithm, the results obtained are all generally applicable to real-time algorithm simulation on multicomputers. In Chapters 5 and 6 the random and 'greedy' topologies were both shown to have better scaling and performance characteristics than previously popular regular topologies such as chains, trees, or chordal rings. By employing either of these topologies to the task farm paradigm of mapping, extremely flexible simulation was made possible. By simulating the majority of the algorithm on each worker PE, it was possible to display the output of any stage.
CHAPTER 7. SUMMARY AND CONCLUSIONS

7.2.2 Comparisons with related work

There are three published related works on the parallel simulation or implementation of recommendation H.261 using multicomputers. All three were described in Section 5.9. The first by Ichikawa and Shamura [148], was a rather curious application of transputers. A total of 100 PEs were employed in an irregular topology tailored specifically to the H.261 algorithm and attached to custom frame-grabbing and display hardware. Real-time results were reported for QCIF data. It is not clear what their aims were; it would have to be assumed that they wished to design an actual H.261 videophone system, the software of which could be changed as minor algorithm improvements emerged. The system is inflexible and, in the opinion of the author, would have been far more efficiently implemented in hardware. This is because significant changes in the algorithm would probably affect the load balancing of this highly customised mapping. Their aims were clearly very different from the work in this thesis. By contrast, the work in this thesis did not achieve real-time results but did develop very flexible techniques for utilising multicomputers to simulate algorithms in general.

The work of Sexton [169] is more closely related in that his aim was to simulate RM6, the draft software specification at the time for H.261 simulation, for BTL's evaluation purposes. His multi-stage pipeline of load-balancing chains was designed in the days before topology independent routing harnesses were available. Thus a topology for which the communication patterns are simple was chosen. An unfortunate side-effect was that only the final results of the algorithm could be displayed, resulting in a low level of algorithm visibility. Sexton used a data-flow decomposition with each division being implemented as a load-balancing chain. Due to the long data paths of the topology through which all the data must flow, frame rates of around 0.1 Hz were achieved by Sexton for colour QCIF data. By
contrast, using Tiny and the random or ‘greedy’ topologies in a task farm scheme, the research of this thesis achieved an order of magnitude improvement: colour QCIF frame rates of around 1 Hz. The rates are approximate because they are data dependent. However, the results are comparable because they used the same standard image sequences as input data.

The third work was by Rudberg and Chong [170, 163]. They aimed to implement a LAN videophone based loosely on recommendation H.261. The speeds they report are achieved by changing the algorithm to be less computationally intensive — e.g. using only monochrome image data and not including motion compensation. Their implementations were all written in occam and did not employ a routing harness. Thus, like Sexton, they were limited to simple topologies. They chose the tertiary tree topology which suffers from the long communication paths as described in Section 5.6.2. The data has to flow from the root of the tree to the leaves and then back up again in order to be displayed, or in their case transmitted. Thus their mapping would not scale well should they gain access to more transputers. The compact graph topologies used in the work of this thesis have been shown to scale extremely well and to have very low average inter-PE distances.

7.2.3 Limitations and suggestions for future work

Difficulties in simulating a complex image compression algorithm in real-time have been discussed. The evolution of the implementation of the simulations was traced, from the use of occam with the Tiny routing harness, through ‘C’ with Meiko’s CS Tools. Unfortunately the hardware configuration of the ECS made real-time display of images physically impossible. This could be rectified by introducing more than one graphics processor to the graphics domains. Each would be assigned
Figure 7.1: Suggested architecture for improved performance.

a portion of the display and so the serial bottle neck at the graphics processor would be removed, and the inevitable inefficiencies predicted by Amdahl’s Law (Section 6.2.1) reduced.

Simply stated, Amdahl’s Law shows that better efficiencies can be achieved by reducing the portion of the parallel program which is of parallelisation degree 1 (sequential). In the simulations, these parts were on the master and graphics processors. The technique described above might also be applied to the master: several masters could be employed, each managing a section of the framestore and the equivalent part of incoming images. Each master would also maintain its own queue of worker PEs so that, in effect, there would be several processor farms. Each farm would handle its own section of the images. In this way, everything except the loading of the image files would be parallelised to some extent (Figure 7.1).
CHAPTER 7. SUMMARY AND CONCLUSIONS

From the results so far recorded (Chapter 6), it is clear that Tiny provides more efficient communications than CS Tools. Also the sequential 'C' programs run faster than the occam due to the availability of pointers in 'C' and the difficulties of optimising for the occam compiler. Thus the next stage will be to dispense with CS Tools and to combine 'C' with Tiny. Tiny provides lower-level communications primitives than CS Tools does. It is faster but more difficult to learn to use. Programs using Tiny are more difficult to maintain — even more so now that the EPCC (Edinburgh Parallel Computing Centre) has withdrawn support for it.

The actual simulation speeds achieved were an order of magnitude improvement on those of Sexton. However, they are still an order of magnitude away from real time. It is doubtful whether the above suggested solutions would make very significant improvements. The T9000 (Section 3.5.1) promises to provide an order of magnitude improvement in performance and will be binary compatible with the T800. Thus it should be possible to achieve real time simply by replacing the T800s with T9000s in the ECS.

Programming in occam on the ECS is very difficult. The OPS environment is totally novel. Thus the learning curve involved in using it is long. The support it provides is minimal making debugging of programs extremely difficult. The situation was vastly improved by Lyndon Clarke (the author of Tiny) who wrote Rian, an advanced terminal emulator, and by the recent release of a debugger by Meiko. The support team find the ECS difficult to manage and provide a reliable service. Even though for the past few years it has been Europe’s largest multicomputer, it is now looked on as an embarrassment. The problems are mainly due to the operating systems being non-standard ones. Anyone now choosing to program in occam is warned that no support will be provided. Users are rapidly migrating to the alternative machines which provide more robust environments and operating
systems such as the DAP and the Connection Machine.
References


REFERENCES


REFERENCES


REFERENCES


[51] R.J. Clarke. ‘Basic principles of motion estimation and compensation’. In *Digest of IEE Colloquium on Applications of Motion Compensation*, pages 1/1–1/7, October 1990.


REFERENCES


REFERENCES


REFERENCES


[93] Texas Instruments. ‘Parallel processing with the remarkable TMS320C40’. In ASPECTs: Application Specific Products for the complete market, page 1, 1991.


REFERENCES


REFERENCES


REFERENCES


REFERENCES


REFERENCES


Appendix A

Original publications

The work contained in this thesis has been reported in the following publications:


†Reprinted in this appendix.

Aspects of the work in this thesis were also presented in:

CONCURRENT TECHNIQUES FOR DEVELOPING MOTION VIDEO COMPRESSION ALGORITHMS

John A. Elliott†, P.M. Grant† and G.G. Sexton‡

† Department of Electrical Engineering, University of Edinburgh, Kings Buildings, Mayfield Road, Edinburgh EH9 3JL, Scotland, UK.
(fax: 031-622 4358; email: jae@uk.ac.ed.ee)

‡ Department of Electrical and Electronic Engineering, Newcastle Polytechnic, Ellison Building, Newcastle Upon Tyne, NE1 8ST, England, UK.

Recently the CCITT set an international standard for motion video codecs, known as H.261 [2], which took several years to develop [1]. It has become clear that there is a real need for fast processing of motion video signals. Given the computing power required and the speed of today's technology, simulations in real-time can only be approached through the use of parallelism. Fortunately the possibility of videophony through the arrival of ISDN coincides with the availability of parallel computer hardware.

Here we discuss parallel techniques for speeding up software simulations of video codec algorithms on a reconfigurable MIMD machine with distributed memory (multicomputer). Our multicomputer, the Edinburgh Concurrent Supercomputer, houses over 400 INMOS T800 transputer floating point microprocessors.

Traditionally parallel image processing problems have been solved using a pipeline [3]. Instead, we advocate the use of compact graphs [4] in conjunction with an optimised software routing harness. They can considerably ease the development of concurrent image processing software and, at the same time, provide results better than those obtained from specifically tailored topologies and software. Additionally, this technique scales to any number of processors with minimal effort. In this way, a complete simulation of H.261 has been written in occam2. Results are presented for topologies using from 2 up to 65 transputers.

References


TECHNIQUES FOR MOTION VIDEO PROCESSING ON TRANSPUTER-BASED RECONFIGURABLE MULTICOMPUTERS

John A. Elliott¹, J.M. Beaumont², and P.M. Grant¹

1 Introduction

The design of motion video image compression algorithms has been very time consuming process when carried out with the aid of serial computers. Without real-time or near real-time simulation aids it is difficult and tedious to assess the performance of different algorithms. Recently the CCITT set an international standard for motion video codecs, H.261, which took several years to develop [3]. This paper addresses the problem of such software simulations. Here we discuss speeding up simulations by the use of a concurrency.

2 The Edinburgh Concurrent Supercomputer

INMOS have produced a range of 16/32 bit microprocessors [4] called transputers which can either be used as single powerful processors or combined into modular, concurrent systems. They combine a central processing unit, four bi-directional serial communications links, and memory in a single device. There are 4 kbytes of internal memory and up to 4 Gbytes of external memory can be addressed. Serial communications between transputers is provided by four 20 Mbit/s links. These links are completely autonomous resulting in 80 Mbit/s bandwidth at the same time as CPU and floating point unit (FPU) operations are being performed.

The Edinburgh Concurrent Supercomputer (ECS) [7] is a Meiko Computing Surface which provides multi-user access to over 400 transputers. It is partitioned into variously sized single-user domains. The interconnection of the transputer or processing element (PE) links within a domain is determined by the user statically before run time by means of a wiring file. Thus a domain can be configured into almost any topology restricted only by the number links available.

3 Difficulties

3.1 The Mapping Problem

Graph theory can be applied to the problems of writing parallel programs. The problem is described as a process graph; the machine's topology is described as a processor graph. The general problem is that of finding the best mapping between these two graphs. The "best mapping" is usually defined as that which produces the fastest execution time. This is known as the Mapping Problem [1]. On reconfigurable machines such as the ECS, we have the additional responsibility of choosing a suitable processor graph rather than accepting a given fixed topology.

¹Department of Electrical Engineering, University of Edinburgh.
²British Telecom Research Laboratories.

IEE COLLOQUIUM, LONDON, UK, APRIL 1991
3.1.1 Topology Dependence

One of the main reasons that parallel programs are so difficult to write is the responsibility for the communications between PEs. Usually the links between PEs are too few to allow direct passing of messages; often they must be routed through other PEs. This means that the process at each node must shoulder the burden of message forwarding.

This makes each program extremely topology dependent — i.e. the programs cannot easily be ported to different machines or configurations. Rather than use the best topology for a particular problem, the programmer might end up choosing one for which the communications are easier to handle.

3.2 The Application in Hand

The work described herein is a simulation of the H.261 algorithm. A simplified flow diagram of the algorithm is given in figure 1.

4 Strategies

4.1 Routing Harnesses

A topology independent routing harness provides the illusion of asynchronous global connectivity. Thus all processes can communicate with all other processes via the harness without worrying about how to route the messages. It does not need a priori information about the topology. Since the communications are asynchronous, processes do not waste time waiting for end to end connectivity; they simply hand their messages to the harness and then continue doing useful work. The task of delivery is left to the harness. This has a dual advantage:

*IEEE Colloquium, London, UK, April 1991*
The program becomes less machine dependent and hence more portable.

Various processor graphs can be tried without having to alter the program.

### 4.2 Compact Graphs

In the past, parallel machines came with fixed topologies such as a hypercube, mesh, ring or some other regular topology. Recent research [5] has shown that regular topologies are rarely optimal. Furthermore, certain irregular topologies are shown to perform very closely to theoretical minima with respect to such measures as diameter, mean inter-processor distance and worst through-routing time. Due to this "closeness" of the nodes in these graphs they are termed compact graphs.

Of particular interest are random Hamiltonian graphs. A graph is termed Hamiltonian if there exists a path which visits every node exactly once before returning to its original node. Thus a random Hamiltonian processor graph is easily configured by wiring the PEs in a ring and then arbitrarily connecting up remaining pairs of links. The ring guarantees that there is at least one route between any pair of processors.

### 5 A Simulation

The aim was to achieve as close as possible to real time processing whilst enabling simulation and evaluation of the algorithm. We were able to use any of the graphics domains in the ECS simply by generating random Hamiltonian wiring files for the required size of domain. A block diagram of one such possible Hamiltonian topology is shown in figure 2.

Analysis of the problem led to the conclusion that three distinct processes were needed:

- A Master process which distributes the data and receives the results. The Master is also responsible for the updating of various parameters as the simulation results return. This process must run on the transputer which has one of its links connected to the host so that it can access the file store efficiently.

- A Worker process which receives data from the master and processes them. The processed data is returned to the Master and any interesting results can be sent to the graphics to be displayed. This Worker process is duplicated \( n - 1 \) times.

- A Graphics process supplies a client/server interface to the graphics board facilities. This is a stripped down version of Meiko's Gfx software. It writes directly to video RAM and provides a minimum of functionality since speed is of the essence.

Incoming data flows through the following processing blocks: Frame Differencing; Discrete Cosine Transform (DCT); Quantiser; Inverse Quantiser; Inverse DCT; Variable Length Coder (VLC). Thus the Worker processes can perform these functions on the data in sequence before returning the result to the master process. This avoids any excessive communications which might have been encountered with the pipelined approach.

The Master maintains the frame buffer and the VLC buffer as processed data arrive back from the Workers. He updates the quantiser step size accordingly as the simulation progresses.

Since the Master is communicating with all the Workers throughout the simulation, we want a topology which puts all of the Workers as close as possible to the Master so as
Figure 2: A random Hamiltonian topology for a graphics domain with 8 workers.

to minimise communications overheads. Thus a random Hamiltonian topology is ideal since its diameter is close to the theoretical minimum, whilst being extremely easy to construct.

5.1 Results

An original serial implementation coded at British Telecom Research Labs (BTRL) in 'C' on a VAX 750, ran at approximately 15 minutes per frame. Sexton’s parallelisation [6], which used 'C' code with a handwritten occam harness on a 14 transputer pipelined topology, produced speeds of less than 1 minute per frame.

Our simulations described here were written purely in occam and all message passing was handled by the ECS routing harness, Tiny [2]. Refer to the graph in figure 3. Speed up is almost linear for up to 16 worker processors indicating virtually no communications overhead. For larger numbers of workers the performance falls off rapidly. This would indicate that there is little to be gained by adding more transputers to the network.

6 Conclusions

Here we have discussed a system that can be used for the evaluation and development of motion video codec algorithms. As an example we described a simulation of the current international standard algorithm for motion video coding, H.261. For a real-time simulation we would need to process a frame in 0.1 seconds. However the speeds achieved here
Figure 3: *Times for processing one frame for varying numbers of worker transputers*
are a considerable improvement on previous work at BTRL and are perfectly adequate for codec algorithm development and evaluation.

Our strategy described in this paper, using a topology independent routing harness and a compact random Hamiltonian processor graph, is very flexible:

- Individual parts of the algorithm are easily modified without causing load balancing problems;
- Output from any or all of the stages of the algorithm can be displayed graphically;
- The processor graphs are very easily generated so this solution can be applied to any desired number of transputers.

7 Acknowledgements

The authors would like to thank BTRL for help with funding and permission to publish this paper. This work made use of the Edinburgh Concurrent Supercomputer facilities in the Edinburgh Parallel Computing Centre.

References


REAL TIME VIDEOPHONE IMAGE CODING ALGORITHM SIMULATION ON A CONCURRENT SUPERCOMPUTER

John A. Elliott†, J.M. Beaumont‡, P.M. Grant† and J.T.E. McDonnell†
† University of Edinburgh, UK
‡ British Telecom Research Laboratories, UK

Abstract
A flexible means of evaluating/developing videophone image compression algorithms is required. Here we analyse the mapping we presented in [1, 2] and contrast it with currently popular alternative ones. Additional results are presented for speed-up, efficiency and absolute simulation speeds achieved.

1 Introduction
In this work, our aim has been to provide a means of simulating videophone image compression algorithms in close to real-time. The simulation had to be flexible and provide visibility of the output of all the stages of the algorithm.

As a typically complex algorithm we took the recently standardised H.261 [3]. The computational complexity demands that we use concurrency to approach real-time.

In the following sections we describe our concurrent supercomputer and address how to map the problem onto its processing elements (PEs). In addition the problem of “best” topology for our application is analysed. Finally we present the results so far achieved.

2 Target Architecture
2.1 INMOS Transputer
The transputer, the world’s first computer-on-a-chip, combines a central processing unit, four bi-directional serial communications links, and memory in a single RISC device. There are 4 kbytes of internal memory and external memory can be addressed.

Serial communications between transputers is provided by four 20 Mbit/s links. The links and CPU run autonomously and concurrently. Thus to gain maximum concurrency the programmer should strive to overlap communications with computations on each processor. The IMS T800 is a transputer with a floating point unit which provides very efficient hardware for floating point operations.

2.2 Edinburgh Concurrent Supercomputer
The Edinburgh Concurrent Supercomputer (ECS) [4] is a Meiko Computing Surface which houses 425 T800s. These are shared amongst single-user domains of various sizes and resources — e.g. graphics board, frame grabber, etc. Each T800 has 4 Mbytes of external RAM. The domain topology is configured by the programmer prior to run-time. This allows for easy experimentation with various topologies.

2.3 I/O Problems
Ideal as the ECS is for image processing problems, our particular real-time simulation presents some special I/O bandwidth problems.

2.3.1 Input
Our simulations were performed upon standard image sequences such as “Clare” and “Miss America” stored on disk. Obviously, disk access times would prohibit real-time simulations at 10 frame/s. Instead, a sequence of up to 20 frames was read from disk into the RAM of the Master PE before simulation commenced. Once in RAM the image data could be accessed as quickly as possible.
2.3.2 Output

Displaying results in real-time proved more difficult. Actually displaying 10 frame/s RGB colour images was not a problem. However, the H.261 algorithm operates on YUV colour data which must be transformed pixel by pixel to RGB format before display on conventional monitors is possible. This amount of extra processing dramatically reduces the performance [2]. However, there exist hardware boxes which can do this conversion extremely quickly and so could remove the problem at a very small cost\(^1\). Alternatively the results can be stored to the disk in RGB format and then read back later for real-time viewing.

3 Simulation Analysis

3.1 Mapping Problem

We can describe our algorithm as a set of tasks. Next we need to map these tasks to our set of PEs to minimise the time for the set to complete execution. This “mapping problem” can be tackled by models which assign execution times and communication patterns to tasks.

Proposed mapping models [5, 6] do not match reality very closely and are also difficult to apply. Instead programmers have come to use stereotyped solutions and then perhaps they try to optimise the solution later using heuristics [7, 8]. Experience has shown that if we map the data space instead of the algorithm greater efficiency can be achieved and expensive, complex and inelegant solutions [9] can be avoided.

3.2 Stereotyped Paradigms

Basically there are three choices [10]:

2. Algorithmic or Multifunction Pipeline parallelism
3. Task Farming

or combinations thereof.

The first two suffer from the problems of scaling and load balancing and do not achieve very high efficiencies. In contrast, the latter provides automatic load balancing since task scheduling is dynamic and, using the topologies we suggest can also scale extremely well.

Thus our implementation uses the task farm paradigm of concurrent programming. As previously reported [12], we opted for an implementation with three distinct occam processes:

- **Master**: Distributes tasks, collates results and maintains global structures;
- **Worker**: Accepts tasks and performs the main coding functions;
- **Graphics**: Displays colour blocks received from any PE.

Having the complete main H.261 functions on each Worker meant that we were easily able to experiment with various topologies with little reprogramming.

3.3 Restrictions of H.261

For our particular problem the frame of data is already broken down into a hierarchical structure. From this, the smallest element upon which all the functions of the H.261 (DPCM, DCT, quantisation, variable length coding, etc.[12]) can be performed independently is the macro block (MB) which represents a colour picture area of 16 x 16 pixels. The MB was chosen as the grain size. Any smaller would not be practical to operate upon independently; any larger would not utilise the number of PEs available and allow good load balancing. There are 99 MBs in a single image frame.

3.4 Parallel Program Efficiency

The efficiency of a parallel program can be defined as follows

\[
Efficiency = \frac{T_{\text{computation}}}{T_{\text{computation}} + T_{\text{communications}}} \tag{1}
\]

reflecting that any inefficiency is due to the communications overhead. Thus we wish to minimise communications and maximise the overlap of communications with computation which the T800 can provide. Section 5 shows how our implementation reflected these aims.

---

\(^1\)British Telecom Research Laboratories use such boxes
4 Analysis of Topologies

4.1 Definitions

Distance: It can be shown that the time, \( T \), for a message of fixed size, \( w \), to travel between two PEs in a network is directly proportional to the number of links traversed called the distance, \( l \):

\[ T(l, w) \propto l \]  

Bottleneck: A link in the topology which carries a disproportional amount of messages. Its performance determines the overall performance.

Mean Inter-PE Distance: The average of the shortest paths between all pairs of PEs in the topology.

Diameter: The largest of the shortest paths between all pairs of PEs in the topology.

Scalability: A measure of topology performance as the number of PEs is varied.

Narrowness: The worst ratio of communications to computation power. An indication of whether certain links will have to do a lot of through-routing for other PEs.

Now, since \( w \) is fixed by our chosen granularity, which is the MB, we can only hope to minimise the communications by minimising the number of links involved in passing the messages between PEs. To this end we are interested in such measures of topology as diameter and mean inter-PE distance [13].

4.2 Topology Choice

Processor farms have often been implemented as chains (or load balancing pipes), triple chains (or parallel pipes), and tertiary trees [14]. However, the best topology is defined by the problem in hand, and we show that none of these is particularly suited to our application. The following discussion explains how we came to favour compact graph [13] topologies. Throughout we will assume one Master PE scheduling tasks, \( n \) Worker PEs processing them and one Graphics PE displaying their results.

Figure 1: Chain topology

Figure 2: Tertiary tree topology

Figure 3: Random Hamiltonian compact graph topology
4.2.1 Chain and Tree Topologies

Chains were used to implement a multifunction pipeline in the work immediately prior to this project which was carried out at British Telecom Research Labs (BTRL) by Sexton [15]. Figure 1 shows a typical chain topology. M represents the Master who distributes tasks to the numbered Workers on demand. The one advantage that chains have is that the routing algorithm for them is extremely easy to write. However, now that topology independent routing harnesses are available this is no longer of much importance.

The chain is an intuitively bad topology since two out of the four links are unused. This is expressed as a high narrowness:

\[ N_{\text{chain}} = \frac{n}{1} = n \]  

which represents the link to the Master having to ferry messages to all of the Workers.

Another problem is that the mean Master-Worker distance is large and directly proportional to \( n \):

\[ \mu_{\text{mem-}\text{worker}_{\text{chain}}} = \frac{n(n+1)}{2} \times \frac{1}{n} = \frac{1}{2}(n+1) \]  

Minimising \( \mu \) is desirable because we thereby

- disturb less PEs through-routing messages
- leave more bandwidth free
- reduce message source to destination time

A better topology for link utilisation is the tertiary tree (Figure 2). Equation 5 refers to the narrowness of the Master's links to the subtrees.

\[ N_{\text{tree}} = \frac{n}{3} \]  

Here more links are employed but still at the leaves only one of the four is used. All PEs are close to the Master, as shown by equation 6, but it is not obvious where to put the Graphics PE with which all the Workers must communicate.

\[ \mu_{\text{mem-}\text{worker}_{\text{tree}}} \propto \log_2 n \]  

The problem with the tree for our application is that the general mean inter-PE distance is large due to the leaves on separate branches being relatively distant.

Link loading on both the tree and the chain is unbalanced, severely so for the chain. Assume a message is sent from the Master to every Worker in each unit time-step. Then \( n \) messages pass through the first link, \( n-1 \) through the second, and so on down to the last link which only carries one message. As \( n \) grows, the links near the Master become bottlenecks and the bandwidth of the ones farthest away is underutilised, i.e. again we see the chain scales badly.

Generally, link loading is higher than it need be since for both topologies not all links are utilised and there are no alternate routes between PEs that a router might take advantage of during times of bandwidth saturation.

4.2.2 Compact Graph Topologies

In contrast to the chain and tree topologies, compact PE graphs perform very close to theoretical bounds for all the measures discussed so far [13]. Additionally the Graphics PE can be included anywhere in the network without introducing undue degradation [12]. The presence of alternative paths between PEs allows an intelligent router to maintain performance despite heavy communications loading.

5 Implementations

5.1 Chain Topology

None of the detailed timing results from Sexton's work [16] have been published. We therefore decided to implement a chain topology to verify the predictions made in our analysis. It was written entirely in occam without the aid of a routing harness.

For the sake of efficiency it is very important to provide buffer processes on all the links being used. This enables the communications to be truly overlapped with computation. The worker process requests a task from and delivers results to the buffers. In this way it is never forced to perform synchronised, blocking communication. The buffers run at high priority to ensure that no other processors are starved of tasks or results.
5.2 Compact Graph Topology

Since compact graphs require complex inter-PE communications patterns, their routing algorithms would normal be prohibitively difficult to implement. However, a topology independent routing harness provides the illusion to the programmer of global interconnectivity. Thus irregularly connected topologies are no more difficult to program than regular ones. Also communications are automatically run at high priority and are non-blocking. There is a slight routing overhead introduced compared to the direct use of links. However, this is a small price to pay considering the freedom of topology choice gained.

6 Experimental Results

6.1 Efficiency

Let us first define some terms:

\[
\text{speed-up} = \frac{T_{\text{serial}}}{T_{\text{parallel}}} \tag{7}
\]

\[
\text{efficiency} = \frac{\text{speed-up}}{n} \leq 1 \tag{8}
\]

We expect to see the law of diminishing returns; as \(n\) increases, the efficiency decreases. However, the compact graph efficiency calculated from (8) falls off much less quickly than that for the chain (Figure 4). The chain can maintain at least 80% efficiency for up to 6 Workers whereas the compact graph can employ around 20 Workers for the same efficiency.

The speed-up graph in Figure 5 derived from (7) shows that adding more than 32 Workers to the chain actually reduces performance. It appears not to be worth using more than 16 Workers for the chain, at which point its efficiency is around 45%. By contrast, the compact graph maintains at least 40% efficiency for up to 64 Workers.

6.2 Task Buffering

By sending each worker two jobs initially, we were able to improve efficiency. The second job is buffered at the Worker PE until it is ready to process it. In this way there is always a job in hand and the communications overhead is

\(^2\)sometimes even random
minimised. An improvement in efficiency of 1–2% was observed due to task buffering.

6.3 Absolute Speeds

The complete occam H.261 simulation with motion compensation (MC) runs at 2–3 frame/s. Without MC, the most computationally intense part of H.261, the simulation runs at 3–4 frame/s. Real time implementation has been reported [9] using 100 T800s, but we sacrifice some speed here for flexibility. Our aim was to produce a simulation tool to aid algorithm evaluation/development with a high degree of visibility (see section 6.4).

Speeds so far quoted are for unoptimised occam. It is predicted that 6–7 frame/s will be achieved by optimising the occam code or using 'C' processes instead.

6.4 Algorithm Visibility

It was our aim to be able to see the output of any stage of the algorithm being simulated. Figure 6 shows, at three times magnification, the absolute luminance values of a frame in DCT space. The surrounding frames display the effect of increasingly reducing by a factor of two the number of DCT coefficients which are retained before performing the inverse DCT. All coefficients are retained in the top left frame; only the single DC coefficient for each block is retained in the bottom right frame.

In Figure 7, successful motion estimation for a MB. On the left is the search area from the previous frame. On the right is the MB from the current frame. The white square in the search area shows the "best match" found.

DPCM is displayed in Figure 8. Two sequential images are displayed at the bottom. At the top left is the luminance of the frame difference. Top right shows the frame difference in DCT space.

6.5 Other Considerations

Initially all coding of the processes was done in occam. The purity of occam makes it ideal for constructing parallel programs from serial processes. Its lack of semantic richness however, coupled with its unfamiliar development envi-
8 Acknowledgements

The authors would like to thank BTRL for help with funding and permission to publish this paper. This work made use of the Edinburgh Concurrent Supercomputer facilities in the Edinburgh Parallel Computing Centre.

References


