A Parallel implementation of an mpeg-2 encoder using message-passing

Jennifer Zenner

Follow this and additional works at: https://scholarworks.rit.edu/theses

Recommended Citation

This Thesis is brought to you for free and open access by RIT Scholar Works. It has been accepted for inclusion in Theses by an authorized administrator of RIT Scholar Works. For more information, please contact ritscholarworks@rit.edu.
A Parallel Implementation of an MPEG-2 Encoder
Using Message-Passing

by

Jennifer L. Zenner

A Thesis Submitted
in
Partial Fulfillment of the
Requirements for the Degree of

Master of Science
in
Computer Engineering

Primary Advisor: __________________________

Dr. Muhammad Shaaban, Assistant Professor

Committee Member: __________________________

Dr. Andreas Savakis, Associate Professor

Committee Member: __________________________

Dr. Kenneth Hsu, Professor

Department of Computer Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, New York
August 16, 2002
Release Permission Form
Rochester Institute of Technology

A Parallel Implementation of an MPEG-2 Encoder
Using Message-Passing

I, Jennifer L. Zenner, hereby grant permission to the Wallace Library of the Rochester Institute of Technology to reproduce this thesis, in whole or in part, for non-commercial and non-profit purposes only.

_______________________________
Jennifer L. Zenner

8/10/02
Date
ABSTRACT

The days of film are waning as digital cameras and digital video cameras are becoming commonplace. Uncompressed digital video can consume large amounts of space, making it cumbersome to store efficiently. A method of video compression was developed by the Motion Pictures Expert Group (MPEG), and is now an international standard with the International Organization for Standardization (ISO). This thesis deals with the MPEG-2 Video standard, ISO/IEC 13818-2 [2].

The goal of this thesis is to explore the applications of MPEG-2 encoding in a parallel processing paradigm. To achieve this, a sequential MPEG-2 software encoder was obtained from the MPEG Software Simulation Group (MSSG) [18] and modified to be run, in parallel, on a cluster of single-processor Linux workstations using the Message Passing Interface (MPI) [11, 10, 3]. A multi-threaded pipeline of the encoding process was created using Pthreads [6]. The resulting pipelined parallel encoder has been shown to produce compliant elementary MPEG-2 bitstreams for progressive video sequences. Results of simulation showed that the parallel encoder always performed better than the sequential version as the number of processors scaled. However, it did not exhibit the ideal linear speedup that all parallel programs aim to achieve. This is due to the program executing on a set of resources not ideal for the multi-threaded pipeline.

The ensuing chapters will provide the motivation for this work, and an overview of MPEG in addition to parallel processing and programming. Also forthcoming will be how it was achieved and the results produced. Supplementary applications of this work will also be discussed.
# TABLE OF CONTENTS

List of Figures iii  
List of Tables v  
Acknowledgments vi  
Glossary vii  

## Chapter 1: Introduction 1

## Chapter 2: MPEG 4

2.1 Overview of MPEG ............................... 4  
2.2 MPEG-2 ............................................. 5  
  2.2.1 Basics ........................................... 5  
  2.2.2 Motion Estimation and Prediction .................. 11  
  2.2.3 Discrete Cosine Transform (DCT) .................. 14  
  2.2.4 Quantization and Variable-Length Coding .......... 15  
  2.2.5 Rate Control .................................... 19  
  2.2.6 Compressed MPEG-2 Bitstream ................... 22

## Chapter 3: Parallel Processing and Programming 24

3.1 Parallel Processing ............................... 24  
3.2 Parallel Architectures ............................ 27  
  3.2.1 Shared-Memory ................................. 28  
  3.2.2 Message-Passing ............................... 30  
3.3 Parallel Programming ............................. 33
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3.1</td>
<td>Parallel Language Constructs</td>
<td>34</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Parallel Virtual Machine (PVM)</td>
<td>38</td>
</tr>
<tr>
<td>3.3.3</td>
<td>Message Passing Interface (MPI)</td>
<td>39</td>
</tr>
<tr>
<td>3.3.4</td>
<td>Pthreads</td>
<td>40</td>
</tr>
<tr>
<td>3.4</td>
<td>Parallel Approaches to MPEG</td>
<td>42</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Message-Passing Approach</td>
<td>42</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Shared-Memory Approach</td>
<td>47</td>
</tr>
<tr>
<td>4.1</td>
<td>Sequential Encoder</td>
<td>50</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Input Parameters</td>
<td>50</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Code Structure</td>
<td>54</td>
</tr>
<tr>
<td>4.2</td>
<td>Parallelization of Sequential Encoder</td>
<td>57</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Message-Passing Parallelism using MPI</td>
<td>57</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Multi-Threaded Pipeline using Pthreads</td>
<td>60</td>
</tr>
<tr>
<td>5.1</td>
<td>Setup</td>
<td>67</td>
</tr>
<tr>
<td>5.2</td>
<td>Experimental Results and Analysis</td>
<td>68</td>
</tr>
<tr>
<td>6.1</td>
<td>Accomplishments</td>
<td>75</td>
</tr>
<tr>
<td>6.2</td>
<td>Limitations</td>
<td>75</td>
</tr>
<tr>
<td>6.3</td>
<td>Future Work</td>
<td>76</td>
</tr>
<tr>
<td>A</td>
<td>Appendices</td>
<td>77</td>
</tr>
<tr>
<td>A</td>
<td>Bibliography</td>
<td>78</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

2.1 MPEG-2 Encoder Flow Diagram ............................................. 6
2.2 MPEG-2 Inter-frame Dependencies ....................................... 7
2.3 MPEG-2 Video Layers ............................................................. 8
2.4 Motion Estimation Example .................................................. 12
2.5 Frame vs. Field-Based Macroblocks [12] .............................. 14
2.6 MPEG-2 Quantization Scan Orders ........................................ 17
2.7 Huffman Coding Example [20] ............................................. 18
2.8 MPEG-2 Video Compressed Bitstream [24] ............................ 23

3.1 Steps of Parallelization [8] .................................................... 25
3.2 Interconnection Schemes for Shared Memory Multiprocessors [8] ... 29
3.3 Mesh and Torus Network Topologies ..................................... 31
3.4 Binary Tree Topology ............................................................ 32
3.5 Four-Dimensional Hypercube [4] ......................................... 32
3.6 Butterfly Topology [8] ........................................................... 33
3.7 Deadlock .......................................................................... 37
3.8 Frame-based Parallelization Methods [22] ............................ 43

4.1 GOP to Processor Assignment .............................................. 59
4.2 MPEG-2 Pipeline Diagram .................................................. 60
4.3 State Machine for Pipeline Scheduling .................................. 61
4.4 Data Flow through Pipeline ................................................ 63
4.5 Diagram of Pipeline with Stalls .......................................... 64

5.1 BBC3 Simulation Results ..................................................... 69
5.2 Table Tennis Simulation Results .................................................. 70
5.3 Speedup vs. Number of Processors for Sequence Division Method [22] 73
# LIST OF TABLES

<table>
<thead>
<tr>
<th></th>
<th>Table Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Comparison of Encoding Order and Display Order</td>
<td>7</td>
</tr>
<tr>
<td>2.2</td>
<td>MPEG-2 Profiles and Levels [13]</td>
<td>10</td>
</tr>
<tr>
<td>2.3</td>
<td>Quantization Matrices</td>
<td>16</td>
</tr>
<tr>
<td>4.1</td>
<td>Frame Data Structures</td>
<td>56</td>
</tr>
<tr>
<td>4.2</td>
<td>Macroblock Information Structure</td>
<td>57</td>
</tr>
<tr>
<td>5.1</td>
<td>Execution Times</td>
<td>71</td>
</tr>
</tbody>
</table>
ACKNOWLEDGMENTS

I would like to thank everyone who supported or helped me in any way to complete this thesis:

- Sharon Bacanskas (Mama) – For her unfailing support, encouragement, and unconditional love.

- Dr. Muhammad Shaaban – For being my advisor and for giving me insight and input.

- Dr. Andreas Savakis – For providing help and counsel when needed.

- Dr. Kenneth Hsu – For his encouragement and interest in this thesis.

- Dr. Roy Czernikowski – For suggestions and pointers.

- William Scheidel – For his suggestions, criticism and friendship.

- Doug Hoffman – For being a sounding board and providing suggestions and help with the cluster.

- Rick Tolleson, Paul Mezzanini – For technical support with the cluster and other related things.

- Paul Volcko – For providing me with the MPEG-2 Video Standard.
GLOSSARY

B FRAME: Bi-directionally predicted frame.

DATA DEPENDENCE: When one task needs data that another task produces.

DCT: Discrete Cosine Transform.

GOP: Group of Pictures in a video sequence.

IDCT: Inverse Discrete Cosine Transform.

I FRAME: Intra-coded frame.


LOAD BALANCING: Assigning data to processes such that the amount of work performed by each process is equal.

MAD: Mean Absolute Difference.

MESSAGE-PASSING: The act of sending messages between tasks, processes or processors for communication.

MISD: Multiple Instruction-Single Data.

MIMD: Multiple Instruction-Multiple Data.

MPEG: Motion Pictures Expert Group.

MPI: Message Passing Interface.
MPMD: Multiple Program-Multiple Data.

MSSG: MPEG Software Simulation Group.

NETWORK TOPOLOGY: Configuration of the network connections between nodes.

NODE: Processor in a cluster.

P FRAME: Forward predicted frame.

PROCESS: A mechanism that executes tasks.

PTHREADS: Library to create threads in C programs.

PVM: Parallel Virtual Machine.

SHARED-MEMORY: Shared address space between processors.

SISD: Single Instruction-Single Data.

SIMD: Single Instruction-Multiple Data.

SMP: Symmetric Multiprocessor.

SPEEDUP: A factor indicating increase of speed in a parallel implementation over a sequential one.

SPMD: Single Program-Multiple Data.

TASK: Smallest amount of computation in a parallel program.
Chapter 1

INTRODUCTION

The days of film are waning as digital cameras and digital video cameras are becoming commonplace. Even cinemas are now turning to digital video technology. The larger the amount of digital video produced, the more there is a need for compression. Uncompressed digital video can consume large amounts of space, making it cumbersome to store efficiently. The standard method of video compression was developed by the Motion Pictures Expert Group (MPEG), and is now a standard with the International Organization for Standardization (ISO). This thesis deals with the MPEG-2 Video standard, ISO/IEC 13818-2 [2].

MPEG-2 video compression exploits the temporal and spatial redundancy of frames in a video sequence. This is done temporally by reordering the frames at the input of the encoder to allow both forward and backward motion estimation and prediction. Spatially, the frames are transformed into the frequency domain by the Discrete Cosine Transform (DCT), and higher-frequency components are discarded or quantized. Because the human eye cannot perceive higher-frequency detail, the removal of that information does not effect the visual quality of the video. Furthermore, quantization and Huffman coding are also used to compress the DCT coefficients.

Image processing techniques have long been considered ideal for parallelization due to the inherent parallelism in both low-level computations and high-level processing.
There have been many successful attempts at parallelizing MPEG. Such attempts include dividing frames among processors in message-passing architectures and dividing video sequences by groups of pictures (GOPs) [22, 21]. Other approaches consist of using shared-memory architectures to run multi-threaded parallel implementations [9].

This thesis explores the applications of MPEG-2 encoding in a parallel processing paradigm, with the goal of matching, or exceeding, performance of previous parallel approaches. Since video technology is rapidly becoming primarily digital, there is a need for faster processing. When digital video cameras are used to film entire movies, the volume of uncompressed video is extremely large. It would benefit the entertainment industry to use parallel implementations to decrease the time it takes to compress large quantities of digital video.

A parallel approach to MPEG-2 was implemented using a sequential MPEG-2 software encoder obtained from the MPEG Software Simulation Group (MSSG) [18] and modified to be run, in parallel, on a cluster of single-processor Linux workstations using the Message Passing Interface (MPI) [11, 10, 3]. The novel approach is a multi-threaded pipeline of the encoding process, created using Pthreads [6]. While the final code was not run on the ideal set of resources due to unavailability, the applications of this code to a cluster of Symmetric Multiprocessors (SMPs) will be discussed.

The organization of the remainder of this document is as follows: Chapter 2 gives some background on MPEG and discusses the specifics in detail; Chapter 3 discusses parallel processing and programming techniques, as well as other parallel
approaches to MPEG; Chapter 4 provides specific details on the pipelined parallel encoder and how it was implemented; Chapter 5 presents the results obtained from various simulations using the parallel encoder. Finally, Chapter 6 presents conclusions for this effort, including limitations and ideas for future work.
Chapter 2

MPEG

This chapter gives detailed information regarding the origins and future of MPEG, the specifics of MPEG-2 and the operation of an MPEG-2 encoder.

2.1 Overview of MPEG

In 1992, the International Organization for Standardization (ISO), along with the International Electrotechnical Committee (IEC), standardized the first of the digital video coding methods produced by the Motion Pictures Expert Group (MPEG). This standard is known as MPEG-1, or ISO/IEC 11172. Then, in 1994, MPEG-2 became ISO/IEC 13818. The main purpose of these MPEG standards was to facilitate storage of digital multimedia such as on CD-ROMs and Digital Video Disks (DVDs). Currently, the industry standard for DVD video compression is MPEG-2. Since MPEG-2 is a superset of MPEG-1, it is backward compatible with MPEG-1 video streams.

MPEG continued to refine and refocus their efforts on new standards for other forms of multimedia. In 1998, the MPEG-4 standard was finalized as ISO/IEC 14496. This standard uses media objects to describe the content of the video pictures, whether the objects are real or synthetically generated [15]. MPEG-7, or ISO/IEC 15938, was finalized in 2001, and is a metadata standard that goes hand-in-hand with MPEG-4. MPEG-4 serves to represent the content of pictures, while MPEG-7 serves to describe that content [14].
Currently, MPEG is working on the upcoming MPEG-21 standard, which deals with the global picture of various types of multimedia and their interoperability. It is being touted as the Multimedia Framework, and as stated in [5],

“The vision for MPEG-21 is to define a multimedia framework to enable transparent and augmented use of multimedia resources across a wide range of networks and devices used by different communities.”

2.2 MPEG-2

ISO/IEC 13818, also known as Information Technology – Generic coding of moving pictures and associated audio information, is a standard divided into 9 parts [7]. For the purposes of this thesis, the Video portion of this standard, ISO/IEC 13818-2 [2], will be discussed at length.

2.2.1 Basics

MPEG-2 is a video compression method based upon algorithms containing motion compensation and the Discrete Cosine Transform (DCT) [12]. For the individual frames to be encoded, they must first be represented in the proper format. MPEG requires the frames to be in YUV format. This format contains one luminance (Y) component, and two chrominance components (U, V). Luminance deals with the brightness of the picture, while chrominance deals with hue and saturation of color [25].

\[
\begin{pmatrix}
  Y \\
  U \\
  V
\end{pmatrix} = \begin{pmatrix}
  0.299 & 0.587 & 0.114 \\
  -0.147 & -0.289 & 0.436 \\
  0.615 & -0.515 & -0.100
\end{pmatrix}\begin{pmatrix}
  R \\
  G \\
  B
\end{pmatrix}
\]
The luminance-chrominance format is much easier for the human eye to discern. This format can be derived from the RGB color system, which is based upon the knowledge that the human eye senses all colors as some linear combination of the three primary colors: red, green and blue [25]. All luminance information is retained since it contains most of the spatial information. The chrominance components are subsampled both horizontally and vertically. For a ratio of 4:2:0, the chrominance components are subsampled at a 2:1 ratio in both directions. For a ratio of 4:2:2, the chrominance components are subsampled at a 2:1 ratio horizontally, and there is no subsampling in the vertical direction. The transformation from RGB to YUV is represented by Equation 2.1 [25].

A flow chart of the encoding process can be viewed in Figure 2.1.

![MPEG-2 Encoder Flow Diagram](image)

Figure 2.1: MPEG-2 Encoder Flow Diagram

As seen in the diagram, the first stage is preprocessing. During this stage, the encoder
determines the coding type of the current frame. There are three different frame

 types in MPEG: I, P, and B frames. I frames are intra-coded and contain all of the

 information of the original frame. P frames are predictive frames that are computed

 based upon a previous I or P frame. B frames are bi-directional frames that use either

 previous or future I or P frames, or both, to calculate the coded information for that

 frame. Figure 2.2 shows the dependencies between the frames in a typical MPEG

 sequence.

 ![Figure 2.2: MPEG-2 Inter-frame Dependencies](image)

 Also taking place in the preprocessing stage is the reordering of the incoming
 frames. This is done to exploit the temporal redundancy between the frames. In other
 words, since successive frames are not expected to be much different from the previous
 frame in the sequence, this is what allows the use of bi-directional interpolation for
 B frames. Table 2.1 shows how the encoding order and display order compare to one
 another.

<table>
<thead>
<tr>
<th>Display Order</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encoding Order</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>6</td>
<td>4</td>
<td>5</td>
<td>9</td>
<td>7</td>
<td>8</td>
<td>12</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>Frame Type</td>
<td>I</td>
<td>P</td>
<td>B</td>
<td>B</td>
<td>P</td>
<td>B</td>
<td>B</td>
<td>P</td>
<td>B</td>
<td>B</td>
<td>I</td>
<td>B</td>
<td>B</td>
</tr>
</tbody>
</table>

 Table 2.1: Comparison of Encoding Order and Display Order

 It can be seen from the above sequence that after reordering takes place, all necessary
I or P frames needed as reference frames to encode a B frame are encoded prior to that frame.

The next topic of importance is the structure of the MPEG sequence. A video
sequence can be viewed in terms of several layers. Figure 2.3 gives a visual representation. At the highest level is the sequence layer, which is the entire video sequence. There is no set upper limit for the number of frames allowed in the sequence. The lower limit is one frame. The second layer is the Group of Pictures (GOP). The boundaries of each GOP are delineated by the I frames in the sequence. The user can dictate the distance between successive I frames (known as $N$), as well as the distance between successive I and P frames (known as $M$). Typical values of $N$ range from 12 to 15, and values of $M$ range from 1 to 3 [25]. So, in Figure 2.3, $N$ is 12 and $M$ is 3. The third layer of the hierarchy is the Picture layer, which consists of one entire frame. Each frame can be divided into slices, which represent the Slice layer. The slices contain numerous Macroblocks, the fifth layer. MPEG-2 specifies that a slice may be no more than one row of macroblocks in a picture [19]. Each macroblock has dimensions of 16 pixels by 16 pixels for the Y component and 8 pixels by 8 pixels for the U and V components. Blocks are the smallest elements in the hierarchy, and have dimensions of 8 pixels by 8 pixels. For a sampling ratio of 4:2:0, a macroblock would contain 4 luminance blocks and 2 chrominance blocks, while a 4:2:2 ratio would have 4 luminance and 4 chrominance blocks. Another ratio not currently allowed in MPEG-2 is 4:4:4. There would still be 4 luminance blocks, but also 8 chrominance blocks.

Each layer is used in specific ways during the encoding process. In the motion estimation stage, which will be discussed later, motion vectors are calculated for each macroblock using a block-matching algorithm. The prediction stage uses the output of the motion estimation for each macroblock to predict the value of the current macroblock from the reference frame(s). Macroblocks are also used in the quantization
stage. Blocks are used for the DCT. GOPs are necessary because of the predicted frames. If there were no GOP structure, and the distance between I frames was an entire video sequence, an error in one P frame would propagate to all successive frames, rendering the encoded sequence incorrect. With the use of GOPs, I frames are guaranteed to occur at fixed distances. Therefore, any error in one GOP will not propagate to the next, provided the GOP is designated as closed. A closed GOP means the first set of B frames in the sequence cannot use predictions from any frames in the previous GOP. The frequent recurrence of I frames makes random access into the video sequence of a video player possible.

<table>
<thead>
<tr>
<th>Level</th>
<th>Simple</th>
<th>Main</th>
<th>4:2:2</th>
<th>SNR</th>
<th>Spatial</th>
<th>High</th>
</tr>
</thead>
<tbody>
<tr>
<td>High</td>
<td>4:2:0</td>
<td>1920x1152</td>
<td>90Mb/s</td>
<td></td>
<td></td>
<td>4:2:0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4:2:2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1920x1152</td>
<td>100Mb/s</td>
<td></td>
<td></td>
<td>4:2:0</td>
</tr>
<tr>
<td></td>
<td>4:2:0</td>
<td>1440x1152</td>
<td>60Mb/s</td>
<td></td>
<td></td>
<td>4:2:0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1440x1152</td>
<td>4:2:2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1440x1552</td>
<td>60Mb/s</td>
<td></td>
<td></td>
<td>4:2:0</td>
</tr>
<tr>
<td>High 1440</td>
<td>4:2:0</td>
<td>1440x1152</td>
<td>60Mb/s</td>
<td></td>
<td></td>
<td>4:2:0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1440x1552</td>
<td>4:2:2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1440x1552</td>
<td>80Mb/s</td>
<td></td>
<td></td>
<td>4:2:0</td>
</tr>
<tr>
<td>Main</td>
<td>4:2:0</td>
<td>720x576</td>
<td>720x608</td>
<td>4:2:2</td>
<td>50Mb/s</td>
<td>4:2:0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>720x576</td>
<td>720x576</td>
<td></td>
<td>15Mb/s</td>
<td>4:2:0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>720x576</td>
<td>15Mb/s</td>
<td></td>
<td>15Mb/s</td>
<td>4:2:0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>720x576</td>
<td>20Mb/s</td>
<td></td>
<td></td>
<td>4:2:0</td>
</tr>
<tr>
<td>Low</td>
<td>4:2:0</td>
<td>352x288</td>
<td>352x288</td>
<td>4:2:0</td>
<td>4Mb/s</td>
<td>4:2:0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>352x288</td>
<td>4Mb/s</td>
<td></td>
<td></td>
<td>4:2:0</td>
</tr>
</tbody>
</table>

Table 2.2: MPEG-2 Profiles and Levels [13]
MPEG-2 also has different combinations of profiles and levels that were defined with specific applications in mind. Table 2.2 shows the allowed combinations of profiles and levels, as well as some basic parameters. As can be seen in the table, not all profiles are defined for all levels. The default combination is Main Profile at Main Level, or MP@ML. For MP@ML, the maximum dimensions of a frame are set at 720x576 at a maximum bit-rate of 15Mb/s with a luminance-chrominance ratio of 4:2:0.

There are two types of frames in MPEG-2, progressive and interlaced. The frames discussed to this point are progressive frames, or frame-based pictures. Interlaced frames are

"...created by dividing each frame into a set of two interlaced fields, with odd lines from the frame belonging to one field and even lines to the other. The fields are transmitted in interlaced video one after the other, separated by half a frame time [12]."

Interlaced frames are generally referred to as field-based pictures in MPEG-2. Field-based pictures exploit the inability of the human eye to perceive high-resolution components by focusing on the low-resolution components. The original detail of the picture can be maintained without increasing bandwidth. Unlike MPEG-1, MPEG-2 can deal with both frame and field-based pictures.

2.2.2 Motion Estimation and Prediction

The next stage that takes place in the encoding process, shown in Figure 2.1, is motion estimation. The general purpose of motion estimation is to exploit temporal redundancy between frames to ultimately reduce the amount of spatial information
that must be encoded due to motion in a picture sequence. This is accomplished by a block-matching algorithm in which macroblocks of the current picture are compared with those of a reference picture. When a good match is found for the macroblock, motion vectors are created. The motion vectors serve as coordinates for the decoder to determine where in the reference picture to find the current macroblock. An illustration of this technique is provided in Figure 2.4. The solid arrow is pointing to the place in the reference frame that has the best match for that macroblock. The dashed line is the vector translation on the current frame from where the current macroblock is to where the previous one was.

![Motion Estimation Example](image)

**Figure 2.4: Motion Estimation Example**

There are several methods for block-matching, but for the purposes of this thesis, a simple technique called full-search block matching was used. This block-matching algorithm takes each macroblock of the current image and compares it to a fixed number of macroblocks in the reference frame. This fixed number comes from a pre-defined range surrounding each macroblock. Full-search block matching is exhaustive within the given range, and is highly computationally intensive. In reality, one cannot expect to get exact matches between macroblocks, because there are many different
facets of motion that can cause the image to appear larger, smaller, or otherwise distorted.

\[ D(s, t) = \frac{1}{lm} \sum_{j=1}^{l} \sum_{k=1}^{m} |f_n(j, k) - f_{n-1}(j + s, k + t)| \]  

(2.2)

To ascertain that a match has been made, the Mean Absolute Difference (MAD) is computed for each macroblock in the search range. The one with the smallest difference is considered the closest match. This difference is also encoded with the motion vectors such that the decoder can accurately reconstruct the moving object. Equation 2.2 is used to compute the MAD [25].

The block-matching described above is considered full-pixel searching. MPEG also allows half-pixel searching, in which values are interpolated at half distances both vertically and horizontally from neighboring pixels in the reference macroblock. The interpolated values are then used to compare with the current macroblock.

After motion estimation calculates all of the motion vectors for all macroblocks of a frame, the prediction stage uses the motion vectors to retrieve the reference macroblock so that it can be subtracted by the current macroblock. The resulting error is then passed on for further compression. As mentioned before, MPEG-2 allows both frame and field-based pictures. Prediction for field-based pictures is done differently from frame-based pictures. The field-based macroblock is split into two fields, top and bottom. The top field is then predicted from the top fields in the reference frame(s). The bottom field is likewise predicted from the bottom fields in the reference frame(s). Figure 2.5 shows the relative differences between frame and field-based macroblocks.
2.2.3 Discrete Cosine Transform (DCT)

Once the prediction errors have been tabulated, they are further compressed by the Discrete Cosine Transform (DCT) stage. From each macroblock, a number of blocks are extracted; 4 from the luminance macroblock and 1 from each of the chrominance macroblocks. The blocks are 8x8 in size and form the basic unit for the DCT.

\[
C(u, v) = \alpha(u)\alpha(v) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} f(x, y) \cos \left[ \frac{(2x + 1)u\pi}{2N} \right] \cos \left[ \frac{(2y + 1)v\pi}{2N} \right]
\]

for \( u, v = 0, 1, 2, ..., N - 1 \)

\[
f(x, y) = \sum_{u=0}^{N-1} \sum_{v=0}^{N-1} \alpha(u)\alpha(v)C(u, v) \cos \left[ \frac{(2x + 1)u\pi}{2N} \right] \cos \left[ \frac{(2y + 1)v\pi}{2N} \right]
\]

for \( x, y = 0, 1, 2, ..., N - 1 \)
\[ a(u) = \begin{cases} \sqrt{\frac{1}{N}} & \text{for } u = 0 \\ \sqrt{\frac{2}{N}} & \text{for } u = 1, 2, ..., N - 1 \end{cases} \] 

The MPEG standard does not define a specific way to accomplish the forward DCT, because it would significantly limit allowable implementations of the standard. The basic equations for the 2-dimensional DCT and IDCT are shown in Equations 2.3 and 2.4 [20]. The equation for \( a \) is given in Equation 2.5 [20]. The resulting 8x8 block will have a DC coefficient in the upper-left corner and AC coefficients in all other locations.

As it was for prediction, the DCT can be either frame or field-based. An interlaced picture should always be coded using field DCT, where the top and bottom fields are separated as shown in Figure 2.5. For frame pictures, it can be decided at the macroblock level which version to choose, as long as the type chosen is specified for each macroblock. The decision should be based on the type of motion in the picture. Frames with overall vertical motion are best coded as field frames, while frames with small areas of vertical motion can be coded as frames [19].

2.2.4 Quantization and Variable-Length Coding

Following the DCT stage is the quantization stage. Quantization is necessary because after the DCT, the resulting coefficients are usually 12 bits or more, which is an increase over the starting pixels represented at 8 bits each. There are two separate forms of quantization, intra macroblock quantization and non-intra macroblock quantization. Intra refers to macroblocks that were intra-coded, meaning not predicted.
Non-intra refers to macroblocks that were predicted. MPEG-2 specifies default quantization matrices for both types. Both matrices are shown in Tables 2.3(a) and 2.3(b) [25].

\[
\begin{array}{cccccccc}
8 & 16 & 19 & 22 & 26 & 27 & 29 & 34 \\
16 & 16 & 22 & 24 & 27 & 29 & 34 & 27 \\
19 & 22 & 26 & 27 & 29 & 34 & 34 & 38 \\
22 & 22 & 26 & 27 & 29 & 34 & 37 & 40 \\
22 & 26 & 27 & 29 & 32 & 35 & 40 & 48 \\
26 & 27 & 29 & 32 & 35 & 40 & 48 & 58 \\
26 & 27 & 29 & 34 & 38 & 46 & 56 & 59 \\
27 & 29 & 35 & 38 & 46 & 56 & 69 & 83 \\
\end{array}
\]

(a) Intra Quantization Matrix

\[
\begin{array}{cccccccc}
16 & 17 & 18 & 19 & 20 & 21 & 22 & 23 \\
17 & 18 & 19 & 20 & 21 & 22 & 23 & 24 \\
18 & 19 & 20 & 21 & 22 & 23 & 24 & 25 \\
19 & 20 & 21 & 22 & 23 & 24 & 25 & 27 \\
20 & 21 & 22 & 23 & 24 & 25 & 27 & 28 \\
21 & 22 & 23 & 24 & 25 & 27 & 28 & 30 \\
22 & 23 & 24 & 25 & 27 & 28 & 30 & 31 \\
23 & 24 & 25 & 27 & 28 & 30 & 31 & 33 \\
\end{array}
\]

(b) Non-intra Quantization Matrix

Table 2.3: Quantization Matrices

\[
Q_{DC} = \frac{dc}{8} \quad (2.6)
\]
\[
Q_{DC}(9\text{bit}) = \frac{dc}{4} \quad (2.7)
\]
\[
Q_{DC}(10\text{bit}) = \frac{dc}{2} \quad (2.8)
\]
\[
ac(i, j) = \frac{(16 \times ac(i, j))}{W_N(i, j)} \quad (2.9)
\]
\[
QAC(i, j) = \frac{[ac(i, j) + \text{sign}(ac(i, j) \times ((p \times mquant)/q))]/(2 \times mquant)}{(2 \times mquant)} \quad (2.10)
\]
\[
ac(i, j) = \frac{(16 \times ac(i, j))}{W_N(i, j)} \quad (2.11)
\]
\[
QAC(i, j) = \frac{ac(i, j)}{(2 \times mquant)} \quad (2.12)
\]
The calculations for quantization are different for the resulting DC and AC coefficients of the DCT. For the DC coefficient, the quantizer step sizes are 8, 4, 2 and 1. Equations 2.6, 2.7 and 2.8 show the calculations required for the DC coefficient [25]. Note that the division in these equations represents integer division, meaning the result is rounded to the nearest integer. Results less than 1 are rounded to zero. For the AC coefficients, Equations 2.9, 2.10, 2.11 and 2.12 apply [25]. Integer division is also utilized in these equations. Equations 2.9 and 2.10 are used for intra-coded macroblocks. The $W_i(i,j)$ term in Equation 2.9 refers to the specific location in the intra quantization matrix. In Equation 2.10, the $mquant$ factor refers to an adaptive quantization factor which will be explained in the next section. In the MSSG code [18], the factor $p$ is 3 and $q$ is 4. Equations 2.11 and 2.12 are used for non-intra coded macroblocks, where $W_N(i,j)$ refers to the $(i,j)th$ location in the non-intra quantization matrix.

There are two ways to scan the coefficients in order to quantize them. Figure 2.6 shows the zig-zag and alternate scan orders. The coefficients are processed in zig-zag order because the most energy is usually concentrated in the lower-order coefficients. It also allows for more efficient run-length coding [25].

![Zig-Zag Scan](image1.png)  ![Alternate Scan](image2.png)

Figure 2.6: MPEG-2 Quantization Scan Orders
After quantization, the resulting quantized coefficients are then run-length encoded using Huffman coding. Huffman coding is guaranteed to create the optimal set of codes for each symbol. In this case, the symbols are the quantized coefficients. The first step of Huffman coding is to order the symbols by decreasing probability of occurrence. The next step is to combine the lowest two probabilities and reorder the remaining probabilities. This continues until there are only two probabilities left. Finally, the codes are assigned starting with the final two probabilities, which each get 0 and 1. The two codes that created the combined probability then get a 0 or a 1 appended to that code, and so on. Figure 2.7 shows an example of Huffman coding with arbitrary symbols and probabilities.

<table>
<thead>
<tr>
<th>Original Source</th>
<th>Probability</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbol</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a2</td>
<td>0.4</td>
<td>1</td>
<td>0.4</td>
<td>0.4</td>
<td>0.4</td>
</tr>
<tr>
<td>a6</td>
<td>0.3</td>
<td>00</td>
<td>0.3</td>
<td>00</td>
<td>0.3</td>
</tr>
<tr>
<td>a1</td>
<td>0.1</td>
<td>011</td>
<td>0.1</td>
<td>011</td>
<td>0.2</td>
</tr>
<tr>
<td>a4</td>
<td>0.1</td>
<td>0100</td>
<td>0.1</td>
<td>0100</td>
<td>0.1</td>
</tr>
<tr>
<td>a3</td>
<td>0.06</td>
<td>01010</td>
<td>0.1</td>
<td>0101</td>
<td></td>
</tr>
<tr>
<td>a5</td>
<td>0.04</td>
<td>01011</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.7: Huffman Coding Example [20]

MPEG uses modified Huffman codes to reduce the amount of memory taken up by codes with very small probabilities. For example, consider a group of sources that consist of $2^v$ binary sequences of length $v$. Then consider that the probabilities are distorted such that there are large numbers of improbable symbols. The improbable symbols are assigned to a new category, named ESCAPE [12]. The remaining symbols are divided into two disjoint categories, $S_1$ and $S_2$, which are described by Equations
2.13 and 2.14 [25]. After all the symbols are grouped into $S_1$, $S_2$ or ESCAPE, the same Huffman coding as shown in Figure 2.7 is performed on the new groups. The exception is that the resulting code for ESCAPE is added as a prefix to the existing codes in that group. All other codes in $S_1$ and $S_2$ remain the same.

$$S_1 = \left\{ s_i \mid p(s_i) > \frac{1}{2^v} \right\} \quad (2.13)$$

$$S_2 = \left\{ s_i \mid p(s_i) \leq \frac{1}{2^v} \right\} \quad (2.14)$$

MPEG also uses runlength amplitude coding, in which repeated sequences of numbers are given their own Huffman codes, which further improves the compression ratio.

2.2.5 Rate Control

The encoder uses Rate Control to ensure the video sequence being encoded will not overflow the decoder buffer. This is done with the use of a virtual buffer verifier (VBV). Since the standard does not specifically define how the encoder should accomplish rate control, the MSSG [18] rate control method will be discussed. There are three steps to this rate control method: Target Bit Allocation, Rate Control, and Adaptive Quantization.

**Target Bit Allocation**

This step is needed to estimate the number of available bits for coding the current picture. Several factors, such as picture type, buffer fullness and picture complexity influence the estimation. The complexity factors are determined by the number of bits used and the quantization parameter for the same type of previous picture in the current GOP.
\[ X_i = 160 \times \frac{\text{bitrate}}{115} \]  
(2.15)

\[ X_p = 60 \times \frac{\text{bitrate}}{115} \]  
(2.16)

\[ X_b = 42 \times \frac{\text{bitrate}}{115} \]  
(2.17)

\[ X_i* = S_i Q_i \]  
(2.18)

\[ X_p* = S_p Q_p \]  
(2.19)

\[ X_b* = S_b Q_b \]  
(2.20)

\[ T_i = \max \left\{ \frac{R}{1 + \frac{N_p X_p}{X_i K_p} + \frac{N_b X_b}{X_i K_b}}, \frac{\text{bitrate}}{8} \times \frac{\text{picture}}{\text{rate}} \right\} \]  
(2.21)

\[ T_p = \max \left\{ \frac{R}{N_p + \frac{N_b X_b}{X_b K_p}}, \frac{\text{bitrate}}{8} \times \frac{\text{picture}}{\text{rate}} \right\} \]  
(2.22)

\[ T_b = \max \left\{ \frac{R}{N_b + \frac{N_p X_p}{X_p K_p}}, \frac{\text{bitrate}}{8} \times \frac{\text{picture}}{\text{rate}} \right\} \]  
(2.23)

Equations 2.15, 2.16 and 2.17 are the initial complexity measures for I, P and B frames, respectively. Equations 2.18, 2.19 and 2.20 show how those complexity measures are updated [25]. In the update equations, the \(S\) terms are the number of bits used to encode the previous same type of frame, and the \(Q\) parameters are the previously used quantization values for the same type of frame. Using these numbers, the target number of bits is then computed for the specific frame type using Equations 2.21, 2.22 and 2.23. \(K_p\) and \(K_b\) are constants dependent on the quantization matrices. For the purposes of this thesis, they are 1.0 and 1.4, respectively. \(R\) is the remaining number of bits assigned to the GOP. \(N_p\) and \(N_b\) are the remaining number of P and B pictures left in the GOP [25].
Rate Control

In this step, the buffer fullness is used to calculate the quantization step size.

\[
\begin{align*}
  d_j^i &= d_0^i + B_{j-1} - \frac{T_i(j - 1)}{MB\_cnt} \\
  d_j^p &= d_0^p + B_{j-1} - \frac{T_p(j - 1)}{MB\_cnt} \\
  d_j^b &= d_0^b + B_{j-1} - \frac{T_b(j - 1)}{MB\_cnt}
\end{align*}
\]  

(2.24)  

(2.25)  

(2.26)

\[
\begin{align*}
  d_0^i &= 10 * r / 31 \\
  d_0^p &= K_p * d_0^i \\
  d_0^b &= K_b * d_0^i \\
  Q_j &= \frac{d_j * 31}{r}
\end{align*}
\]  

(2.27)  

(2.28)  

(2.29)  

(2.30)

\[ r = 2 * \text{bitrate/picture} - \text{rate} \]  

(2.31)

Equations 2.24, 2.25 and 2.26 calculate the virtual buffer fullness. \( MB\_cnt \) is the total number of macroblocks in the picture, and \( B_{j-1} \) is the total number of bits generated by encoding all macroblocks in the picture up to \( j \). The terms \( d_0^i, d_0^p \) and \( d_0^b \) are the initial values of buffer fullness, calculated by Equations 2.27, 2.28 and 2.29, respectively [25]. The quantization parameter is calculated via Equation 2.30, and parameter \( r \) is the reaction parameter, given by Equation 2.31[25].

Adaptive Quantization

The final step to rate control is adaptive quantization. The premise behind this step is that the human eye is more sensitive to active areas of an image than static areas. Therefore, the quantization step size obtained from the previous step is increased for
active areas and decreased for static ones.

$$\text{act}_j = 1 + \min_{s_{blk}=1,8} (\text{var}_s_{blk})$$  \hspace{1cm} (2.32)

$$\text{var}_s_{blk} = \frac{1}{64} \sum_{k=1}^{64} (P_k - P_{\text{mean}})^2$$  \hspace{1cm} (2.33)

$$P_{\text{mean}} = \frac{1}{64} \sum_{k=1}^{64} P_k$$  \hspace{1cm} (2.34)

The first calculation for the current macroblock is the spatial activity measure. This is done using four frame-based blocks as well as four field-based blocks with half-pixel interpolated values. Equation 2.32 shows this calculation, where \(\text{var}_s_{blk}\) is the variance of each 8x8 block. This calculation is shown in Equation 2.33. In that equation, \(P_k\) is the pixel value in the original frame and \(P_{\text{mean}}\) is the mean value of the block, calculated as shown in Equation 2.34.

$$N_{\text{act}_j} = \frac{2 * \text{act}_j + \text{avg}_\text{act}}{\text{act}_j + 2 * \text{avg}_\text{act}}$$  \hspace{1cm} (2.35)

$$m\text{quant}_j = Q_j * N_{\text{act}_j}$$  \hspace{1cm} (2.36)

Next, the normalized activity factor is calculated using Equation 2.35, where \(\text{avg}_\text{act}\) is the average of \(\text{act}_j\) for the last encoded picture. Finally, the modulated quantization step is obtained from Equation 2.36 [25].

2.2.6 Compressed MPEG-2 Bitstream

Now that the mechanics of MPEG-2 have been explained in detail, it is necessary to show how all the information calculated, or created, is expressed in the compressed bitstream. Figure 2.8 shows this in detail. The actual coded values for each field can be found in the MPEG-2 Video standard [2].
The next chapter discusses parallel processing and programming techniques. Previous parallel approaches to MPEG are also discussed.
This chapter will explore the need for parallel processing and available architectures. Various methods of parallelization and programming techniques will also be discussed. The chapter concludes with a discussion of previous parallel approaches to MPEG.

3.1 Parallel Processing

Traditionally, programs written for everyday use are sequential. A single processor is used to execute one line of code at a time. For programs with trivial computations, this method is sufficient. However, there are many applications requiring intensive computations which do not perform well on uniprocessor systems. Examples of these are programs involving numerical modeling, engineering calculations and weather forecasting [4]. To facilitate the these types of applications, parallel processing is necessary. The goal of parallel processing is to “...improve performance by obtaining speedup over the best uniprocessor execution [8].”

Not all programs are good candidates for parallelism. Since parallelism relies on the ability to perform tasks concurrently, programs whose tasks are highly dependent upon the data produced in other tasks will not benefit. For this reason, it is necessary that the sequential program be thoroughly understood such that any inherent parallelism may be exploited. To begin this process, three concepts of parallelism must be defined. A task is a unit of computation performed by the program. Each
task is executed by *processes*, which execute on *processors* [8]. The program is then reorganized using the steps shown in Figure 3.1.

![Figure 3.1: Steps of Parallelization [8]](image)

**Decomposition**

Decomposition refers to taking the sequential program and breaking it into smaller tasks which will run concurrently. Ideally, processes should be busy at all times in a parallel program to justify the overhead used in creating them. If there are too many concurrent tasks and not enough work to be done in those tasks, the cost of the overhead will outweigh the computational benefit. Therefore, the goal in this stage is to expose enough concurrency such that all processes are busy at all times. The effect of the concurrency in a program can be calculated using Amdahl’s Law, shown in Equation 3.1 [8]. The *s* term is the fraction of the computation that is sequential, and the *p* term is the number of processors used to perform the computation.

\[
\text{Speedup}(p) = \frac{1}{s + \frac{1-s}{p}}
\]  

(3.1)
Tasks may have a varying amount of work associated with them. This is known as task granularity [8]. The granularity of a task may range from fine-grained to coarse-grained. Fine-grained tasks involve little computation, whereas coarse-grained tasks involve intensive computation. Tasks that fall in the middle are called medium-grained tasks. The granularity of a task becomes important when considering the amount of work each process must complete.

\[
\text{Communication to Computation Ratio} = \frac{\text{Computation time}}{\text{Communication time}} = \frac{t_{\text{comp}}}{t_{\text{comm}}} \tag{3.2}
\]

An important consideration when determining task granularity is the expected communication to computation ratio, shown in Equation 3.2 [4]. The amount of communication between tasks should not outweigh the amount of computation being performed. Therefore, the ratio should be as high as possible without compromising the concurrency of the program.

**Assignment**

In the assignment stage, the concurrent tasks are assigned to processes. Tasks may be assigned statically before execution, or dynamically at run-time. The goal here is to ensure that the workload of each process is relatively equal. This is also known as load balancing [8]. Decomposition and assignment together are known as partitioning, and are largely independent of the parallel architecture.
Orchestration

Orchestration is heavily dependent upon the architecture used. In this step, all facets of process communication, synchronization and data access are determined. The way in which data is organized can highly impact the degree of communication needed between processes. Any data dependencies between tasks are addressed in this stage to avoid violating dependencies. Other goals of this step are to reduce communication between processes and reduce synchronization needed [8].

Mapping

Mapping is the final stage of the process, shown in Figure 3.1, which involves mapping the processes onto processors. This can be achieved explicitly within the programming environment, or implicitly by allowing the operating system to schedule the processes. In either case, it is dependent upon the environment being used. Mapping heuristics within different network topologies remains an active area of research interest.

3.2 Parallel Architectures

In deciding which architecture to use for a specific program, it helps to have a method of classification. Such a classification exists in computer architecture and is known as Flynn’s taxonomy [8]. The following are those classifications:

- **Single Instruction-Single Data (SISD)** - This is a typical sequential computer, where one instruction can operate on one set of data at a time.

- **Single Instruction-Multiple Data (SIMD)** - SIMD involves one instruction operating on different sets of data within synchronized processing elements. Vector processors fall into this category.
• *Multiple Instruction-Single Data (MISD)* - In this classification, there are multiple instructions executing simultaneously on a single set of data. This lends itself to pipelining, where different stages operate concurrently on the same data set.

• *Multiple Instruction-Multiple Data (MIMD)* - MIMD is the classification for parallel machines which are usually built of multiple conventional processors. This classification has two sub-classifications - *Single Program-Multiple Data (SPMD)* and *Multiple Program-Multiple Data (MPMD)*. The SPMD classification is used when a single program is run on different processors that are not synchronized. All processors may not execute the same sections of code in the program. MPMD is used when there are multiple copies of multiple programs executing on different processing elements. For example, in a master-slave configuration, one program runs on the master and another program runs on the slaves. Likewise, the master-slave configuration is possible under SPMD, where portions of a program are designated for the master processing element, and others are designated for the slaves [4].

Once a program is classified, the specific architecture used to run the program can be determined. The following sections will expand upon various parallel architectures.

### 3.2.1 Shared-Memory

Shared-memory multiprocessors are extremely important architectures in the world of parallel processing. Processes implicitly communicate through memory accesses. This architecture type provides better throughput for multi-threaded programs, and parallel programs that must have access to large amounts of global data. The idea of shared-memory multiprocessors is an extension of a conventional computer. In a
conventional system, the processor shares address space with other peripherals such as I/O devices. This concept is extended to include other processors and additional memory modules. There are different interconnection schemes for shared-memory systems. The configuration impacts the memory access latency if all processors are not equidistant from all memory elements. Figure 3.2 gives some typical examples of shared-memory multiprocessor interconnection networks. Note that the “S” indicates the cache for different processors.

A very specific and widespread type of shared-memory multiprocessor is the symmetric multiprocessor (SMP). In SMP architecture, cache access and memory access latencies are identical among processors. A commercially available SMP is the Intel quad-processor Pentium Pro, which is a bus-based shared-memory multiprocessor [8].
3.2.2 Message-Passing

Message-passing architectures are multi-computer based. Instead of having multiple processors inside one system, there are many single-processor systems, usually referred to as nodes, that are connected through a network. Each node maintains separate memory from all other nodes. The only way one node can retrieve data from another is to retrieve it explicitly with a send/receive operation. This is the major difference between message-passing and shared-memory architectures. The major advantage of message-passing architectures over shared-memory architectures is scalability [4]. It is much easier to connect another node to an existing networked cluster than it is to integrate another processor into a shared-memory multiprocessor system. Clusters are also easily upgraded to the newest technology, whereas shared-memory multiprocessors are not.

Since all communication between nodes is explicit, the type and configuration of the interconnection network is crucial. Some important network design issues are bandwidth and network latency. Bandwidth is typically measured in bits/s and indicates the amount of data that can be transmitted over the network per second. The network latency is the time it takes to transmit data through the entire network. Also important is the communication latency, which is the total time a message takes to be completely transferred, including startup overhead. Another factor in determining delay is the minimum number of links between the farthest points on the network. This is called the diameter of the network [4].

There are many network configurations. The following is a list of some of the available configurations and their descriptions:
• *Completely Connected* - In this network topology, all nodes are connected to all other nodes through one switch.

![Mesh and Torus Network Topologies](image)

Figure 3.3: Mesh and Torus Network Topologies

• *Mesh and Torus* - Mesh topologies are based on grids, where the nodes are connected to their nearest neighbors. The nodes on the boundaries have less connections than those in the center of the grid. Tori extend the mesh such that each boundary node connects to the boundary node on the other end of the grid. In the cases of the corner nodes, there are connections both vertically and horizontally. Figure 3.3 shows both configurations.

• *Line/Ring* - Linear topologies consist of nodes that are only connected to their two neighboring nodes, with the exception of the two end nodes. A ring topology extends the line by connecting the two end nodes to each other.

• *Tree* - A binary tree network begins with the root node at the topmost level. The root has two links to other nodes. Those nodes also have two links, and so on. The tree topology may also be extended from binary to m-ary, where m is
the number of links between the nodes in the hierarchy. Figure 3.4 shows the binary tree.

![Binary Tree Topology](image)

Figure 3.4: Binary Tree Topology

- **Hypercube** - In a hypercube topology, each node connects to one node in each dimension of the cube [4]. Hypercubes in their simplest form are three-dimensional. However, they can be extended to higher order dimensions as needed. Figure 3.5 shows a four-dimensional hypercube.

![Four-Dimensional Hypercube](image)

Figure 3.5: Four-Dimensional Hypercube [4]

- **Butterfly** - The butterfly topology is an extension of the tree topology; the difference being multiple roots. It is composed of "...2 x 2 blocks that correct
one bit in the relative address [8].” Figure 3.6 shows the basic building block and a 16-node butterfly network.

3.3 Parallel Programming

Various parallel languages and coding techniques exist to facilitate parallelization of sequential programs. This section discusses the basic constructs of both message-passing and shared-memory parallel programming. Parallel Virtual Machine (PVM), the Message Passing Interface (MPI) and Pthreads are discussed as specific examples.
3.3.1 Parallel Language Constructs

All parallel programming languages have core sets of functions required by the specific architecture to exploit parallelism. This section will discuss the basic constructs of message-passing and shared-memory parallel programming.

Message-Passing Programming

The first and most basic ability of message-passing languages is to create processes. This is usually done by a spawn function. This spawn function may register the process with the programming environment through a task identifier or a name of some kind. Some languages allow spawning of processes within other spawned processes. The spawn function initiates a previously compiled program or piece of code that will serve as the process.

Once processes are created, they must have some way of communicating with each other, especially since the nodes in the message-passing architecture maintain separate memory. This is accomplished through the use of send and receive functions. The source process initiates the send routine. The data being passed must be specified in the call, as well as the destination process id. The destination process must execute a matching receive routine, indicating the destination for the incoming data and the process id of the sender. The send and receive routines may be synchronous, meaning that they will not allow the calling processes to proceed until all data has been successfully transferred and received. These types of send/receive pairs are also called blocking routines [4]. Conversely, there are non-blocking send/receive pairs. This indicates that the sending process will return immediately after transferring the data, regardless of completion of the matching receive routine. These routines require
the use of a message buffer to store incoming data transfers until the receive routine is completed. Messages may also be identified by an extra message tag parameter. If used, the destination process will only accept messages from the specified source with the specified tag. This can be used to ensure that multiple messages from the same source are received in the proper order. Other constructs may also be implemented to allow any process to receive the sent message.

Another form of communication between processes is collective communication. This is when data is exchanged from one process to all other processes, or from all processes to all other processes. These functions are commonly known as broadcast, scatter and gather. In a broadcast routine, a source process will send data to all other processes in the group. Both the source process and the group of participating processes must be specified in the function call. A scatter routine will take pieces of data from a source buffer and scatter the elements across processes. For example, if a source process wished to scatter an array of three values across three processes, the first element would go to the first process (usually itself), the second element would be sent to the second process, and the third to the third process. The source specified in a gather process receives individual pieces of data from all other processes in the group and stores them in a buffer, which is specified in the call to the routine. A common addition to gather is the reduce function, which combines the gather operation with an arithmetic operation. The operation is specified in the function call, and may be anything from addition to computing the average of all pieces of data.
Shared-Memory Programming

As with message-passing, shared-memory programming languages need process or thread creation routines. These are commonly called *fork* or *create* functions, and are similar to, if not identical to, *spawn*. Along with the creation routine is a *join* routine. Join serves to reunite forked processes with the parent process. So, if a parent process needed data from a child process, a join statement specifying the process id of the child would be executed. The parent thread then waits for the termination of the child thread before continuing with processing. Termination of the child process may happen implicitly when the function being executed by the child returns, or explicitly with an *exit* routine call. A detached thread is any thread that does not need to join with the parent. These threads may or may not be designated as detachable during creation, dependent upon the specific programming library.

Since shared-memory processes communicate implicitly, threads that access the shared data are subject to *race conditions*. This occurs when more than one thread tries to access the same resource at the same time [6]. To ensure shared resources are only accessed by one thread at a time, a technique known as *mutual exclusion* is applied [4]. To achieve mutual exclusion, a process will *lock* a shared resource. The lock signifies that only the locking thread may change the value of the specified data. Any other thread wishing to do so must wait for the lock to be removed by the locking thread, and then must apply its own lock to it.

Sections of code that modify shared data are called *critical sections* [4]. In threaded applications, the programmer must take great care to avoid a situation called *deadlock*. This occurs when two threads execute critical sections requiring the
resources that are locked by the other thread. Figure 3.7 illustrates this situation. Deadlock can be avoided by using semaphores to mutually exclude a critical section being executed by one thread from other threads. Semaphores are positive integers that are operated on by an increment function and a decrement function. At the beginning of a critical section, a thread will test the semaphore to see if it has been incremented by another process. If so, it will wait for the semaphore to be decremented by the same process before it enters. Since semaphores can be incremented past 1, two processes trying to execute the locked critical section can make decisions based upon the value of the semaphore.

Semaphores are only useful if the programmer remembers to do the bookkeeping involved with using them properly. One way to guarantee this is to use monitors. A monitor uses semaphores to control a critical section. The monitor then becomes the only way to access that critical section. Once a thread executes a monitor procedure, no other thread may execute that procedure. Other calling threads are placed in a queue and are allowed to access the monitor only when the currently executing thread has finished [4].
Critical sections may only need to be executed under certain conditions. For these instances, condition variables are used. To access condition variables, three operations exist: wait, signal and status. Wait is used by a process to wait for the condition variable to be set. The section of code that controls the condition variable will issue a signal on that variable when the condition has been met. A status call will return the number of processes waiting on the condition variable [4].

3.3.2 Parallel Virtual Machine (PVM)

PVM is a set of libraries that can be used in C or Fortran programs to run parallel programs. The number of processes PVM can create is not dependent on the number of processors being used. PVM will allocate the processes to processors automatically, unless the user specifies the allocation. This makes PVM useful for both shared-memory and message-passing architectures. The most common PVM programs are master/slave configurations, where one master process controls other slave processes.

To create and end a process, the routines pvm_spawn() and pvm_exit() are used. Each process is assigned a task identifier that is retrieved by executing pvm_mytid() at any time. The communication routines are pvm_send() and pvm_recv(). All PVM send routines are non-blocking, while the receive routines may be either blocking or non-blocking [4]. PVM uses buffers for sending and receiving. To send data, it must be packed into the send buffer using specific packing commands. Once packed, the send routine will initiate transfer of the data. A message tag is required as part of the send function call. On the receiving end, the data must be unpacked from the receive buffer. Collective communications are performed using pvm_bcast(), pvm_scatter(), pvm_gather() and pvm_reduce(). The group of processes wishing to communicate is
specified by the `pvm_joingroup()` routine.

### 3.3.3 Message Passing Interface (MPI)

Although MPI libraries are also used with C and Fortran, it is quite different from PVM. MPI is almost exclusively used for MIMD applications [17]. MPI does not explicitly create processes or threads. Instead, MPI uses processor ranks to identify the participating processors in the cluster. For a program to enter into the MPI environment, `MPI_Init()` must be called prior to any other MPI calls. To terminate the environment, the `MPI_Finalize()` routine must be called to exit properly. All processors specified at run-time are entered into a global group. A *communicator* is then created for the group called `MPI_COMM_WORLD`. This is the default communicator used for most communication schemes. The number of processors in the communicator can be extracted using `MPI_Comm_size()`. Each processor may retrieve its numerical rank within the communicator by executing `MPI_Comm_rank()`, specifying the global communicator and the address of the local rank variable. The rank is used to determine which sections of code should be executed by what processors. This is the SPMD programming method, discussed previously.

The basic MPI message-passing routines are `MPI_Send()` and `MPI_Recv()`, which are blocking functions. The send routine requires the address of the buffer where the data is contained, the number of elements in the buffer, the data type, the rank of the target processor, a message tag and the communicator. MPI defines many data types, such as `MPI_INT`, `MPI_CHAR`, `MPI_DOUBLE`, etc. The receive routine requires the address of the receive buffer, the number of incoming elements, the data type, source rank, message tag and an `MPI_STATUS` variable. MPI also has non-blocking ver-
sessions of the same functions, \texttt{MPI.\textbf{I}send()} and \texttt{MPI.\textbf{I}recv()}. To complete non-blocking communication operations, \texttt{MPI.\textbf{W}ait()} and \texttt{MPI.\textbf{T}est()} are used. The wait function waits for the specified request to complete, while the test function returns a Boolean integer indicating whether the operation has completed.

MPI also has the same collective communications functions as PVM, plus some additional ones. \texttt{MPI.Bcast()}, \texttt{MPI.Gather()}, \texttt{MPI.Scatter}, and \texttt{MPI.Reduce()} are the same functions as discussed previously. \texttt{MPI.Alltoall()} sends data from all processors to all other processors. \texttt{MPI.Reduce-scatter()} combines results from across processors, and distributes the result of that operation across all processors.

The basic send and receive routines provide an implicit synchronization within the program. To specify explicit synchronization between processes, \texttt{MPI.Barrier()} is used. The barrier function accepts the communicator of the group of processors that will be synchronized. As mentioned before, MPI uses communicators to group processors. It is possible to create communicators other than the default \texttt{MPI.COMM.WORLD}. This proves to be a powerful option when working with different network topologies by helping to map the programming environment onto the topology for faster processing.

3.3.4 \textit{P}threads

Pthreads is an implementation of the IEEE Portable Operating System Interface (POSIX) [4]. They are primarily used for shared-memory configurations. To create a pthread, the \texttt{pthreads\textunderscore create()} function is used. The create routine must be supplied with the address where the handle of the thread is stored, the user-defined attributes,
a void pointer to the function the thread is to execute and a void pointer to the
arguments structure. The handle is of type `pthread_t` and the attributes are of type
`pthread_attr_t`. The attributes should be initialized by the `pthread_attr_init()` function
and should be destroyed by a call to `pthread_attr_destroy()`. The detach state of the
thread is set by calling `pthread_attr_setdetachstate()` and is set to `PTHREAD_CREATE_JOINABLE` for threads that will be joined with the parent after termination. Those that will not be joined can be detached by `pthread_detach()` after they have been created as joinable. To join a thread after termination, the parent executes `pthread_join()`, which will cause the thread to wait for the child thread specified by the handle passed to the function to terminate. Since the create routine only accepts one argument pointer, structures are used to pass multiple arguments to the function being called by the thread [16].

As mentioned in the previous section on shared-memory parallel programming
constructs, mutual exclusion is a vital part of the programming process. Pthreads
uses `mutex` variables to accomplish this. The creation and destruction of mutexes
are achieved using `pthread_mutex_init()` and `pthread_mutex_destroy()`. To lock and
unlock mutexes, `pthread_mutex_lock()` and `pthread_mutex_unlock()` are used. A useful
function in preventing deadlock is `pthread_mutex_trylock()`, which will return an error
code if the mutex has already been locked [16].

Condition variables are created and destroyed using `pthread_cond_init()` and
`pthread_cond_destroy()`. The wait and signal routines are `pthread_cond_wait()` and
`pthread_cond_signal()`. Wait should only be called if the mutex has already been
locked, otherwise it will not properly block the thread. The mutex must be unlocked
after the signal function has been executed, or any threads waiting on the signal will always be blocked [16]. Another function exists for signaling multiple threads that are waiting on the same condition variable, called *pthread_cond_broadcast()*.

### 3.4 Parallel Approaches to MPEG

Image processing techniques have long been considered ideal for parallelization due to the inherent parallelism in both low-level computations and high-level processing. There have been many successful attempts at parallelizing MPEG. A few of the efforts influencing this thesis will be discussed in this section.

#### 3.4.1 Message-Passing Approach

In “Study of Data Distribution Techniques for the Implementation of an MPEG-2 Video Encoder [22],” T. Olivares et. al used the SPMD programming model with a cluster of workstations to parallelize an MPEG-2 encoder. The cluster consisted of 20 64-bit, 167MHz SUN Ultra-1 workstations interconnected by a ForeRunner ATM switch. MPI was used to facilitate inter-processor communication by mapping a two-dimensional grid onto the underlying architecture. This allowed each processor to have \( x \) and \( y \) coordinates, which were useful in identifying neighboring processors. Two separate parallelization methods were investigated: frame-based and sequence-based. Three methods of frame division were used: block, vertical and horizontal. Figure 3.8 gives a visual representation of the three methods and how they were mapped onto the processor grid.
Frame division

Mapping equations were defined for all three methods.

\begin{align}
\alpha &= \text{MOD}(M, (m \times 16)) \\
\beta &= \lambda(M - \alpha)/m \\
\gamma &= \text{MOD}(N, (n \times 16)) \\
\delta &= \lambda(N - \gamma)/n
\end{align}

Equations 3.3, 3.4, 3.5 and 3.6 were used to facilitate the mapping process. The dimensions of the frame are $M \times N$. $M = h \times \text{Macroblock\_size}$ and $N = v \times \text{Macroblock\_size}$, where $h$ and $v$ are the numbers of macroblocks and slices in
the frame, respectively. The size of the processor grid is \( mxn \), and the processors are numbered column-wise, denoted by \( r \). The set of processors is denoted by \( P_{ij} \), where \( i = 1,2,...,k,...m \) and \( j = 1,2,...,l,...n \). The \( \alpha \) term is the number of unassigned pixels on dimension \( m \), and \( \beta \) is the number of assigned pixels to each processor on that dimension. Likewise, \( \gamma \) is the number of unassigned pixels on dimension \( n \), and \( \delta \) is the number of assigned pixels in that dimension.

\[
\alpha = s \times 16 \quad (3.7)
\]
\[
\gamma = t \times 16 \quad (3.8)
\]

The equations are collapsed into smaller equations for \( \alpha \) and \( \gamma \), shown in Equations 3.7 and 3.8, respectively. In the reduced equations, \( s \) and \( t \) are positive integers, and the equations represent the number of unassigned macroblocks in each dimension.

\[
P_{ij} \text{ size} = \begin{cases} 
(\beta + 16) \times (\delta + 16) & \text{if } i \leq k, j \leq l \\
\beta \times (\delta + 16) & \text{if } i > k, j \leq l, k \leq s \\
(\beta + 16) \times \delta & \text{if } i \leq k, j > l, l \leq t \\
\beta \times \delta & \text{if } i > k, j > l 
\end{cases} \quad (3.9)
\]

\[
P_{ij} \text{ size} = \begin{cases} 
\beta \times (\delta + 16) & \text{if } r < t \\
\beta \times \delta & \text{if } r \geq t 
\end{cases} \quad (3.10)
\]

In the block method, the size of the data per processor is determined as shown in Equation 3.9. The horizontal frame division method redefines the equations for \( \alpha, \beta, \gamma \) and \( \delta \), such that \( \alpha = 0, \beta = M, \gamma = MOD(N,(m \times n \times 16)) \) and \( \delta = (N - \gamma)/(m \times n) \). Therefore, \( \gamma = t \times 16 \), since the local frame data size for each processor will be \( M \times (N/m \times n) \). This gives the new equations for the local size of data at \( P_{ij} \), as
shown in Equation 3.10.

\[
P_{ij} \text{ size} = \begin{cases} 
(\beta + 16) \times \delta & \text{if } r < s \\
\beta \times \delta & \text{if } r \geq s
\end{cases} \tag{3.11}
\]

The final frame division method is vertical. For this method, \(\alpha = MOD(M, (m \times n \times 16))\), \(\beta = (M - \alpha)/(n \times m)\), \(\gamma = 0\), and \(\delta = N\). So, the equation for \(\alpha\) then becomes \(\alpha = s \times 16\). Equation 3.11 determines the local size of data at each processor \(P_{ij}\).

**Sequence Division**

Sequence division is accomplished at the GOP level because the frames within a GOP are dependent upon each other for motion estimation.

\[
\alpha = MOD(ngops, (m \times n)) \tag{3.12}
\]

\[
\beta = (ngops - \alpha)/(m \times n) \tag{3.13}
\]

Equations 3.12 and 3.13 are the number of unassigned GOPs and the number of assigned GOPs, respectively. For these equations, \(nframes\) is the number of frames in the sequence, \(N\) is the number of frames in each GOP, \(ngops = nframes/N\) is the number of GOPs in the sequence, and \(m \times n\) denotes the processor grid.

\[
ng = r + (k \times m \times n) \begin{cases} 
if \ \alpha = 0, k = 0...\beta - 1 \\
if \ \alpha \neq 0, if \ r < \alpha, k = 0...\beta \\
if \ \alpha \neq 0, if \ r \geq \alpha, k = 0...\beta - 1
\end{cases} \tag{3.14}
\]

\[
nf = x, ..., (x + N - 1) \text{ if } ng = x \tag{3.15}
\]

Equation 3.14 gives the GOPs assigned to processor \(P_{ij}\) with rank \(r\). Then, Equation 3.15 is used to determine the number of frames assigned per processor.
Experimental Results in [22]

The experiments using the four methods described consisted of running the encoder with a video sequence of 96 frames. It was found that the sequence division method provided greater speedup over the frame division methods. The speedup of the sequence division method was further improved by pre-fetching the video sequences locally to each processor, eliminating the need for fetching them from the server workstation during the encoding process. The frame division methods performed poorly as the number of processors were scaled up due to increased I/O operations, whereas the GOP method maintained increased speedup.

Another message-passing approach to parallelizing MPEG-2 is discussed in “Performance of Software-Based MPEG-2 Video Encoder on Parallel and Distributed Systems [21].” In this approach, Shahriar M. Akramullah et al. explored numerous cluster configurations: Intel Paragon/XP-s, Intel iPSC/860 hypercube, and clusters of HP, SGI and SUN workstations. This version of a parallel encoder also uses the SPMD programming paradigm, where the frames are divided among the processors that are treated as a virtual grid. Frames are divided into equal blocks and distributed to each processor. Due to the dependencies of motion estimation on blocks in the defined search window, the entire search window is assigned to each processor, producing redundant data over the processors. Since the search windows are user-defined, the amount of redundancy will vary.

\[ X_{local} = \left[ \frac{Q}{p_h} + 2W \right] \left[ \frac{P}{p_v} + 2W \right] \] (3.16)

The local frame size per processor is given by Equation 3.16, where \( P \) and \( Q \) are the height and width of the frame, and \( p \) is the total number of processors. Then, \( p_h \).
is the number of horizontal processors, and $p_v$ is the number of vertical processors in the virtual grid. The search window is defined as $\pm W$ in both directions.

Since the data is spread across all processors, some modifications to the methods of encoding were made. As mentioned before, the entire search window for a given set of data resides on each processor, so there is no need for inter-processor communication during motion estimation. For the DCT, each processor works on one or more of the 8x8 blocks of data. The quantization and variable-length coding are dependent upon the quantization parameters determined by the rate control. For this implementation, the rate control is localized for each processor based upon the target number of bits for the current picture. Each node calculates its local expected target determined by the amount of data on which it is operating.

The experimental results in [21] were obtained using five different video sequences containing various types of motion. The dimensions were either 360x240 or 352x288, and were 50 frames each. It was shown that a nearly linear speedup was obtained on all platforms as the number of processors was scaled. The implementation was found to be useful for both real-time and non-real-time applications.

3.4.2 Shared-Memory Approach

The number of message-passing approaches to parallelizing MPEG far exceed those of shared-memory approaches. However, a shared-memory approach to encoding MPEG-1 was discussed in "Parallelizing MPEG Video Encoding using Multiprocessors [9]," by Denilson M. Barbosa et. al. The method defines two main tasks, GOP and picture. The GOP task encodes GOPs and the picture task encodes frames.

47
There is also a coordinator task to organize the GOP tasks, and a write task to output the encoded frames. A read buffer is defined, where all of the uncompressed frames reside, and the write buffer is where the compressed frames are written before being output to the encoded file by the write task. The method also uses encoding synchronization to preserve dependencies. This is accomplished by ensuring that the GOP tasks create as many picture tasks as there are frames in the GOP. Writing synchronization is also employed to prevent tasks writing to the write buffer when it is occupied. Flags are used to signal full and empty buffer conditions. The writing of the encoded data must be sequential due to the nature of the MPEG bitstream.

This approach was implemented in a multi-threaded fashion using Pthreads to alleviate the dependence of task scheduling on the underlying architecture. With Pthreads, the operating system can allocate the tasks to the available processors. To keep track of the shared data in the system, the authors implemented task tags for both GOP and picture tasks. The task tags are created by high-level tasks and used by the lower-level tasks. These tags also serve as both synchronization and communication between tasks.

Two methods of creating picture tasks were investigated: fork-join and bag of tasks. The fork-join method involves each GOP thread having its own unique set of picture threads and vector of picture tags. The limitation to this method is that once a picture task completes, the picture thread cannot encode another frame until all other tasks in the GOP have completed. Otherwise, the frame ordering in the buffers will be violated. In the bag of tasks method, the picture tasks are not associated to any one GOP task. Therefore, the GOP tasks will process the first available picture
task in the vector, eliminating the bottleneck of the previous method.

The experimentation results of these two methods showed that the fork-join method exhibited a speedup that did not scale well with the number of threads used. The bag of tasks version did exhibit nearly linear speedup as the number of threads increased up to 8. The overhead generated in creating higher numbers of threads limited speedup. It was also observed that write synchronization had negligible effect on the speedup; longer sequences performed slightly better overall.

The next chapter will discuss the pipelined parallel encoder implemented for this thesis, including an overview of the sequential encoder that was parallelized and specific implementation details of the message-passing and shared address space programming utilized for parallelization.
Chapter 4

PIPELINED PARALLEL MPEG-2 ENCODER

This chapter will discuss the specific implementation details of the pipelined parallel
MPEG-2 encoder.

4.1 Sequential Encoder

The sequential encoder obtained from the MPEG Software Simulation Group (MSSG)
[18] was originally used during the development of MPEG as a proof-of-concept test program. It is also commonly referred to as Test Model 5 in texts and other MPEG ref-
erence materials, and has been widely used as a starting-point for academic research.
The Test Model produces compliant MPEG-1 and MPEG-2 elementary bitstreams, however, it is not the most sophisticated or updated encoder implementation. The purpose of this thesis is not to design a cutting-edge parallel implementation, but to validate the concept of a parallel pipeline on the most basic MPEG implementation. Therefore, the MSSG encoder was a logical choice for this thesis.

4.1.1 Input Parameters

The encoder executable accepts as input a parameter file containing values that may be user-defined. This provides a straightforward way to vary many specific parameters to suit the video sequence being encoded. Following is a list of some parameters that may be adjusted by the user:
- **Name of Input Frames** - This tells the encoder what the prefix of the source input frame files are.

- **Name of Reconstructed Frames** - The encoder has the ability to save the reconstructed frames as files. This option lets the user decide what the file name should be, or to disable the feature by inserting a hyphen in place of a file name.

- **Name of Intra Quantization Matrix** - The intra quantization matrix may be user defined, or the default matrix specified in the MPEG standard may be used. This option accepts either the input file name or a hyphen to indicate the default should be used.

- **Name of Non-Intra Quantization Matrix** - Same as the intra quantization matrix, except this specifies the matrix to be used for Non-intra coded blocks.

- **Name of Statistics File** - The name of the file to store the statistics output is specified here. If a hyphen is specified, the statistics will print to standard terminal output.

- **File Format of Input Pictures** - This encoder can accept uncompressed frames stored in three different formats: separated Y, U and V files, combined YUV files or Portable PixMap (PPM).

- **Number of Frames** - Total number of frames in the video sequence.

- **Number of First Frame** - This is usually 0; the use of any positive integer is acceptable.
• **Time Code of First Frame** - The format for this parameter is `hh:mm:ss:ff`, where `hh` refers to the hour, `mm` to minutes, `ss` to seconds and `ff` the frame number. The default setting is `00:00:00:00`.

• **N** - This quantity specifies the number of frames in each Group of Pictures of the video sequence.

• **M** - The distance between I/P frames. `N` must be a multiple of `M`.

• **ISO/IEC 11172-2 Stream** - This is set to 1 if the sequence is to be encoded as an MPEG-1 elementary bitstream.

• **Picture Format** - If set to 0, the video sequence is encoded on a frame basis; if set to 1, it is encoded by fields and is only permitted for interlaced video.

• **Horizontal Size** - The pixel width of the frames.

• **Vertical Size** - The pixel height of the frames.

• **Frame Rate Code** - This sets the frame rate for the video sequence. The accepted codes are 1 for 23.976 frames/s, 2 for 24 frames/s, 3 for 25 frames/s, 4 for 29.97 frames/s and 5 for 30 frames/s. The setting of this value requires knowledge of the video format with which the uncompressed frames were created.

• **Bit-rate** - A floating-point value in bits/s representing the target bit-rate of the encoded stream.

• **Profile** - Sets the MPEG profile to be used. The value can be 1 for High Profile, 2 for Spatial Scalable Profile, 3 for SNR Scalable Profile, 4 for Main Profile and 5 for Simple Profile. This value is usually set to 4 for Main Profile.
• **Level** - Sets the corresponding MPEG level. Not all levels are compatible with all profiles, as discussed in Chapter 2. Accepted level codes are 4 for High Level, 6 for High 1440 Level, 8 for Main Level and 10 for Low Level. Main Level is usually chosen.

• **Progressive Sequence** - Set to 1 for progressive sequences and 0 for interlaced video.

• **Chroma Format** - Sets the sampling ratio for the chrominance components of the frames. A code of 1 represents 4:2:0, 2 is 4:2:2 and 3 is 4:4:4. This value is usually set at 1 for 4:2:0. The 4:4:4 ratio is currently not allowed in any defined profile and level combination.

• **Video Format** - This parameter goes along with the frame rate code. The supported video format codes are 1 for PAL, 2 for NTSC, 3 for SECAM, 4 for MAC and 5 for unspecified format.

• **Display Horizontal Size** - Intended width of the display screen.

• **Display Vertical Size** - Intended height of the display screen.

• **Alternate Scan** - Set to 0 if zigzag scan should be used to code the quantized DC coefficients and 1 for the alternate scan pattern.

• **Search Range** - This encompasses a set of parameters designed to define the forward and backward motion estimation search ranges for I, P and B pictures.

There are many other settable parameters that are explained fully in the MSSG encoder documentation [18].
4.1.2 Code Structure

The code for the MSSG encoder is written in standard C and may be compiled using gcc with the optimization parameter of -O2. The main function calls an initialization function which reads in the input parameters from the specified file and checks their validity. After the initialization, the function performing the actual encoding process is called. Within that function, the code is structured in a loop that processes all frames in the video sequentially. The ensuing process is categorized as follows:

- **Setup** - This portion determines what type of frame the current frame is; based upon the frame number, the number of frames in the current GOP $N$ and the distance between the I/P frames $M$. This ensures the reference frames from the previous frame, as well as the previous frame, are used as reference frames for motion estimation of the current picture. (This is discussed in more detail later.) Lastly, the actual frame data is read in from the uncompressed video files.

- **Motion Estimation** - The motion estimation function accepts the current frame, and all of its reference frames, as input, along with the data structure containing the macroblock information for the current picture. In this step, each macroblock of the current picture is assigned a coding type based upon the results of the full-search block-matching against the reference frames, unless it is an I frame. In the case of I frames, all macroblocks are intra-coded using JPEG (Joint Photographic Experts Group) compression. Also decided is the necessity of prediction use and whether to code the prediction error of each macroblock. Forward or backward prediction is decided for each macroblock in P and B frames, with the motion vectors being calculated accordingly.
• **Prediction** - This step takes the results of the motion estimation and, for each macroblock, will perform the specified prediction. Prediction is accomplished by subtracting the current macroblock from the reference macroblock and passing on the result for compression.

• **DCT Type Estimation** - Whether to use frame or field based DCT is decided for each macroblock in the picture. This is based on interfield correlation for field pictures. For frame pictures, the user-defined values for each frame type determine the use of frame or field-based DCT.

• **DCT** - Discrete Cosine Transform is performed on each block of every macroblock in the current picture. The documentation for the encoder states that the Chen-Wang fast DCT algorithm was used. The resulting DCT coefficients are stored in a data structure containing all the information for all blocks of a picture.

• **Quantization and Variable-Length Coding** - At this stage, all encoded data is created and output. The DCT coefficients are quantized and coded using the same method as described in Chapter 2. For compliance with the standard, the appropriate header information is printed to the encoded file before the encoded data is output.

• **Inverse Quantization** - This step reverses the effects of quantization and variable-length coding. The algorithm used is defined in the MPEG-2 standard, which essentially reverses what has been done in the previous step.

• **Inverse DCT** - The IDCT is also defined in the standard, and conforms to IEEE Standard 1180-1990 for 8x8 Inverse Discrete Cosine Transforms [2]. After
completion of this step, the reconstructed frames are saved for use as reference frames.

- **Statistics Output** - This is a specific feature of this encoder. The statistics output gives in-depth detail regarding each frame such as the quantization parameters, coding and prediction types used for each macroblock. All information produced from this stage is useful in determining any problems or to verify the bitstream produced was the one expected.

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned char *neworg[3]</td>
<td>Contains the uncompressed frame data for the current frame. The first element is the luminance information and the second two contain the chrominance information.</td>
</tr>
<tr>
<td>unsigned char *newref[3]</td>
<td>Same as neworg, except that the reconstructed frame data is stored here.</td>
</tr>
<tr>
<td>unsigned char *oldorgframe[3]</td>
<td>Contains the original frame data for the reference frame used in forward prediction.</td>
</tr>
<tr>
<td>unsigned char *oldrefframe[3]</td>
<td>Same as oldorgframe, except it contains the reconstructed frame instead of the original.</td>
</tr>
<tr>
<td>unsigned char *neworgframe[3]</td>
<td>Contains the original frame data for the reference frame used in backward prediction.</td>
</tr>
<tr>
<td>unsigned char *newrefframe[3]</td>
<td>Same as neworgframe, except it contains the reconstructed frame data.</td>
</tr>
<tr>
<td>unsigned char *predframe[3]</td>
<td>Stores the predicted frame data after the prediction stage.</td>
</tr>
<tr>
<td>short (*blocks)[64]</td>
<td>Stores the 64 DCT coefficients for each block in the picture.</td>
</tr>
</tbody>
</table>

Table 4.1: Frame Data Structures

**Data Structures**

All information for the frames is stored in data structures. The most important data structures are those containing the original and reconstructed frame data, the
predicted frame data, the macroblock information for the current picture and the data for each block in the picture. Table 4.1 describes the structures containing the frame data and the block information and Table 4.2 gives a description of the macroblock information structure. The space for each data structure is allocated dynamically at runtime by using the incoming frames sizes to calculate the amount of memory allocation needed.

### 4.2 Parallelization of Sequential Encoder

The following section describes the sequential encoder parallelization process.

#### 4.2.1 Message-Passing Parallelism using MPI

The first step in parallelizing the sequential encoder was to determine the task size. Initially, the task size was chosen to be the frame level, where each frame would be sent to the encoder sequentially, and the parallelism would occur in the various
encoder stages. However, this attempt did not perform well due to the excessive communication needed between the steps. The data structures described previously are very large and it is necessary for all data to be available for each stage, therefore, this approach was abandoned. From previous work [22], it is noted that parallelism is possible at the sequence level, which produces the coarsest granularity. The video requires partitioning by GOPs as opposed to frames.

\[ A = nframes - (N - (M - 1)) \]  \hspace{1cm} (4.1)
\[ B = A/N \]  \hspace{1cm} (4.2)
\[ F = \begin{cases} B + 1 & \text{if } A \mod B = 0 \\ B + 2 & \text{otherwise} \end{cases} \]  \hspace{1cm} (4.3)

Equations 4.1, 4.2 and 4.3 were derived for partitioning, making the implementation scalable. \( A \) is the number of frames in the sequence minus the number of frames in the first GOP because there are less frames in the first GOP than in all others (due to frame reordering). \( B \) calculates the number of GOPs aside from the first of the picture. \( F \) calculates the total number of GOPs in the sequence. If the number of frames in the sequence, minus the number of frames in the first GOP, is evenly divisible by the number of GOPs minus the first, then there are \( B + 1 \) GOPs in the entire sequence. Otherwise, there are \( B + 2 \) GOPs to account for the leftover frames.

\[ gpp = F/nprocs \]  \hspace{1cm} (4.4)

To determine how many GOPs per processor there should be, the following decisions are made: If \( F \) modulo the number of processors \( nprocs \) is 0, then Equation 4.4 holds. Otherwise, in addition to Equation 4.4, for \( i = 0...F/nprocs \), if the rank
of the current processor is equal to \( i \), increment \( gpp \). This is done to balance the loads across the processors in the event the number of GOPs does not divide evenly by the number of processors. This method ensures there is never more than 1 GOP’s difference between any of the processors.

\[
gpp_{\text{offset}} = 0; \quad \text{for } i = 0 \ldots \text{rank}, \quad \text{if } i \leq (F \mod nprocs) \text{ then } gpp_{\text{offset}} += (F/nprocs) + 1; \quad \text{else } gpp_{\text{offset}} += (F \mod nprocs).
\]

Therefore, this will calculate the number of GOPs assigned to all processors with lower ranks than the current processor.

Next, the frame numbers of the GOP boundaries were calculated for each processor based upon its rank, the number of GOPs assigned to it and the total number of frames in each GOP. Care was taken to ensure the processor with the highest rank did not exceed the highest frame number.

Figure 4.1 illustrates how the GOPs from the video sequence are logically assigned to the processors. To determine the frame numbers of the GOP boundaries, each processor must know how many GOPs come before its assigned GOPs in the video sequence. The GOP offset \( gpp_{\text{offset}} \) is calculated as follows: \( gpp_{\text{offset}} = 0; \) for \( i = 0 \ldots \text{rank} \), if \( i \leq (F \mod nprocs) \) then \( gpp_{\text{offset}} += (F/nprocs) + 1; \) else \( gpp_{\text{offset}} += (F \mod nprocs) \). Therefore, this will calculate the number of GOPs assigned to all processors with lower ranks than the current processor.
Minimal MPI calls were made to achieve message-passing parallelism. The only MPI functions called were *MPI_Init()* , *MPI_Comm_rank()* , *MPI_Comm_size()* and *MPI_Finalize()* . However, there was implicit communication between the root node and the worker nodes, because the root node stored all uncompressed frame data. MPI enabled each processor to access the data on the root without any explicit function calls, which minimized overhead.

### 4.2.2 Multi-Threaded Pipeline using Pthreads

To increase the frame throughput on each node, a multi-threaded pipeline was implemented on each. However, because MPI only requires one source file, only one code version was necessary to implement the pipeline. MPI allowed the specified processors to execute the pipelined code. The first step in creating the pipeline was to choose the stages. In this case, the easiest way to partition the workload was to make each step of the encoding process into a stage. The steps of the encoding process were explained in a previous section of this chapter. Figure 4.2 shows the resulting pipeline.

```
<table>
<thead>
<tr>
<th>Setup</th>
<th>Motion Estimation</th>
<th>Prediction</th>
<th>DCT Type Estimation</th>
<th>DCT Quantization</th>
<th>IDCT</th>
<th>Inverse Quantization</th>
<th>Statistics</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>
```

*Figure 4.2: MPEG-2 Pipeline Diagram*

The two largest challenges regarding pipeline implementation were the data structure management and pipeline scheduling. As a result of the threads sharing the same memory space, and therefore data structures, it was necessary to create indexed data structures. In effect, each stage of the pipeline was able to maintain its own state. All
global variables and data structures to be modified in multiple stages were increased by one dimension. For example, the array of three unsigned character pointers that contained the original frame data, became a two-dimensional array of 9 rows. Each row containing three unsigned character pointers. Any local static variables in low-level functions were also indexed in the same manner.

Scheduling the pipeline involved designing a state machine to handle the data flow through the pipeline. Figure 4.3 is a diagram of the resulting state machine.

![Figure 4.3: State Machine for Pipeline Scheduling](image)

A major consideration was the dependencies of different stages of the pipeline. The most prevalent data dependency was between the 1st and 8th stages of the pipeline,
because the setup stage requires the reconstructed frames from the previous frame in the GOP. Those frames are produced at the end of the 8th stage. To ensure this dependency was not violated, the pipeline was filled with unrelated frames. For example, the first frame of the first GOP would enter the pipeline, followed by the first frame of the second GOP, etc. To make sure that the data for each stage of the pipeline was pushed through, a function was written to specifically shift all indexed data by one index. Consequently, the data in index 0 was pushed out, effectively exiting the pipeline, while the data in index 1 is pushed into index 0, etc. New data is always written to index 8.

An indexed structure containing the frame number, GOP number and offset of the frame from the first frame in the GOP was created to keep track of which frames were in which stages of the pipeline. This structure is initialized before execution of the state machine, so the state machine is built around that knowledge.

In the first state, the frame in index 0 is sent into the pipeline. In the second state, the second frame enters and the frame in index 0 is sent into the second stage. This continues until the frame in index 8 enters the pipeline and the state machine is in state 8. The frame entering in stage 8 is always the next frame in the GOP to the frame in stage 0. The state machine will continue to be in state 8, while processing all stages of the pipeline and after each round the data in the indexed data structures will be shifted by the method explained above. In the case of static data in low-level functions, the data from stage 0 is copied into stage 8 to mimic the sequential version. When the last frame to be processed enters the first stage of the pipeline, the next state is set to state 9. The remaining states continue to process the remaining frames
in the pipeline until the last frame exits.

The threads are created and joined within each state of the state machine. Figure 4.4 gives an example of the indexed data on which each thread operates, within each stage of the pipeline. Given that the `pthread_create()` routine only accepts one argument to pass to the specified function, structures were created for each stage to assure the appropriate arguments could be passed. These structures are initialized before each call to the create routine. After all the create routines have been executed, the main thread will issue a `pthread_join()` call for each thread created. This is done to guarantee that the main thread does not proceed to the next state of the state machine without the data from the child threads.

<table>
<thead>
<tr>
<th>States</th>
<th>S0</th>
<th>S1</th>
<th>S2</th>
<th>S3</th>
<th>S4</th>
<th>S5</th>
<th>S6</th>
<th>S7</th>
<th>S8</th>
<th>S9</th>
<th>S10</th>
<th>S11</th>
<th>S12</th>
<th>S13</th>
<th>S14</th>
<th>S15</th>
<th>S16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setup</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ME</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pred</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DCTE</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DCT</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Quant</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IQQuant</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IDCT</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stats</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.4: Data Flow through Pipeline

Ideally, the best throughput occurs when the pipeline is filled. To achieve a full pipeline, 8 GOPs must be assigned to each processor. However, this is not always realistic. Therefore, there are times when the pipeline stalls to wait for frames in the
pipeline to be processed completely before allowing the next frame in that GOP to enter. To keep track of which GOP the stall belongs to, the stalls themselves have GOP and offset numbers corresponding to the “missing” GOPs. This enables the scheduling algorithms to know when to place more stalls into the pipeline and when to allow additional frames to enter (if there are any). As an example, consider a processor that has 4 GOPs assigned to it, where the GOP length is 2 frames. Figure 4.5 shows what happens, logically, in the pipeline under these conditions. Within the code, if the frame at the index at which a thread requests access is a stall, the thread will not be created. This may seem like a stall, but instead of processing a frame, nothing was done with that stage. The throughput was decreased as a result of the stall, which is the same effect as if the thread was created and did no work.
Sometimes, there may be more than 8 GOPs assigned per processor. To handle this case, 8 GOPs are processed at a time. As the last frame of the 8th GOP exits stage 1, the next GOP is immediately inserted in the pipeline. Execution continues until there are no more frames.

As mentioned previously, a data dependency exists between the 1st and 8th stages of the pipeline, with respect to the reconstructed frames. Although GOPs can be coded as closed GOPs, where they do not use any predicted data from previous GOPs, this method produced an undesired result in the video output. For that reason, the problem was solved by having each I frame read with the original frame data for the required reference frame from the previous GOP set. This was done by accessing the file directly on the root, and is similar to pre-fetching the frames for each processor, which was done in [22].

The discussion to this point has been about the MPEG-2 encoder. However, the decoder must also be mentioned. Within the MPEG-2 encoder, decoder operations also take place; namely the inverse quantization and inverse DCT are two steps performed in both. Rate control exists for incoming frames in the decoder as opposed to outgoing frames in the encoder. Therefore, it is conceivable that a similar approach could be applied to the decoder. However, MPEG-2 decoders are primarily built in hardware due to real-time processing requirements.

In the following chapter, the results of simulations using the pipelined parallel encoder will be presented and examined. They will also be compared to results from
previous work.
Chapter 5

RESULTS

This chapter discusses the experimental results of this thesis. The pipelined parallel encoder completed for this effort produced error-free bitstreams when encoding progressive sequences. The validity of the encoded bitstreams was tested by using the mpeg2play command-line MPEG-2 bitstream player with debug output that is freely available from the MPEG Software Simulation Group [18]. The correctness of the bitstream was also ascertained by visual inspection alongside the original encoded sequence using the Elecard Video Player [1].

5.1 Setup

Several simulations were run on a cluster-based message-passing architecture. The cluster consists of 15 Linux workstations running Redhat Linux release 7.2. The nodes are in a completely connected network topology that uses a Cisco 2948G Ethernet switch with a 24 Gigabit switch fabric. Each node uses an Intel® Celeron™ CPU with a 1.2 Megahertz clock speed.

The MPI programming environment installed on the cluster is MPICH 1.2.4, developed by Argonne National Laboratories [3]. Pthreads were used within MPICH by re-compiling MPICH with the listener signal option set to SIGUNUSED, since both MPICH and Pthreads use the same Linux system signals. Pthreads exists as an integrated library in most newer versions of Linux. On the cluster, the gcc standard C
compiler libraries are version 2.96. The parallel code was compiled using mpicc MPI C compiler with compiler optimization flag -O2 and flag -D_REENTRANT (which is necessary for Pthreads).

5.2 Experimental Results and Analysis

Several sets of results were generated using the parallel encoder. The goal of this experimentation was to compare the speedup achieved using parallel encoder with the ideal linear speedup. The video sequences used to test the encoder, as well as generate the simulation results, were obtained from the TEKTRONIX MPEG Elementary Streams website [23]: which are recognized sequences used to test MPEG video compression. Those two chosen for experimentation were the BBC3 test sequence and the table tennis test sequence. Each video is a progressive sequence with a dimension of 352x288 pixels and containing 375 frames.

Simulation Results

Three simulations were run for each video sequence. In the first simulation, frames in a GOP $N$ was set to twelve (12) frames with the distance between anchor frames $M$ at three (3), setting the number of GOPs in the sequence at thirty-two (32). The 375 frames were encoded using processor numbers ranging from 2 to 14. A sequential encoder was used to obtain the encoding time for one processor. The sequence was encoded 5 times, at each number of processors, to obtain an average encoding time. For the second simulation, $N$ was set to 8 and $M$ to 2, yielding 47 GOPs in the sequence. This was done to vary the amount of work per task, as well as the total number of GOPs assigned to each processor. All 375 frames were encoded for this second simulation. The third simulation was a variant of the first simulation, in that
Figure 5.1: BBC3 Simulation Results

$N$ and $M$ were the same, however the number of frames were reduced to 315. This set the number of GOPs in the sequence to be 27. It was necessary to vary the total number of GOPs per processor without varying the amount of work completed by each task to see what the impact on speedup was. The resulting speedups from these simulations are shown in Figures 5.1 and 5.2. Table 5.1 shows the average execution times for the three simulations with both video sequences. Note that all results include I/O operations.
Analysis

It has been shown that the pipelined parallel encoder does not exhibit linear speedup when using these resources to run the simulations. Regardless, the results utilizing the parallel encoder always exceeded those of the sequential encoder. Plateaus in the speedup are caused when the numbers of GOPs are not evenly divisible by the number of processors causing the loads on the processors to be slightly imbalanced. Generally, speedup increases as the number of processors increases. However, there should never
<table>
<thead>
<tr>
<th>Num.</th>
<th>BBC3 Video Sequence (s)</th>
<th>Table Tennis Video Sequence (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pros</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>N=12, M=3, 375 frames</td>
<td>N=12, M=3, 375 frames</td>
</tr>
<tr>
<td>1</td>
<td>75.920</td>
<td>61.385</td>
</tr>
<tr>
<td>2</td>
<td>59.722</td>
<td>55.481</td>
</tr>
<tr>
<td>3</td>
<td>46.442</td>
<td>44.241</td>
</tr>
<tr>
<td>4</td>
<td>29.607</td>
<td>28.848</td>
</tr>
<tr>
<td>5</td>
<td>26.932</td>
<td>26.627</td>
</tr>
<tr>
<td>6</td>
<td>24.259</td>
<td>24.536</td>
</tr>
<tr>
<td>7</td>
<td>21.475</td>
<td>21.817</td>
</tr>
<tr>
<td>8</td>
<td>18.734</td>
<td>19.332</td>
</tr>
<tr>
<td>9</td>
<td>18.849</td>
<td>19.294</td>
</tr>
<tr>
<td>10</td>
<td>18.916</td>
<td>19.278</td>
</tr>
<tr>
<td>11</td>
<td>16.080</td>
<td>16.514</td>
</tr>
<tr>
<td>12</td>
<td>16.128</td>
<td>16.373</td>
</tr>
<tr>
<td>13</td>
<td>16.147</td>
<td>16.489</td>
</tr>
<tr>
<td>14</td>
<td>16.299</td>
<td>16.816</td>
</tr>
</tbody>
</table>

Table 5.1: Execution Times

be more than a 1 GOP load imbalance between any set of processors, so the effects are not as damaging as with a less efficient mapping mechanism. The differences in the three simulations can partly be attributed to load balancing, as well. It is shown that the simulation with the most GOPs produces the best speedup. In the second simulation, the number of GOPs assigned to each processor fills the pipeline more
efficiently than the numbers assigned to those in the others. For example, when running simulation 1 on 14 processors, the number of GOPs is 32. Since 32 is not evenly divisible by 14, there are 4 processors working on 3 GOPs each and 10 processors working on 2 GOPs each. Due to the sequential dependence of frames inside each GOP, the pipeline is less than half full at all times. For the second simulation, the number of GOPs is 47, which is also not evenly divisible by 14, leaving 5 processors each working on 4 GOPs and 9 processors each working on 3 GOPs. Subsequently, the pipelines of 5 processors are at least half full, as opposed to the first simulation where no processors had pipelines that full. A higher throughput, and therefore speedup, results when there are more frames in the pipeline. Given this, it is shown that the selection of processors used to achieve maximum speedup is highly dependent upon the number of GOPs in the video sequence.

The most crucial component effecting lack of speedup is the multi-threaded pipeline running on a uniprocessor. The overhead of the context switching between threads diminishes speedup. This overhead is especially pronounced when there are enough GOPs to fill the entire 9-stage pipeline, because every stage of the encoder will be executing within a separate thread simultaneously. Due to this factor, it is clear that the resources needed to run this parallel encoder is a cluster of SMP nodes, meaning a combination of shared-memory and message-passing architectures. The speedup of processing parts of a video sequence in parallel would be enhanced if the threads running in each node were allowed to run on separate processors concurrently. It can be speculated that, even with four-processor SMP nodes, the speedup would dramatically increase. The overhead of running 9 threads simultaneously on one processor would dramatically decrease if 9 threads were running on four processors. Theoret-
ically, the amount of overhead would be distributed evenly over the four processors in the SMP node, so the speedup should increase by a factor of 4 per node. Coupled with virtually ideal speedup generated by running on a cluster with minimal communication between nodes, the speedup could approach, if not exceed, the expected linear speedup.

Comparison to Previous Approaches

![Graph showing speedup vs. number of processors for sequence division method](image)

Figure 5.3: Speedup vs. Number of Processors for Sequence Division Method [22]

The closest previous approach to the methodology of this thesis was the sequence-division message-passing approach [22] discussed in the parallel processing background chapter. Due to the limitation of resources, the results of this thesis did not match the best performance of the sequence-division method. It did resemble the speedup obtained in [22] when the I/O operations were included in the execution time. In Figure 5.3, M4 is the sequence division method in [22] and M4-local is sequence division with pre-fetching of frames before encoding. M4-I/O and M4-local-I/O include
the I/O operations in the execution time. As can be seen, the M4-I/O has a similar trend to the results obtained for this thesis. At 4 processors, the speedup of M4-I/O is approximately 2, whereas the best speedup in the BBC3 sequence is slightly below 2. For 8 processors, M4-I/O has a speedup of slightly below 5, as does the BBC3 sequence. At 14 processors, M4-I/O has a speedup of approximately 5 and the BBC3 sequence has a speedup of 6 for the second test, out-performing M4-I/O. It can be speculated that at higher numbers of processors, the pipelined parallel encoder would continue to exhibit increased speedup, provided the number of GOPs per processor is not calculated below 1. The maximum number of processors that can be used for any given video sequence is determined by the total number of GOPs.

On the correct set of resources, it is likely the pipelined parallel encoder would rival, or exceed, those results. For example, if the pipeline were run on an SMP node with 2 processors, the speedup at any number of processors is expected to increase by a factor of two. Therefore, the best speedup of 6 at 14 nodes is expected to increase to 12. Since the ideal linear speedup for 14 processors is 14, the speedup would be nearly ideal, and would exceed the performance of M4 in [22]. It is not practical to compare sequence-division message-passing results to the frame-based approaches, as it has been shown in [22] that GOP level granularity provides maximum speedup for parallelism of MPEG.
Chapter 6

CONCLUSIONS

6.1 Accomplishments

This thesis presents a novel approach to parallelizing an MPEG-2 encoder using message-passing with a multi-threaded pipeline. The objective of implementing a pipelined parallel encoder for MPEG-2 video bitstreams was successful. While the results produced did not exhibit ideal linear speedup, they always exceeded the execution time of the sequential encoder. It was also shown that the pipelined parallel encoder did exceed similar results obtained in previous work [22] at higher numbers of processors. The parallel encoder would likely produce better speedup results if run on a cluster of SMP nodes.

6.2 Limitations

The major limitation of this work is that speedup is highly dependent upon the number of frames in a GOP, as well as the number of GOPs in the video sequence. Also, the sequential encoder used to create this implementation was not the most efficient implementation. The pipelined parallel encoder can only process progressive sequences at this time. Another limitation is the possibility of an inefficiently coded pipeline. In addition, the compilation of the encoder was not optimized for the platform on which it was run and could have hindered the results.
6.3 Future Work

There is much more work that can be done with the parallel encoder. The following is a non-exhaustive list:

- Run the pipelined parallel encoder on a cluster of SMP nodes to determine the degree of speedup improvement.

- Optimize the encoder for the platform on which it is run through more efficient compilation.

- Determine the efficiency of the pipeline.

- Investigate other pipeline scheduling algorithms that would preserve the data dependency of the current pipeline.

- Investigate ways to restructure the pipeline to improve performance. This includes re-structuring the individual stages of the pipeline. For example, split the DCT stage into two stages, since it is computationally intensive. Also, combine the DCT type estimation stage with the first part of the DCT stage, since it entails light computation.

- Add support for interlaced frames.

- Parallelize and pipeline an up-to-date sequential encoder that is known to be efficient.

- Determine if this work can realistically be applied to MPEG-4. If so, attempt an implementation.
Appendix A

APPENDICES

The code can be viewed on the CD-ROM included with this thesis.
BIBLIOGRAPHY


[23] TEKTRONIX. *MPEG Elementary Streams.*
