# PARALLEL SOFTWARE IMPLEMENTATION OF PPM (METHOD C) DATA COMPRESSION SCHEME


**Submitted by:**

**Nabeeg Mukhtar**

**Supervised by:**

**Assoc Prof Dr Athar Mahboob**

**THESIS**

**Submitted to:**


**Department of Electronics and Power Engineering,**

**Pakistan Navy Engineering College Karachi,**

**National University of Sciences and Technology, Islamabad**


**In fulfillment of requirements for the award of the degree of**

**MASTER OF SCIENCE IN ELECTRICAL ENGINEERING**

**With Specialization in Communication**

**March 2012**

# ACKNOWLEDGMENTS

First of all, with a deep and profound gratitude, I am grateful to almighty ALLAH (ever Merciful and Beneficent) for his blessing upon me and giving me wisdom, knowledge and understanding without which I would not have been able to successfully complete this thesis work.

It is a great honor for me that after an extensive coursework related to the subject; I have completed this research work. I would like to express my sincere gratitude and acknowledge the guidance of all those who were helpful to me in the course of this work.

My special thanks to Dr Athar Mahboob, Associate Professor Department of Electronic and Power Engineering at PN Engineering College, who is my Supervisor for this thesis work. The able guidance and support, at every step of the research work played a vital role in accomplishment of this work.

In addition to above, I am also thankful to the guidance committee comprising of following faculty members who professionally led me to achieve my target:

- Professor Dr Pervez Akhter
- Associate Professor Dr Arshad Aziz
- Associate Professor Dr. Sameer Qazi

At the end I would pay my sincere gratitude to my parents whose wish I begin this post graduate degree and with whom I am looking forward to share this Degree.

# Abstract


The "Prediction by Partial Matching" (PPM) data compression scheme is a state of the art compression scheme developed by Cleary and Witten. This scheme is capable of achieving very high compression rates proven to be as low as 2.2 bits/character for English test.

The nature of the PPM scheme is such that it is based on two stages heavily dependent on each. The first is the context modeling whereas the second being the encoding stage. Both these stages involve excessive looping, searching and data sharing. These peculiar natures makes PPM scheme perform quite slow at processing time as compared to some others well know compression schemes.

In this work different parallel implementations of a variant of PPM (PPMC) is shown so as to achieve a much faster processing time. The results drawn show that parallelization significantly assist reducing processing time for PPMC to as low as one fourth of the sequential processing time. This work also introduces a few new parallel architectures that can be considered suitable for any process which may require parallel speed up.

# Table of Contents


Abstract .......... 2

Table of Contents .......... 3

List of Figures .......... 5

Chapter 1: Introduction .......... 6

1.1 Introduction .......... 6

1.2 Thesis Scope .......... 6

1.3 Chapters Organization .......... 7

Chapter 2: Data Compression .......... 8

2.1 Data Compression .......... 8

2.2 Classification of Data Compression Schemes .......... 9

2.3 Compression Ratio .......... 11

2.4 Data Compression Corpora .......... 12

2.5 Data Compression Today .......... 14

Chapter 3: PPM Data Compression Scheme .......... 16

3.1 Arithmetic Coding .......... 16

3.2 Prediction by Partial Matching (PPM) .......... 20

3.2.1 PPM Variants .......... 22

3.2.2 PPM method C (PPMC) .......... 23

Chapter 4: Parallel Computing .......... 24

4.1 Parallel Computing .......... 24

4.2 Parallel Architectures .......... 25

4.2.1 Parallel Lines .......... 25

4.2.2 Sequential Pipes .......... 26

4.2.3 Parallel Pipes .......... 28

4.2.4 Hybrid Tree .......... 29

Chapter 5: Parallel Data Compression .......... 30

5.1 Parallel Implementation of Compression Methods .......... 31

5.2 Parallel PPM (method C) .......... 32

Chapter 6: Parallel Software Implementation of PPM (method C) Data Compression Scheme .......... 33

6.1 Software Environment ........ 33
6.1.1 C# Programming Language ........ 34
6.1.2 .Net Framework ........ 34
6.1.3 Relation between C# and .Net Framework ........ 35
6.1.4 Task Parallel Library (TPL) ........ 35
6.1.5 Visual Studio ........ 35
6.1.6 Windows 7 ........ 36
6.2 Hardware Environment ........ 36
6.2.1 Intel Core i5 ........ 37
6.3 PPMC Implementation ........ 37
6.3.1 Context Orders ........ 38
6.3.2 Pseudo Code ........ 40
6.3.3 Sequential PPMC ........ 43
6.3.4 Parallel Lines PPMC ........ 43
6.3.5 Sequential Pipes PPMC ........ 46
6.3.6 Parallel Pipes PPMC ........ 48
6.3.7 Hybrid Tree PPMC ........ 50
Chapter 7: Test Analysis and Results ........ 52
Chapter 8: Conclusion ........ 62
8.1 Concluding Remarks ........ 62
8.2 Future Works ........ 63
Appendix A ........ 64
Information theory behind compression ........ 64
A.1 Entropy ........ 64
A.2 Redundancy ........ 66
A.3 Relation between entropy and redundancy ........ 66
Appendix B ........ 67
Prediction and Probability theory concepts ........ 67
B.1 Prediction ........ 67
B.2 Probability ........ 67
B.3 Number line (probability line) concept in probability ........ 68
References ........ 70

## List of Figures


Figure 1: Comparison between a compressed and uncompressed image file. ............................ 8
Figure 2: Data Compression types ............................................................................. 11
Figure 3: Arithmetic Coding ..................................................................................... 17
Figure 4: Arithmetic Coding ..................................................................................... 17
Figure 5: Arithmetic Coding ..................................................................................... 18
Figure 6: Parallel Lines Model ................................................................................... 26
Figure 7: Sequential Pipes Model .............................................................................. 27
Figure 8: Parallel Pipes Model ................................................................................... 28
Figure 9: Hybrid Tree Model ..................................................................................... 29
Figure 10: Parallel Lines PPMC .................................................................................. 44
Figure 11: Sequential Pipes PPMC ............................................................................. 46
Figure 12: Parallel Pipes PPMC ................................................................................. 48
Figure 13: Hybrid Tree PPMC .................................................................................... 50

# Chapter 1: Introduction

## 1.1 Introduction

One of the fastest penetrating instruments in our lives is computers. Computers process data to get information and exchange information. With the passage of time, the amount of data which is processed by computers has increased significantly causing a need to realize the importance of data compression.

Data compression essentially revolves around the problem of efficiently compressing computer's data. The modern field of software based data compression came into existence in the late 1970s. The field of data compression addresses data compression schemes and there use in compressing data. All data compression schemes that efficiently compress data add length to processing time. This lengthy processing time by these schemes is because on the fact that such schemes reprocess data multiple times to produce efficient compressed output. One such scheme is "Prediction by Partial Matching" (PPM). The PPM data compression scheme is an entropy coder capable of coding symbols close to their entropy.

The work here makes use of parallel computing to address the problem of processing time speed up for the PPM (method C) data compression scheme.

## 1.2 Thesis Scope

A lot of research has been done on the PPM data compression schemes. These works address efficient implementation, optimization and processing time speed up for the sequential implementation of PPM schemes. No work on parallel implementation of any PPM schemes has been done till date. This work is the first in line to address the use of parallel computing for a PPM scheme.

The work here is mainly concerned with the parallelization of PPM (method C) data compression scheme so as to achieve a shorter and better processing time. It does not involve

any efficiently implementation or optimization of the stated data compression scheme. This work makes use of software environment to implement, test and analyze the parallelization of PPM (method C). The software environment is composed on C# programming language and the .Net framework. All the functionalities used are part of the core C# and .Net framework. The implementation does not depend upon any external or third party ISVs (Independent Software Vendors) libraries. The hardware underneath which is called by the software environment is a quad core (Intel core i5) system.

Parallel software implementation of PPM (method C) data compression scheme analyses four different parallel implementations of PPMC and compares them with the sequential implementation of PPMC to observe parallel speed up. It also introduces two new parallel architectures and a name based identification for all parallel architectures.

## 1.3 Chapters Organization

Chapters in this thesis report are organized as follows:

- Chapter2 begins with introduction to data compression field.
- Chapter3 is based on PPM data scheme and related knowledge.
- Chapter4 introduces parallel computing.
- Chapter5 discusses parallelism in data compression.
- Chapter6 introduces the software and hardware environments used for parallelization of PPMC.
- Chapter7 completes test and results.
- Chaptere8 adds concluding remarks and suggested future work for this thesis.

Appendix A is on information theory concepts related to data compression theory. It may be handy if require some prerequisite knowledge on related concepts of chapter2.

Appendix B describes some probability concepts related to chapter3.

# Chapter 2: Data Compression

## 2.1 Data Compression

Data compression is a process of removing redundancies or the extra data from data. This process revolves around processing of raw data in such a manner so that it loses its actual size when compressed and regains it when decompressed.

The field of data compression is concerned with the finding, study and development of novel and efficient data compression techniques. The modern software based data compression field is quite new as it got recognized around 30 years back. Over this short period, a huge amount of research and developments have been accomplished in this field. This field has penetrated computer's data to such an extent that a number of its algorithms have become a standard data file types such as mp3, pdf, jpg and many more. Nowadays a lot of computer processes speak to each another in terms of compressed data.

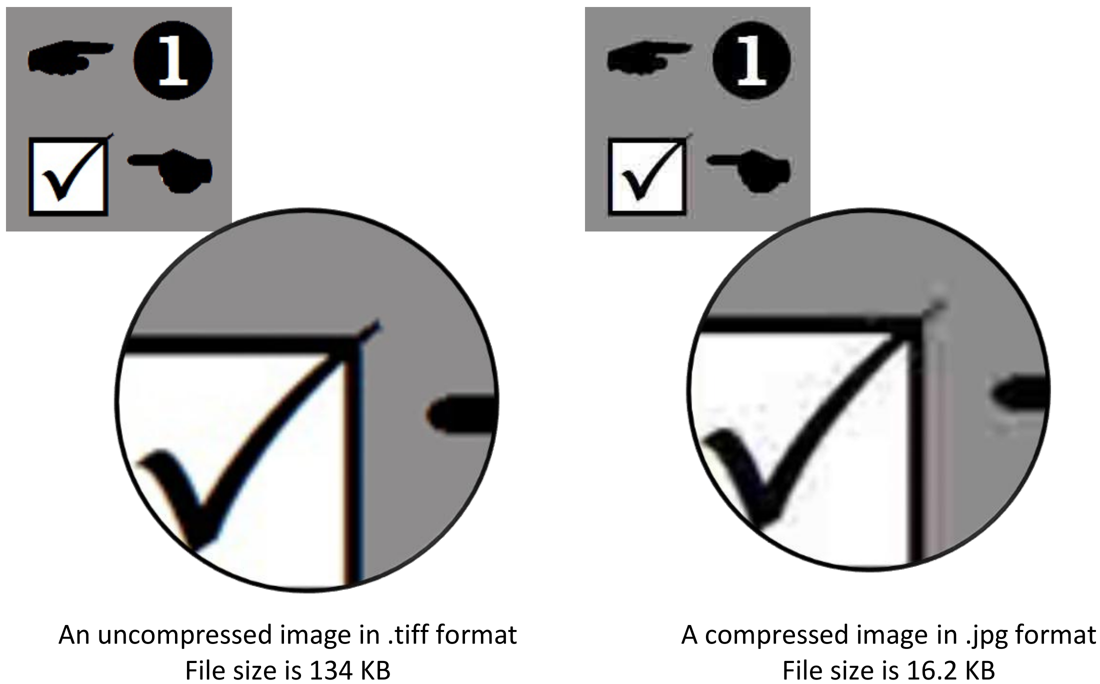



**Figure 1: Comparison between a compressed and uncompressed image file.**

Figure 1 (above) shows two images, one uncompressed and the other as a compressed file. Notice here that the file size has reduced significantly in the compressed version.

During the process of compression, a file (data stream) is subjected to a compression algorithm. This algorithm checks for repetitive patterns or redundancies that can be safely removed from this stream and later recovered when reprocessed (backward operation). This reprocessing of data is called decompression.

The whole process of compressing and decompressing takes time; usually compression algorithms which produce high quality compression are restricted to more time consuming computing. This leads to classification and categorizing of compression algorithms.

## 2.2 Classification of Data Compression Schemes

Data compression schemes, algorithms or techniques can be classified and group together in a number of way. Most common classification and grouping is based on similar characteristics that are observed by these schemes. Some of the most common and known classifications are as follows:

The following is based on how correctly a compression scheme performs:

- ***Lossless and lossy types***

  The most common classification is to group compression methods as either lossless or lossy. Lossy types give better compression at the price of losing some information. When their output compressed stream is decompressed, the result is not identical to the original data stream [1]. If the loss of data is small the difference may be negligible. Lossy compression methods are commonly used to compress images, video, or audio. However in executable, source, text and other similar files no loss can be tolerated as it will alter the content of such files, here the use of lossless methods are best suited. Lossless methods return complete file on decompression without altering or losing any of the original file contents.

Examples include:

Lossy type: JPEG, mp3, etc.

Lossless type: PPM, LZW, etc.

Another classification is based on how a compression scheme processes data, these includes:

- ***Statistical methods***

  A statistical method populates the statistical records of all unique symbols in a data stream and later on these assign compression bits.

  Examples include: Huffman coding, Arithmetic coding, PPM, etc.

- ***Dictionary methods***

  Dictionary based methods makes a storage (dictionary) marking the occurrence of unique symbols during compression, any symbol marked previously is not restored.

  Examples include: LZW, LZ77, LZMA, GIF images, etc.

Another well-known classification of compression techniques is based on files types, these includes:

- ***Text compression methods***

  These are usually associated with compressing of text files.

  Examples include: Shannon-Fano coding, Huffman coding, etc.

- ***Image compression methods***

  These methods are for image files compression.

  Examples include: JPEG, JPEG 2000, PNG, GIF, WEBP, etc.

- ***Audio compression methods***

  Associated with audio files.

  Examples include: WAVE Audio Format, FLAC, ACC, Dolby AC-3, mp3, etc.

- ***Video compression methods***

  Used for video formats compression.

  Examples include: MPEG, MPEG-4, VC-1, H.264, etc.

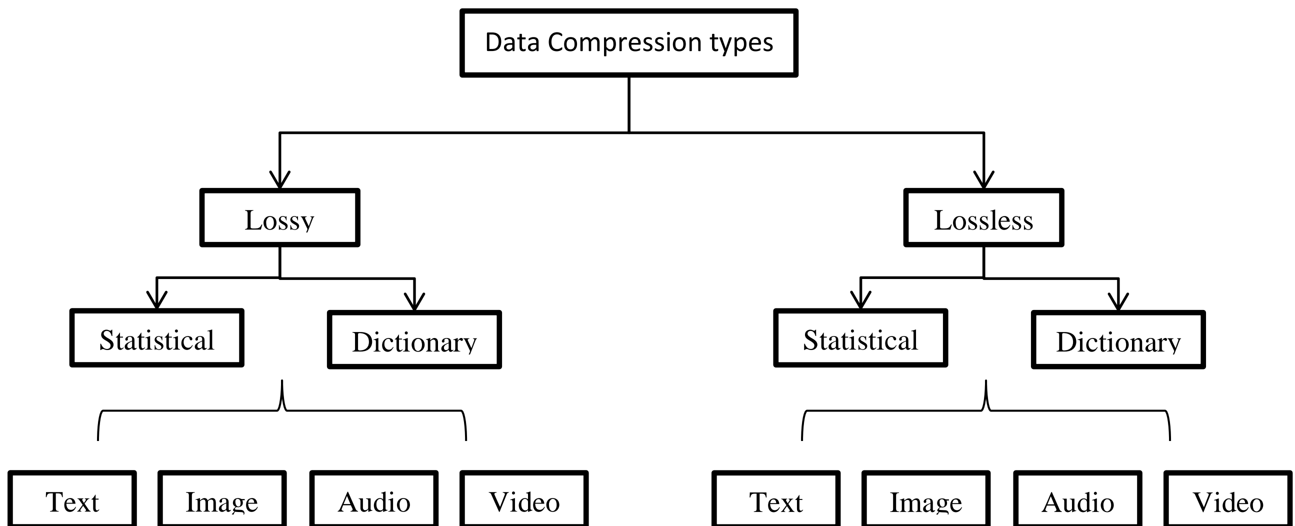



**Figure 2: Data Compression types**

Figure 2 gives the diagrammatic chart representation of all the data compression types. All these methods are aimed at reducing data size. The principles related to data compression suggest that any compression algorithm capable of reducing data to about half its size is considered as a good compression algorithm. The measure of this efficiency is defined by compression ratio.

## 2.3 Compression Ratio

Compression ratio is a simple and effective method of measuring how good a compression scheme can perform. It is defined as:

$$Compression\ Ratio = \frac{size\ of\ the\ output\ stream}{size\ of\ the\ input\ stream} \qquad [1]$$

This measurement for a given data or data set determines the ratio between the compressed data output bits to the raw data input bits of the data set. A ratio of half or less is considered as good compression. Over the past many years some standard data sets have been adopted to

measure the effectiveness of compression schemes. These standards data sets are referred to as “corpus”.

## 2.4 Data Compression Corpora

A corpus is a distinct set of files, used for evaluating the practical performance of different compression schemes [2]. The compression rate is measured in bits per symbol (bps) and is the resulted ratio of the size of the output bytes to the size of the input bytes. A value of 8 bps means no compression; smaller ratio values represent better (stronger) compression. [2]

There are different corpora for different data types. Some corpora contain smaller files whereas others have larger files. Some corpora put the emphasis on text files, others on picture, video, sound or other data type files.

The following is a list of some known available corpus:

- ***The Calgary Corpus***

  Authors: Ian Witten, Timothy Bell and John Cleary

  Year: 1987

  Location: University of Calgary, Canada.

  The Calgary Corpus is a set of 18 files traditionally used to test data compression algorithms and implementations. They include text, image, and object files [1]. The Calgary corpus is a standard for text based and lossless compressions methods. This corpus is available at [3]. The work here uses this corpus for testing and analyzing.

- ***The Canterbury Corpus***

  Authors: Ross Arnold and Timothy Bell

  Year: 1997

  Location: University of Canterbury, New Zealand.

  The Canterbury corpus came as an alternative to the Calgary corpus. The design of Canterbury corpus follows the new era of file’s types which were not present at the

time of Calgary corpus. There are two editions of Canterbury corpus; these are “Standard Canterbury corpus” and “large Canterbury corpus”. This corpus can be found at [4]. This work also uses Large Canterbury corpus for testing and evaluation.

- ***Lukas Corpus***

  Author: Jurgen Abel

  Year: 2006

  Location: Germany.

  The Lukas corpus is a set of medical images mostly considering of two-dimensional (2D) radiographs in different image files format. These files are used to evaluate the practical performance of lossless compression algorithms in the medical imaging field. This corpus is available at [2] and [5].

- ***The Protein Corpus***

  Authors: Craig Nevill-Manning and Ian Witten

  Year: 1999

  Location: Paper from the IEEE Data Compression Conference 1999, Snowbird, Utah, United States of America.

  The Protein Corpus is a set of 4 files, which were used in the article "Protein is incompressible" by Craig Nevill-Manning and Ian Witten from the DCC 1999. Compressing such files is difficult so this corpus provides a good evaluation ground. The corpus is available at [2].

- ***The Silesia Corpus***

  Author: Sebastian Deorowicz

  Year: 2003

  Location: Silesian University of Technology, Poland.

  The Silesia corpus can be considered as a modern file based corpus. This corpus involves large files ranging up to 50 MB for evaluation. This corpus is available at [6].

## 2.5 Data Compression Today

With the rise and advancement of computer technology, data compression has evolved into an art of its own. Today almost every file in a home user computer is in some compressed form. Today's modern data compression schemes and software can easily compress a billion bytes of data to about a quarter of its size or even less. Some of these well-known schemes and software are listed here.

**List of Compression Schemes:**

- Run-Length Encoding (RLE)
- Shannon-Fano Coding
- Huffman Coding
- Arithmetic Coding
- Prediction by Partial Matching (PPM)
- LZ77
- LZW
- UNIX Compression (LZC)
- GIF Images
- Deflate: Zip and Gzip
- LZMA and 7-Zip
- PNG
- JPEG
- DjVu
- MPEG
- MP3
- OGG
- The Burrows-Wheeler Transform Method

- ACB
- Portable Document Format (PDF)

**List of Compression Software:**

- WinRAR
- WinZip
- WinAce
- Stuffit
- WinUHA
- 7-Zip
- ALZip
- BitZipper
- The Unarchiver
- PeaZip
- IZArc

# Chapter 3: PPM Data Compression Scheme

This work parallelizes the PPM data compression scheme. PPM is a two stage data compression scheme that makes efficient use of Arithmetic Coding. The Arithmetic Coding is a statistical lossless data compression scheme often referred to as entropy encoder. Understanding of Arithmetic Coding is a prerequisite to the understanding of PPM data compression scheme.

## 3.1 Arithmetic Coding

Arithmetic coding is a compression technique that is capable of achieving excellent compression on raw data. Its principle strength lies in the fact that it can generate code of length close to Shannon's entropy [7]. Arithmetic coding is a lossless compression technique and part of the statistical methods family.

The principle of arithmetic coding goes back to the early 1960s when it was first proposed by Peter Elias [1]. In 1987 Ian Witten, Timothy Bell and John Cleary came with a practical software implementation of Arithmetic coding that has become well known [8].

Arithmetic coding overcomes the problem of assigning integers codes to individual symbols (or bytes from a data stream) [1]; it does this by assigning a large code to the file. This technique works in two-passes; first it calculates the probabilities of each occurring symbol. The second pass involves the coding of these symbols respective to their probabilities. This two-pass method is illustrated here with some description:

i. Consider a data stream of three reoccurring symbols.
   Let the symbols be $s_1$, $s_2$ and $s_3$.

ii. Divide these symbols along a number line [0, 1).
   Initially all symbols are assigned equal probabilities as shown in figure 3.

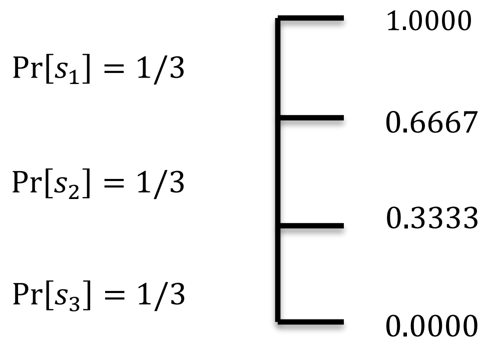



**Figure 3: Arithmetic Coding**

iii. If a symbol occurs and needs to be coded, use the interval respective to its assigned interval on the number line.

To code a symbol say $s_2$ (reference figure 4), its probability range interval is used to redefine the complete number line and all respective probabilities, the occurrence of this symbols $s_2$ also means that now it may have a higher probability. Similarly if another symbol $s_1$ is to be coded (reference figure 5), its new probability range interval is used, every symbol is redefined in this range as before and the probability of this symbol is increased.

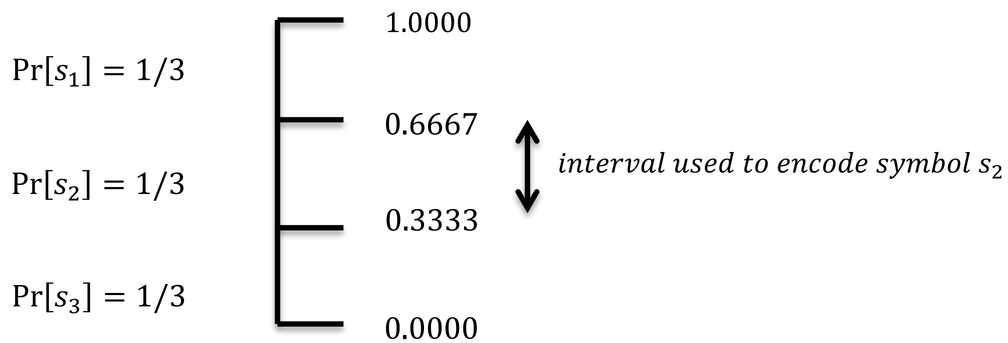



**Figure 4: Arithmetic Coding**

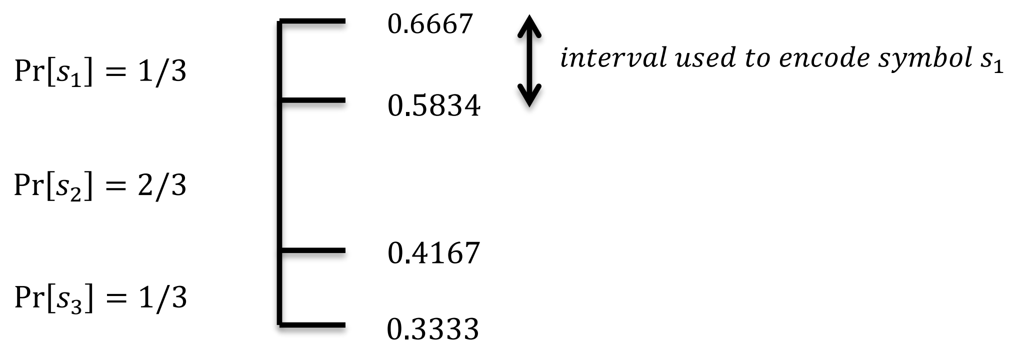



**Figure 5: Arithmetic Coding**

iv. Any value that can exist between the final interval ranges gives the arithmetic code for the data stream.

After coding all the symbols the final probability interval is used which in this case is [0.5834, 0.6667), any value in-between this range can be used to define the complete coded stream.

v. To reverse this process and get the original data stream from the coded (compressed) file, a (similar to forward) backward operation is implemented.

The coded value is used again and again to redefine the complete probability number line [0, 1) until the complete original data stream is achieved.

Although the above method illustrates the concept behind arithmetic coding, in actual practical implementation it is not feasible as it produces a very large decimal value which even today's modern computers can't handle. To overcome this problem the practical implementation uses integer computations. For this, instead of using probabilities it works with frequencies and to represent the current state of a symbol its frequency range is used which in the previous case considering symbols $s_1$ at the beginning of the coding process is:

Low count = 0 & high count = 3

The following pseudo code show steps involved in practical implementation of Arithmetic coding (Referenced from [7]):

- Read symbol from data stream
- Calculate low_count, high_count and total (this is the equal to all symbols read so far)

- Code (**Encode**) the symbol as:

  - ***Set*** r ← range / total
  - ***Set*** low ← low + r × low count
  - ***If*** high count < total then
  - ***Set*** range ← r × (high count − low count)
  - ***Else***
  - **Set** range ← range − r × low count

  - ***While*** range ≤ One_Quarter
    - ***If*** low + range ≤ One_Half
      - bit_plus_follow(0)
    - ***Else If*** One_Half ≤ low
      - bit_plus_follow(1)
    - ***Set*** low ← low − One_Half
    - ***Else***
    - ***Set*** bits_outstanding ← bits_outstanding + 1
    - ***Set*** low ← low − One_Quarter
    - ***set*** low ← 2 × low and range ← 2 × range

  - bi_plus_follow(b):
    - put_one_bit(b)
    - ***While*** bits_outstanding > 0 ***do***
    - put_one_bit(1 − b)
    - ***set*** bits_outstanding ← bits_outstanding − 1

- **Decode** as:
  - State variables *range* and *diff*: $value - low$ are maintained.
  - ***Set*** r $\leftarrow range\ /\ total$
  - ***Set*** $target \leftarrow \min\{ total - 1, diff \text{ div } range \}$
  - search for symbol (s) such that $low\ count \le target < high_count$
  - ***Set*** $diff \leftarrow diff - r \times low_count$
  - ***If*** $high_count < total$
  - ***Set*** $range \leftarrow r \times (high_count - low_count)$
  - ***Else***
  - ***Set*** $range \leftarrow range - r \times low_count$

  - ***While*** $range \le One_Quarter$
    - ***Set*** $range \leftarrow 2 \times range$
    - ***Set*** $diff \leftarrow 2 \times diff + get_one_bit()$
    - ***Output*** s

Altogether Arithmetic Coding is a good compression scheme which can be used as a solo encoder/decoder or as a combination with others useful data compression schemes. One such scheme is PPM which invokes the Arithmetic Coder in an excellent manner.

## 3.2 Prediction by Partial Matching (PPM)

The prediction by partial matching (PPM) is a sophisticated, state of the art compression scheme originally developed by J. Cleary and I. Witten (1984) [10]. This scheme is based on an encoder that maintains a statistical model of the text [1]. PPM falls in the statistical compression methods and is a lossless type data compression scheme.

In PPM, the encoder inputs the next symbol, assign it a probability and send this symbol to an arithmetic encoder so that it can be coded. In simpler words PPM alters the symbol's

probabilities that are used by an arithmetic coder. It improves the symbol’s probabilities to such an extent that they may be coded as close as their entropy.

The arithmetic coder has a drawback; it restricts itself to one context length. On the other hand PPM uses finite context length models (usually up to 3, 4 or more). During encoding, at each stage the symbol to be coded is first searched in the longest context and if it is available it is encoded using this context probability, else an escape symbol which is a special symbol to represent a context order is encoded and the model switches to a lower length context to search again for the symbol. In PPM the model can switch to as low as context order “-1”, here all the symbols are available and are assigned an equal probability distribution. This lowest context order encodes a symbol in its original size. In the decode phase, PPM reads the compressed stream, here if it finds an escape symbol it switches to a lower context order else it uses the current context to decode the symbol.

Similar to an arithmetic coder; PPM is also a two-pass method, first includes the probabilistic modeling of finite length context models and next is the encoding based on these models. For this second pass there is no fix restriction of using arithmetic coder, any other method which may require a good statistical model can be used. Suitable coding methods that can replace arithmetic coder in PPM includes Huffman coding and range coding, however some special variants of dictionary based LZ schemes also make useful use of PPM.

One critical issue that effects a PPM implementation is how and what probability should be assigned when an escape event occurs. To address this issues various mechanisms have been suggested, these are known as variants of PPM.

### 3.2.1 PPM Variants

Each PPM variant approach the problem of assigning probabilities to escape events. Currently there are ten (10) variants of PPM.

- PPMA (Method A)
- PPMB (Method B)

  John G. Cleary and Ian H. Witten, 1984 [10].

- PPMC (Method C)

  Alistair Moffat, 1990 [11].

- PPMD (Method D)

  Howard, 1993 [12].

- PPMII

  Dmitry Shkarin, 2002 [13].

- PPMP ( Poisson Distribution)

  Ian H. Witten and T. C. Bell, 1991 [14].

- PPMX (Poisson Distribution Approximate)

  Ilia Muraviev, 2008 [15].

- PPM*

  John G. Cleary, W. J. Teahan and Ian H. Witten, 1993 [16].

- PPMZ (Method Z)

  Charles Bloom, 1996 [17].

- Fast PPM

  Paul G. Howard and Jeffery Scott Vitter, 1994 [18].

### 3.2.2 PPM method C (PPMC)

PPMC a variant of PPM (Prediction by Partial Matching) data compression scheme comes in line after the original PPM (method A and B). PPMC also known as PPM method C was suggested by A. Moffat in his 1990 paper [11].

Like other PPM variants PPMC also solves the problem of assigning probabilities to the escape events. PPMC assigns a probability of one to an escape symbol every time it sees a new symbol in a particular context. When an escape event is called it uses this (escape symbol) probability to encode the event and move to a lower context order. Every new symbol occurrence increases the escape symbol probability by one. The maximum probability that an escape symbol can attain in PPMC is equal to the number of unique distinct symbols in that context.

# Chapter 4: Parallel Computing

## 4.1 Parallel Computing

Parallel Computing is a computational form where many executions or process are carried out side by side. It implies simultaneous execution of multiple instructions. It is based on the principle that large problems can often be divided into smaller ones, which can then be solved concurrently (in parallel) [19] that is a simple "divide and conquer" strategy. Parallelism in computing has been employed for many years, mainly where high-performance computing is needed. Interest and developments in Parallel Computing has grown lately due to the easily availability of today's multicore/multiprocessor based personal computers.

Parallelism (as in parallel computing) can be achieved at several different stages, including:

- Bit level
- Instruction level
- Data level
- Task level

The central idea behind parallel computation is to achieve a faster computation environment. Parallelism slays the processing time when multiple process are run as processes are no more dependent on one processing unit, and depending on the amount of processing hardware available no queue may be necessary. But as with many other good things this faster computation and lesser processing time come at a price of greater usage of the following:

- power consumption
- processing units (hardware)
- memory consumption

Compared to sequential computer programs, parallel computer programs are difficult to write as concurrency introduces several new classes of potential software bugs [20]; among these communication and synchronization between the subtask are the most common ones which restricts good parallel performance.

Modern parallel programs and software uses data and task level parallelism, with data level being the most common. Data level parallelism is mostly achieved from multithreading whereas task level through task parallelism (process parallelism). In multithreading the threads (parts or instructions of a process) share hardware resources. In task parallelism resources are discrete. Classification of parallel computing falls under parallel architectures.

## 4.2 Parallel Architectures

The word parallel architectures correspond to the parallel models used in the parallelization of PPM data compression scheme. A total of four different parallel models have been implemented in this work. Self-authored names are used for all these parallel models. Some parallel architectures are similar to the ones describe in many literatures on parallel computing, whereas others use a new approach to parallelism. The parallel models show the flow of data through the processing elements (cores). These models are illustrated with some description as below:

### 4.2.1 Parallel Lines

This model is the simplest and the most common known parallel implementation that can be used for almost any process. Although named here as “Parallel Lines” in many literature this has often been mentioned as “task parallelism” or simply “task parallel”. The model work in a simple straight forward manner, it divides the large input data into smaller (input) data chunks and sends each chunk to a separate processing element which runs a copy of the original process.

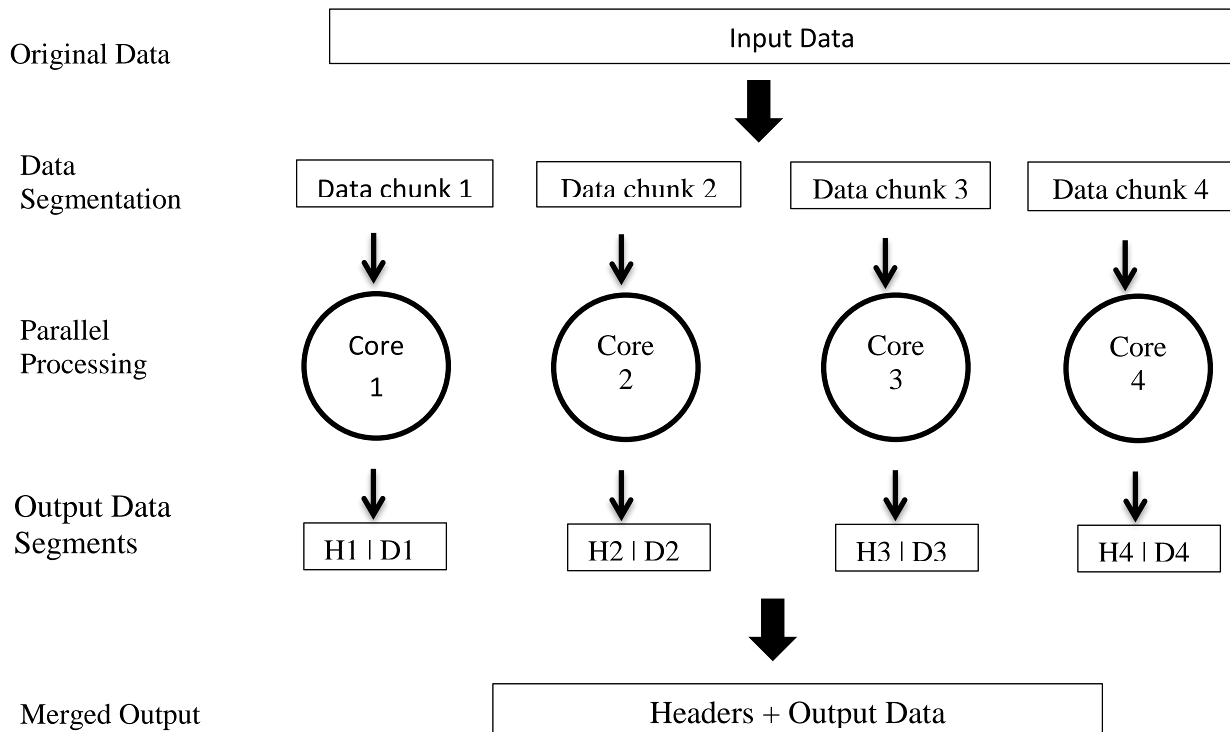



**Figure 6: Parallel Lines Model**

The figure above (figure 4) depicts Parallel Lines model. Core1 processes data chunk-1, core2 processes data chunk-2 and so forth. This model produces two outputs. The first contains the intermediate output data chunks of each core (processing element) and their respective header, whereas the second final output is the merger of all the first stage output's chunks. In this architecture the processing of multiple chunks of original data by multiple processing elements at the same time makes execution faster.

### 4.2.2 Sequential Pipes

The "Sequential Pipes" is a simple sequential (serial) pipeline model that uses multicores. Many Processes can be divided into a two stages or more; therefor this model uses two cores or more

in a pipeline manner to complete the task. Such pipeline architecture is often referred to as "data message passing parallelism".

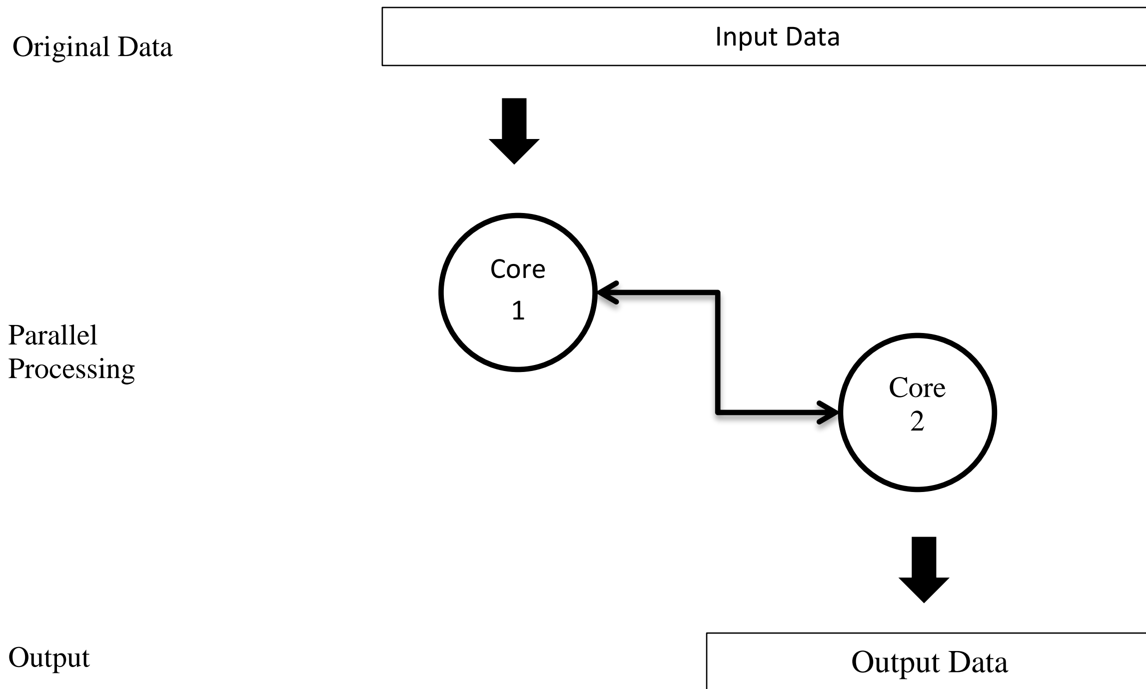



**Figure 7: Sequential Pipes Model**

The figure above (figure 5) shows Sequential Pipes model. In this data is fetched by the first core which processes it and passes it to the second core which reprocesses it. In this implementation the second core purposely lags the first core by a minimum time of t+1 unit (here t depicts first core start time). The output is obtained from the second core. A number of data passing (message passing) stages can be used depending on the task to be accomplished. This architecture saves time only when the original process can be divided into multiple pipeline stages.

### 4.2.3 Parallel Pipes

The Parallel Pipes model is the task parallelization of the Sequential Pipes model. It combines the features of both Parallel Lines (task parallel) and Sequential Pipes (pipeline or data message passing) models in one. Parallel Pipes divides the larger input data file into smaller chunks and then on these chunks applies multicore pipeline processing model.

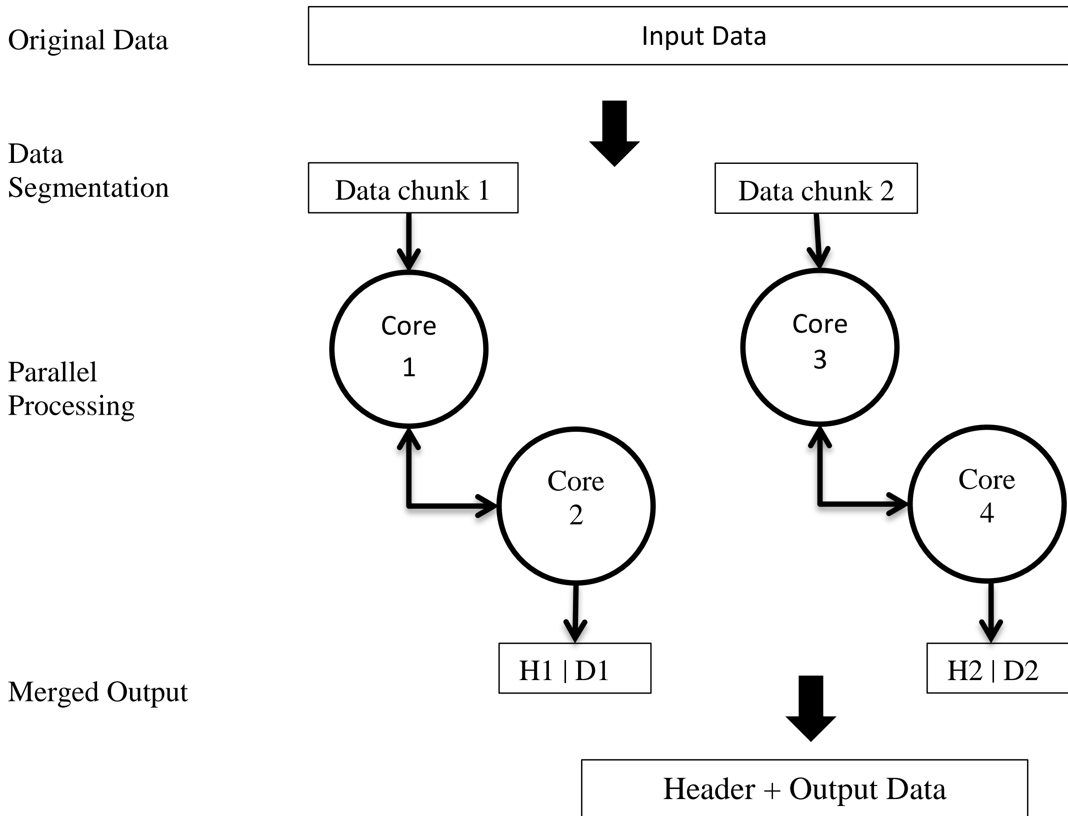



**Figure 8: Parallel Pipes Model**

The figure above (figure 6) shows Parallel Pipes model. This model follows both task parallelism and data parallelism. First the large data stream is divided in to equal smaller chunks and then separately processed in a pipeline style data (message) passing manner. The number of parallel pipeline stages is equal to the number of data chunks.

### 4.2.4 Hybrid Tree

This model is also similar to a multicore pipeline model with a key difference that it uses multiple (in parallel) processing elements in the first stage. This model is based on "decision tree" model where the final stage is been driven by the results of the previous.

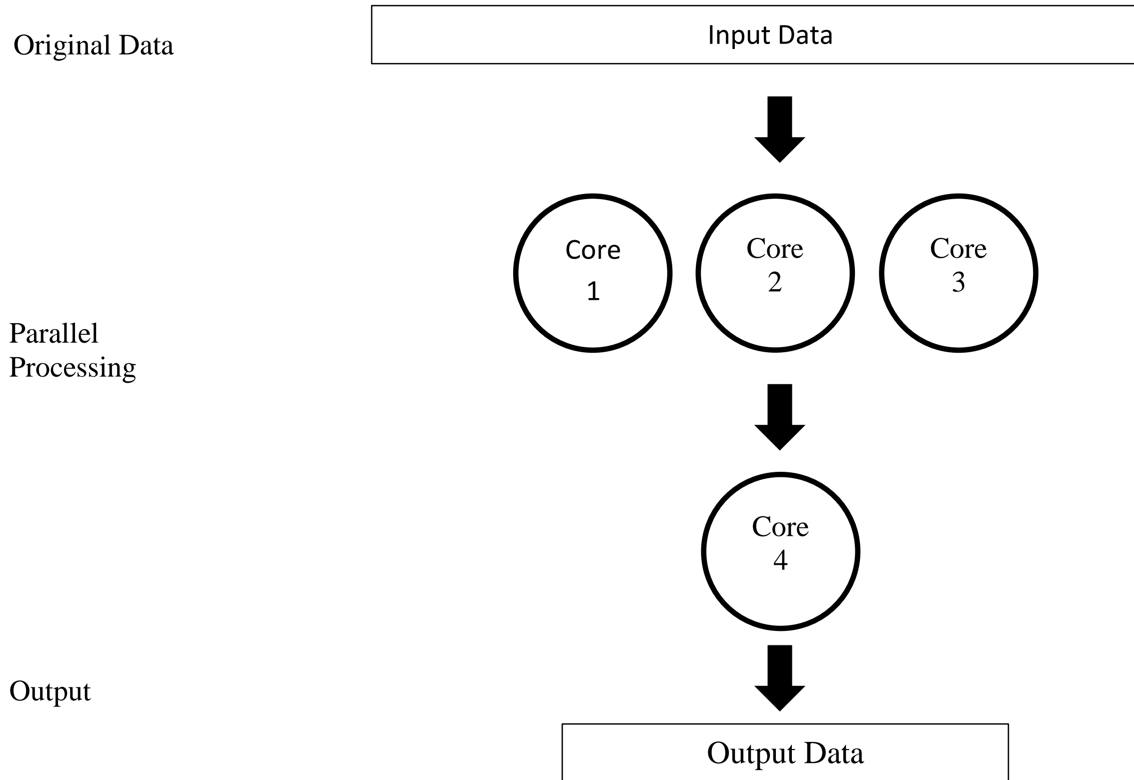



**Figure 9: Hybrid Tree Model**

Figure 7 shows a Hybrid Tree Model. In implementation, first a copy of data is passed to all the parallel processing elements (cores) in the first stage whose processed output is used to drive the final stage. These intermediate outputs can also be used in a decision-making manner by the final stage. Note that the processing elements in the first stage are in parallel and not dependent on one another; any dependencies of the first stage elements on one another can cause a parallel deadlock. Also to be noted here is that each processing element in the first stage forms a pipeline model with the final stage processing element.

# Chapter 5: Parallel Data Compression

Parallelism in data compression involves the use of parallel computing methodologies on compression schemes to reduce processing time. It has been observed that mostly lossless data compression schemes and methods which outperforms in compression (that is have a very low compression ratio) usually performs lazy at processing time. This laziness is inherited from the compression method's algorithm which usually involves excessive looping, searching and data access.

The goal of parallelization is to overcome the critical processing phases which add time to a process execution. In data compression these phases involves (a few common phases) accessing data stream, searching, sorting, storing, looping and memory access.

In general, data compression algorithms and software are not designed to work concurrently; parallel data compression is only applied for very large data streams. Operating a compression method sequentially on a very large file can be quite slow. Often large files can take several minutes to compress. Usually these large files are divided into smaller chunks and send to separate processing units where each processing unit runs through a copy of the applied compression method. This sort of parallel processing is governed by Amdahl's law [21] which states:

"Theoretically, by doubling processing elements, execution time should halve and with the second doubling, execution time should halve again."

Practically, in data compression, a very few parallel implemented methods achieve this optimal speed up and that to when doubling is applied once or a couple of times (not more). Another fact associated with compression methods is that fine-tuning an algorithm to squeeze out the last remaining bits of redundancy from the data gives diminishing returns [1]. Modifying an algorithm to improve compression by 1% may increase the run time by 10% and the complexity

of the program by more than that [1]. So theoretically it can be said that parallel data compression can indeed work out good for data compression schemes.

## 5.1 Parallel Implementation of Compression Methods

Over the past few years a number of researches have been conducted to examine the possibilities of parallelism in data compression. Not every attempt made resulted in success. Here a few of them have been reviewed.

Parallelism to lossless dictionary based compression methods is the most common. Several researchers have worked on parallel implementation of dictionary base LZ variants. In [22] the author decodes LZ2 in a parallel environment. In [23] the BZip2 method is parallelized by dividing the input data into chunks and using the same copy of the compression method on available processors in parallel. Similar parallel environment is seen in [24] parallel image compression. The authors in [25] also applies this data chunks based task parallel architect on lossless textual data compression methods, including statistical based Huffman, Arithmetic coding and dictionary based LZ78 and LZW. All these approach the processing speed up problem by using dividing the larger data chunk into smaller and then applying parallel processing. This approach speeds up processing time in resemblance to Amdahl's law, but in many cases degrades the compression and adds a permanent header to output.

However many have approached this processing problem differently so as to avoid header and degrading of compression. In [26] the authors uses cooperative dictionary for their LZ77 parallel implementation to secure the compression ratio. In [27] parallel Suffix Sorting implementation is applied to improve processing time. In paper [28] on parallel BWT compression also successfully test message passing (multiple stage pipeline architecture) in parallel environment.

## 5.2 Parallel PPM (method C)

PPM and any of it variants can be considered as a two-pass (two stages) based compression methods. The first pass starts modeling and the second deals with encoding. In any adaptive implementation of PPM both these stages are heavily dependent on each another. Processing speed up can be and has been achieved in both these stages in a well designed and implemented sequential environment but no speed up through the use of parallel processing elements have yet been done.

This works deals with processing speed up of PPMC using parallel architect. A total of four (4) different parallel implementations have been described and compared with the sequential architecture. Some parallel architectures are similar to the ones describe in the previous, whereas other uses a new approach to parallelism. Not all of these parallel implementations achieve similar results; also the results may vary for different variants of PPM and for different compression schemes. For the ease of use, self-authored names have been proposed for these parallel architectures.

# Chapter 6: Parallel Software Implementation of PPM (method C) Data Compression Scheme

The PPM (method C) is lossless data compression scheme. Practical implementation of PPMC is quite resource hungry in both memory and processing time. Parallel software implementation of PPMC data compression scheme works on the latter issue by parallelization of the said data compression scheme. This research work implements the previously mentioned parallel architectures for the PPMC method. Various internal and external parameters are studied which effects the parallelization process of PPMC. At the end, the results obtained (of all parallel architectures) are compared. This work utilizes software environment for testing the PPMC's parallel architectures. The tools used are divided into two categories:

- Software Environment
- Hardware Environment

The following topics describe the software n hardware tools/environment and how these have been used for the parallel implementation of PPMC data compression scheme.

## 6.1 Software Environment

The software environment is comprised of the followings:

- C# (Programming Language)
- .Net 4.0 Framework
- Task Parallel Library
- Visual Studio 2010
- Windows 7

### 6.1.1 C# Programming Language

C# (pronounced as C - sharp) is Microsoft's premier language for .NET development. It leverages time-tested features with cutting-edge innovations and provides a highly usable, efficient way to write programs for the modern enterprise computing environment. It is one of the languages that is use to create applications that will run in the .NET CLR. It is an evolution of the C and C++ languages and has been created specifically to work with the .NET platform.

C# was created at Microsoft late in the 1990s and was part of Microsoft's overall .NET strategy. It was first released in its alpha version in the middle of 2000 [29]. C# 1.0 made its public debut in 2001. The advent of C# 2.0 with Visual Studio 2005 saw several important new features added to the language, including Generics, Iterators, and anonymous methods. C# 3.0 which was released with Visual Studio 2008, added extension methods, lambda expressions, and most famously of all, the Language Integrated Query facility, or LINQ. The latest incarnation of the language, C# 4.0, provides further enhancements that improve its interoperability with other languages and technologies. These features include support for named and optional arguments, the *dynamic* type which indicates that the language runtime should implement late binding for an object, and variance which resolves some issues in the way in which generic interfaces are defined. C# 4.0 takes advantage of the latest version of the .NET Framework (also version 4.0). There are many additions to the .NET Framework in this release, but arguably the most significant are the classes and types that constitute the Task Parallel Library (TPL). The TPL makes it possible to use C# for building highly scalable applications that can take full advantage of multi-core processors quickly and easily.

### 6.1.2 .Net Framework

The .NET Framework (now at version 4, version 4.5 available as developers preview) is a revolutionary platform created by Microsoft for developing applications [30]. The .NET Framework consists primarily of a gigantic library of code that can be used from client

languages (such as C#) using object-oriented programming (OOP) techniques [30]. This library is categorized into several different modules. A new key feature to .Net (4.0) is Parallel Extensions to improve support for parallel computing, which target multi-core or distributed systems.

### 6.1.3 Relation between C# and .Net Framework

Although C# is a computer language that can be studied on its own, it has a special relationship to its runtime environment, the .NET Framework. The reason for this is twofold. First, C# was initially designed by Microsoft to create code for the .NET Framework. Second, the libraries used by C# are the ones defined by the .NET Framework. Thus, even though it is theoretically possible to separate C# the language from the .NET environment, the two are closely linked. (Referenced from [29])

### 6.1.4 Task Parallel Library (TPL)

The Task Parallel Library (TPL) is a set of public types and APIs in the System.Threading and System.Threading.Tasks namespaces in the .NET Framework version 4. The purpose of the TPL is to make developers more productive by simplifying the process of adding parallelism and concurrency to applications.

The Task Parallel Library is designed to make it much easier to write managed code that can automatically use multiple processors. Using this library, one can conveniently express potential parallelism in existing sequential code, where the exposed parallel tasks will be run concurrently on all available processors. Usually this results in significant speedups.

### 6.1.5 Visual Studio

Visual studio is a development tool, an integrated development environment (IDE) by Microsoft that supports coding of Microsoft CLR languages including C#. It is a development tools from which simple command-line applications to more complex project types like Microsoft's Office

suite can be designed. The current complete release available is Visual Studio 2010; however a recent Visual Studio Express 2011 as developer's preview is also available.

### 6.1.6 Windows 7

Windows 7 is the latest release of Microsoft Windows (although currently Windows 8 as developer's preview is also available), a series of operating systems produced by Microsoft for use on personal computers, including home and business desktops, laptops, netbooks, tablet PCs, and media center PCs. Windows 7 was released in late 2009. Windows 7 includes a number of new features, such as advances in touch and handwriting recognition, support for virtual hard disks, improved performance on multi-core processors, improved boot performance, direct access, and kernel improvements.

A Key added feature to Windows 7 is its better support to parallel processing. A note from Computex 2009 says; Intel has revealed that Windows 7 features new and improved multi-threading, which will help to improve power consumption and battery life. Previous versions of Windows often swapped threads around cores, which prevented them from entering lower power states and caused cache thrashing as separate cores raced to grab data processed by others. The Windows 7 kernel changes this by improving thread affinity, locking threads to particular cores in order to allow unused CPU cores to enter low power C-states when they're not in use (called thread parking) providing the CPU and motherboard supports this of course.

Windows 7 automates and scales all running tasks and applications in accordance with the underneath available hardware architecture. On a multicore hardware, Windows 7 by default uses the available cores to start multithread activity so that best performance can be attained.

## 6.2 Hardware Environment

The hardware environment containing the parallel processing elements is:

### 6.2.1 Intel Core i5

The Intel core i5 is a multicore (multiple processors) based CPU on the Nehalem microarchitecture. Nehalem processors use the 45 nm process. The core i5 series were introduced in the late 2009. Some of these series processors at the end line are comparable to the core i7 series. The core i5 (750) used in this work is a quad core processor. With four CPUs this is comparable to some similar quad core (4 CPUs) core i7 series and is even consider being faster at performance comparatively.

## 6.3 PPMC Implementation

The PPMC implemented for this work uses arithmetic coder for symbols encoding/decoding, whereas context modeling uses a maximum count of two (2) for context orders due to limited amount of processing elements available within the core i5 hardware. The source code for the PPMC algorithm is written in C#. The context modeling portion source code is based on the theoretical representation of PPMC, whereas the encoder portion uses a self-modified (for PPMC) C# version of “Eric Bodden” arithmetic coder. The original C# version of Eric Bodden’s AC is written by “Sina Momken”. Eric Bodden’s AC work is based on the 1987 original work of “Ian Witten, Timothy Bell and John Cleary” [8] which modify the original work to allow processing of larger data streams.

All the implementations (sequential and parallel) are based on strict OOP (object oriented programming) and C# syntax. No external libraries or data types are used; all data types and structures defined and used are part of the present C# and .Net framework.

The context orders are based on classes (objects) that can uses either arrays or dictionary (similar to hash tables) for storing context that have been occurred. The symbols frequencies are stored as integers (32 bits) data type. Data is read byte by byte so a total of 257 (256 for each individual byte and 1 for the escape symbol) symbols at the most may occur. In C#, an array’s length needs to be defined when initializing the array. Lower orders are less memory hungry therefore for these arrays are suitable as arrays perform fast at runtime.

### 6.3.1 Context Orders

The implementations of these orders are as follows:

***i. Order -1***

This is the lowest order in any PPM implementation. In this order all symbols are present and are assigned an equal probability. As there are a total of 257 symbols so an integer type array of 257 elements initialized to 1 for all elements can be used. But to save memory and make processing faster a simple fact can be used that the low count and high count assigned to a symbol (byte) in this context order is always equal to that byte and byte + 1. Whereas the total for this order remains constant, which in this case is 257. If implemented like this no array may be required to store order -1, low and high counts can be directly computed at runtime. The escape symbol in this context order is used to only specify the EOF (end of file) symbol.

***ii. Order 0***

Order 0 is based on the occurrences of the symbol followed by the same symbol. This order uses an integer type array of 257 elements. If a same symbols occurs followed by itself the frequency (count of that element in the array) is increased. The total count for this order can be stored as a separate integer type or as part of the array. In the latter case the array length is increased by one to accommodate for this total count. In this manner the total memory allocated at runtime to this order is 258 x 32 (bits) which are 1032 bytes.

***iii. Order 1***

This order stores the occurrence of a symbol followed by any other symbol. For such an operation a two-dimensional (2D) array is ideal. This order uses a 2D integers array of 256 x 258 elements. The total memory used by this order is 256 x 258 x 32 (bits), which equals to 264,192 bytes.

### *iv. Order 2*

Order 2 uses two previous contexts to predict the next symbol. An integer's array based implementation of this can become resource hungry and slow down the processing if enough memory is not available at runtime. An array based storage uses three-dimensional (3D) array of 256 x 256 x 258 x 32 bits, this amounts equal to approximately 65 MB (Mega Byte) of memory usage. This could become a hurdle where less memory is available. An alternative method is to use Dictionary or Hash tables to store only the contexts that have been occurred. This saves memory consumption and makes implementation of higher context orders possible.

The C# Dictionary is similar to Hash tables which offer a managed solution without the use of object casting as usually seen with the usage of Hash tables. For higher orders nested dictionaries can be used as a single dictionary cannot store previous contexts. In case of three stages nested dictionary, the total memory usage per symbol is 16 bytes. If the total occurring symbols in this order is less; then this order (dictionary version) saves a very good amount of memory compared to 3D array version for this context order.

The arithmetic coder used for the encoding/decoding part does not require any large memory storage. It only needs to store the range, low and high of every previous operation at runtime. Modification to the arithmetic coder includes a version that uses memory stream to store the compressed output stream. Other modifications are to the data types used by the low, high, total count and the arguments and references passed to the arithmetic coder (applies to both file stream and memory stream version). The difference between file stream and memory stream version of Arithmetic Coder is that the latter version stores complete output data in memory first before writing it to a file on disk.

### 6.3.2 Pseudo Code

The following is the pseudo code of the PPMC implemented:

- **Initialize:**
    - Arithmetic coder (file stream OR memory stream)
    - Symbol (an object type to hold the symbol read, its low, high and total count)
    - Context Order -1
    - Context Order 0
    - Context Order 1
    - Context Order 2
    - File stream OR Memory stream (for input / output)

- **Encoder:**
  - **While** (read symbol from file != EOF)
  - Start from highest order
      - Order 2 (search symbol)
          - ***If*** found
          - send symbol to arithmetic encoder for encoding
          - ***Else***
          - send escape to arithmetic encoder for encoding
          - switch Order (to lower order)
          - Update Order

      - Order 1 (search symbol)
          - ***If*** found
          - send symbol to arithmetic encoder for encoding
          - ***Else***
          - send escape to arithmetic encoder for encoding
          - switch Order (to lower order)
          - Update Order

- Order 0 (search symbol)
    - ***If*** found
    - send symbol to arithmetic encoder for encoding
    - ***Else***
    - send escape to arithmetic encoder for encoding
    - switch Order (to lower order)
    - Update Order

- Order -1 (search symbol)
    - ***If*** found
    - send symbol to arithmetic encoder for encoding

- **Decoder:**
    - Start from highest order
    - **Switch** (Order)

        - Case 2:
            - Symbol ← decode compressed stream
            - Order 2 (search symbol)
            - ***If*** found
            - send symbol to arithmetic decoder for decoding
            - Output → symbol
            - update Order 2
            - update Order 1
            - update Order 0
            - switch Order to highest order
            - ***Else***
            - send escape to arithmetic decoder for decoding
            - switch Order (to lower order)

- Case 1:
    - Symbol ← decode compressed stream
    - Order 1 (search symbol)
    - ***If*** found
    - send symbol to arithmetic decoder for decoding
    - Output → symbol
    - update Order 2
    - update Order 1
    - update Order 0
    - switch Order to highest order
    - ***Else***
    - send escape to arithmetic decoder for decoding
    - switch Order (to lower order)

- Case 0:
    - Symbol ← decode compressed stream
    - Order 0 (search symbol)
    - ***If*** found
    - send symbol to arithmetic decoder for decoding
    - Output → symbol
    - update Order 2
    - update Order 1
    - update Order 0
    - switch Order to highest order
    - ***Else***
    - send escape to arithmetic decoder for decoding
    - switch Order (to lower order)

- Case -1:
    - Symbol ← decode compressed stream
    - Order -1 (search symbol)

- ***If*** found
- send symbol to arithmetic decoder for decoding
- Output → symbol
- update Order 2
- update Order 1
- update Order 0
- switch Order to highest order

All the PPMCs in this work including the sequential and parallel implementations use this above pseudo code with exception to some modifications in the letter case. The maximum context order used in all implementations is two ‘2’. Time performance is measured using the standard C# .Net “System.Diagnostics” namespace’s stopwatch time class. The following describe all these PPMC’s implementation.

### 6.3.3 Sequential PPMC

The sequential PPMC is used as a benchmark to compare every parallel implementation. The time factor of this implementation is compared with the parallel implementations time to note the speed up if any. The sequential PPMC is a non-multicore and non-multithread based version. It is the same as the pseudo code given previously. It uses file stream version of the arithmetic coder along with all the context model orders mentioned previously.

### 6.3.4 Parallel Lines PPMC

This is a parallel PPMC implementation based on the Parallel Lines architecture as defined before. This PPMC accommodates for two, three and four (2, 3 and 4) parallel processing elements (in line with the core i5 architecture) or parallel lines at the same time. The implementation of Parallel Lines PPMC uses memory-mapping to map the input data stream to the memory. This is necessary as multiple accesses to a same file on disk at the same time are

not possible; there is only one read head to a disk drive which cannot be shared at the same time.

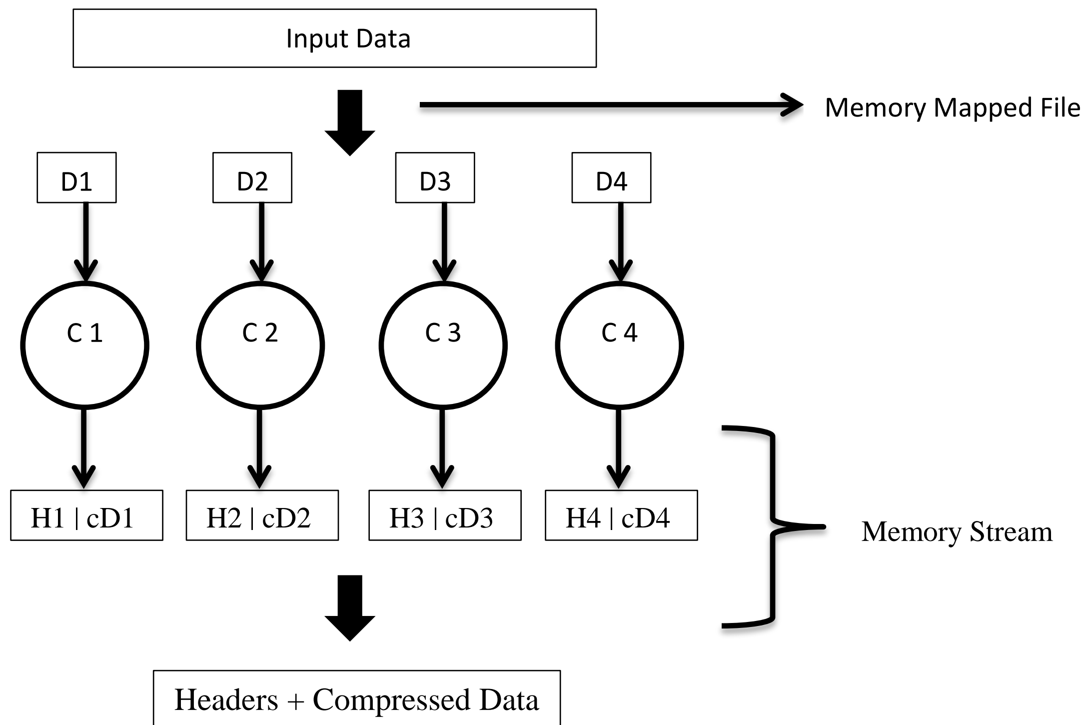



**Figure 10: Parallel Lines PPMC**

The memory-mapping or the memory mapped file class is introduced with the .Net 4.0 framework. Memory mapped files maps a file from the disk to the memory so that multiple process can access it. Using memory mapping multiple access of input data stream at the same time becomes possible for the parallel lines PPMC. This PPMC uses Memory stream version of arithmetic coder to store the intermediate output of each parallel element. The final output is a merger of this previous output.

The merged output (compressed) file size is more than the sequential PPMC compressed output size. One reason for this is the addition of header to the compressed data.

Note that the usage of memory mapping and memory stream adds up to the memory in use during runtime. Typically this amount for memory mapping is approximately equal to the input

file (or input buffer size) and for memory stream equals to the compressed output of each processing element.

The following is the pseudo coding for Parallel Lines PPMC encoder:

- **Initialize:**
  - Arithmetic coder (memory stream)
  - Symbol (an object type to hold the symbol read, its low, high and total count)
  - Context Order -1
  - Context Order 0
  - Context Order 1
  - Context Order 2
  - File stream and Memory stream (for input / output)
  - Memory mapped file (pointed to complete input stream)
  - Memory mapped file assessors (equal to the number of parallel tasks)

- **Encoding:**
  - Divide input mapped file to chunks
  - Call parallel tasks (equal to the number of chunks)
  - Pass data chunks (one to each task)
  - Run PPMC on each task
  - Store compress output as separate memory streams
  - Wait for all tasks to finish
  - Add header to all (sub) outputs
  - Combine outputs
  - Write output to file

### 6.3.5 Sequential Pipes PPMC

The Sequential Pipes PPMC in this implementation use just one pipeline stage. It is based on the fact that PPMC like other PPMs is a two-pass scheme. The pipeline stage starts from the modeling and ends at the encoder. Both these stages use separate processing elements (cores) and the latter stage lags the first by a time of t+1 unit.

The second processing elements use the results of the first. If a direct communication is made between the two parallel units a deadlock can occur. To avoid parallel deadlock a buffer is used. The result of first gets stored in the buffer, the second processor then fetches data from this buffer.

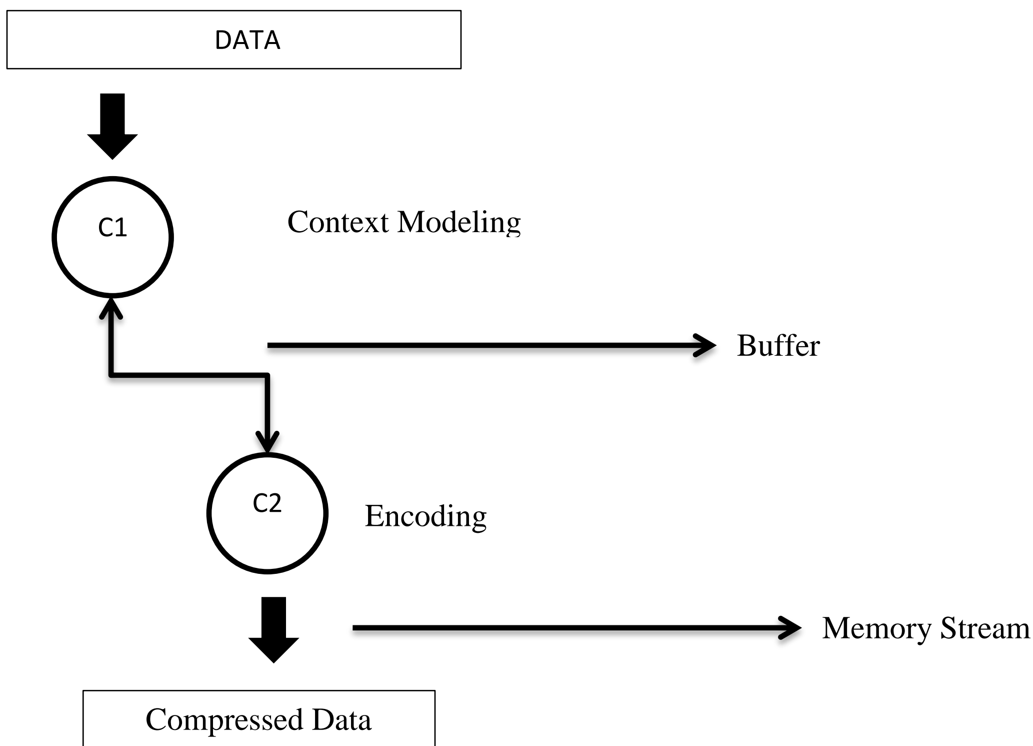



**Figure 11: Sequential Pipes PPMC**

The Sequential Pipes PPMC uses the file stream for input access and memory stream to temporary handle output until the disk drive head becomes available for the file write operation. The usage of buffer to pass data between processes adds up to the total memory usage. At the most without any restriction, the buffer size approximates to:

$$\text{input file length} \times \text{number of context order} \times 4(\text{symbol}, \text{low}, \text{high}, \text{total counts})$$

The compressed output file size in this case is same as that achieved using Sequential PPMC. The following is the pseudo code for Sequential Pipes PPMC encoder:

- **Initialize:**
    - Arithmetic coder (memory stream)
    - Symbol (an object type to hold the symbol read, its low, high and total count)
    - Context Order -1
    - Context Order 0
    - Context Order 1
    - Context Order 2
    - File stream and Memory stream (for input / output)
    - Initialize buffer (having appropriate length to hold all symbols)

- **Encoding:**
    - Call two parallel tasks
    - Assign task 1 to read data from input and output to buffer (use PPMC context model)
    - Assign task 2 (t-1 seconds) delay and read from buffer and output to memory (use Arithmetic coder)
    - Wait for all tasks to finish
    - Write output from memory to file

### 6.3.6 Parallel Pipes PPMC

The Parallel Pipes PPMC is a combination of both Parallel Lines and Sequential Pipes PPMC. This implementation splits the input data stream so that multiple instances of pipeline PPMCs can work on the data chunks.

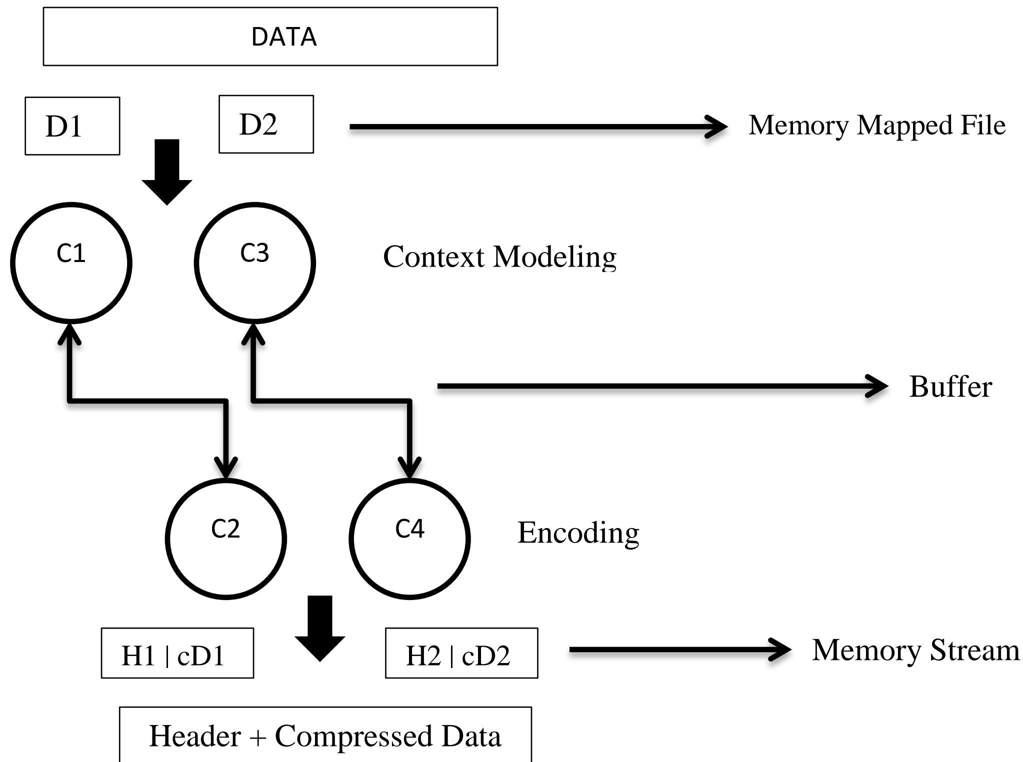



**Figure 12: Parallel Pipes PPMC**

As parallel pipes PPMC is a combination of parallel lines and sequential pipes PPMC it uses the features of both implementations. For multiple accesses to the input data stream it uses memory mapped file feature of parallel lines PPMC and to store the outputs of first and second stages processors it uses buffer and memory stream feature of sequential pipes PPMC.

The parallel pipes PPMC implementation in this work uses only two stage task parallelism that is only two parallel lines. Additional memory usage at runtime is proportional to the memory

consumed by memory mapping, buffer and memory streams, this memory count can go up to (if no restriction is impose on the inherited features):

$$\text{input file length} + \text{input file length} \times \text{number of context order} \\ \times 4(\text{symbol, low, high, total counts}) \\ + \text{each intermediate compressed output file length}$$

The compressed output file size of Parallel Pipes PPMC measures more than Sequential PPMC but less than Parallel Lines PPMC for the same number of cores used.

The following is the pseudo code for Parallel Pipes PPMC encoder

- **Initialize:**
  - Arithmetic coder (memory stream)
  - Symbol (an object type to hold the symbol read, its low, high and total count)
  - Context Order -1
  - Context Order 0
  - Context Order 1
  - Context Order 2
  - File stream and Memory stream (for input / output)
  - Memory mapped file (pointed to complete input stream)
  - Memory mapped file assessors (equal to the number of parallel tasks)
  - Initialize buffer (having appropriate length to hold all symbols)
- **Encoding:**
  - Divide input mapped file to chunks
  - Call parallel tasks inside call pipeline PPMC tasks
  - Wait for all tasks to finish
  - Add header to all (sub) outputs
  - Combine outputs
  - Write output to file

## 6.3.7 Hybrid Tree PPMC

Hybrid Tree PPMC implementation use the Hybrid Tree (parallel) architecture as describe previously. In this implementation it takes advantage of the multiple context orders of PPMC and processes them separately. The result of modeling stages is fetched and used by the encoder which acts as if a pipeline stage to the modeler.

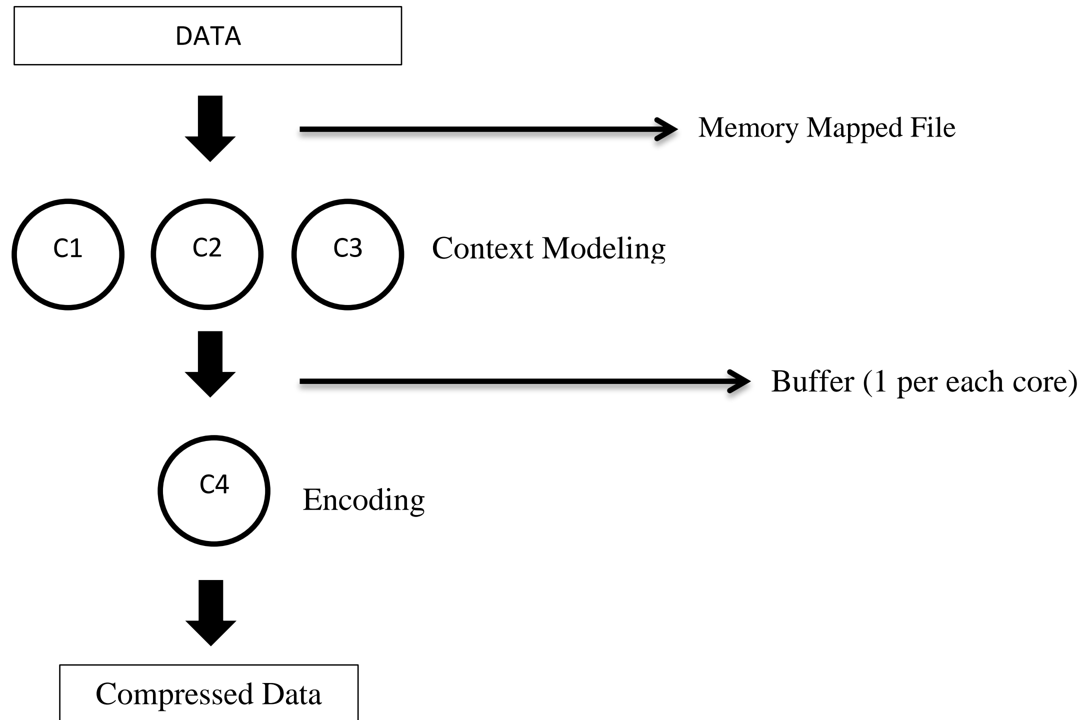



**Figure 13: Hybrid Tree PPMC**

The entire context modeling stages reads the input data stream at the same time. The input data stream is not fragmented but is memory mapped so that multiple processes can access it at the same time. This implementation assigns one processing element (core) to each context model. No communication exists between the context models. This is necessary to avoid any possible parallel deadlock. The modeling stages output is stored separately in a buffer which is then fetched by the encoder processor. One buffer is assigned to each parallel context modeling stage, again this is a necessary precaution use to avoid parallel deadlock. The startup time for the encoder processor is “t (last) +1” where “t (last)” signifies the startup time for the

last started context modeling processor. Additional memory storage at runtime includes memory mapping and buffers, this if no restriction can be equal to:

$$\text{input file length} + \text{input file length} \times \text{number of context orders} \times 4(\text{symbol}, \text{low}, \text{high}, \text{total})$$

The output file size in Hybrid Tree PPMC is better from all other task parallelism based implementations. In this case the compressed file size is same as that of sequential PPMC.

The following is the pseudo code for Hybrid Tree PPMC encoder:

- **Initialize:**
    - Arithmetic coder (file stream)
    - Symbol (an object type to hold the symbol read, its low, high and total count)
    - Context Order -1
    - Context Order 0
    - Context Order 1
    - Context Order 2
    - File stream (for input / output)
    - Memory mapped file (pointed to complete input stream)
    - Memory mapped file assessors (equal to the number of parallel tasks)
    - Initialize buffer (having appropriate length to hold all symbols)

- **Encoding:**
    - Call parallel task and assign order 0
    - Call parallel task and assign order 1
    - Call parallel task and assign order 2
    - Call parallel task and assign encoder
    - Wait for all tasks to finish
    - Write output to file

# Chapter 7: Test Analysis and Results

The PPMCs implemented are tested for time speed up. All implementations (as previously describe) are tested on quad core architecture. At test time, the test environments specifications are:

- Windows 7 (operating system)
- Intel core i5 750 (quad core)
- 2 GB physical memory (RAM)

Before testing the system was completely checked for any malfunctioning; including physical memory, cores, motherboard and hard disk malfunctioning. No hardware components are over clocked, all cores run at their default specifications. The operating system is also tested for the presence of software bugs, viruses and malfunctioning. At testing time all unnecessary process running through the operating system were shut off. Same environment is sustained for all PPMC's implementations testing.

As the goal of this work is to measure parallel time speed up, therefore no memory restrictions are imposed on any PPMC under test. Hard disk access is limited to the hard disk default read/write rate. For read/write operations only the standard primary hard disk (containing the operating system) is used no other secondary disk storage medium is used.

The parallel PPMC implementations uses memory mapping, memory streams, and parallel task library, therefore as a prerequisite time based measurements of these are taken. Note that memory mapping, memory stream and task parallel library are part of C# and .Net framework.

Perquisite measurements:

i. **Memory mapping**

   In C# programming language (through .Net framework) memory mapping of a file on disk implies creating an image of the file in memory that points to the original file. The memory mapped file can be of the same size (as that of the original file) or can be custom size. Initializing a memory mapped files requires time. In general the average approximate time can be:

   (Test performed on file name "world192.txt" available as part of the large Canterbury Corpus. File size is 2.35 MB)
   Initialization time: 375 milliseconds

Read access time: 42 milliseconds
Total time (approx.): 417 milliseconds

Compared to file stream that is direct file access:

| Unit = milliseconds | Memory Mapped File | File Stream |
|---|---|---|
| **Initialization time** | 375 | 210 |
| **Access time** | 42 | 27 |
| **Total time** | 417 | 237 |

**ii. Memory stream**

Memory stream which although is much similar to file stream resides only in physical memory. The average approximate time to initialize and perform a read/write access to a memory stream is:

*Test 1:*
(Test performed on custom made file having same length as that of "world192.txt" from the large Canterbury Corpus. File size is 2.35 MB)

Initialization time: 0.0123 milliseconds
Write time: 29 milliseconds
Read time: 13 milliseconds

*Test 2:*
Reading from file to memory and writing from memory to file.
(Test performed on file name "world192.txt" available as part of the large Canterbury Corpus. File size is 2.35 MB)

Initialization time: 0.2325 milliseconds
Write time: 59 milliseconds
Read time: 51 milliseconds

Comparison between file stream and memory stream:

| Unit = milliseconds | Memory Stream | File Stream | Composite |
|---|---|---|---|
| **Initialization time** | 0.0123 | (write access)<br>0.3941<br>(read access)<br>0.2105 | 0.2325 |
| **Write time** | 29 | 33 | 59 |
| **Read time** | 13 | 27 | 51 |

iii. **Task parallelism**

Starting a parallel processing task takes time. The time taken to initialize a parallel task is:

(Task chosen is to print “Hello World!” to console window)
Parallel task time: 2.1946 milliseconds

Compared to sequential processing:

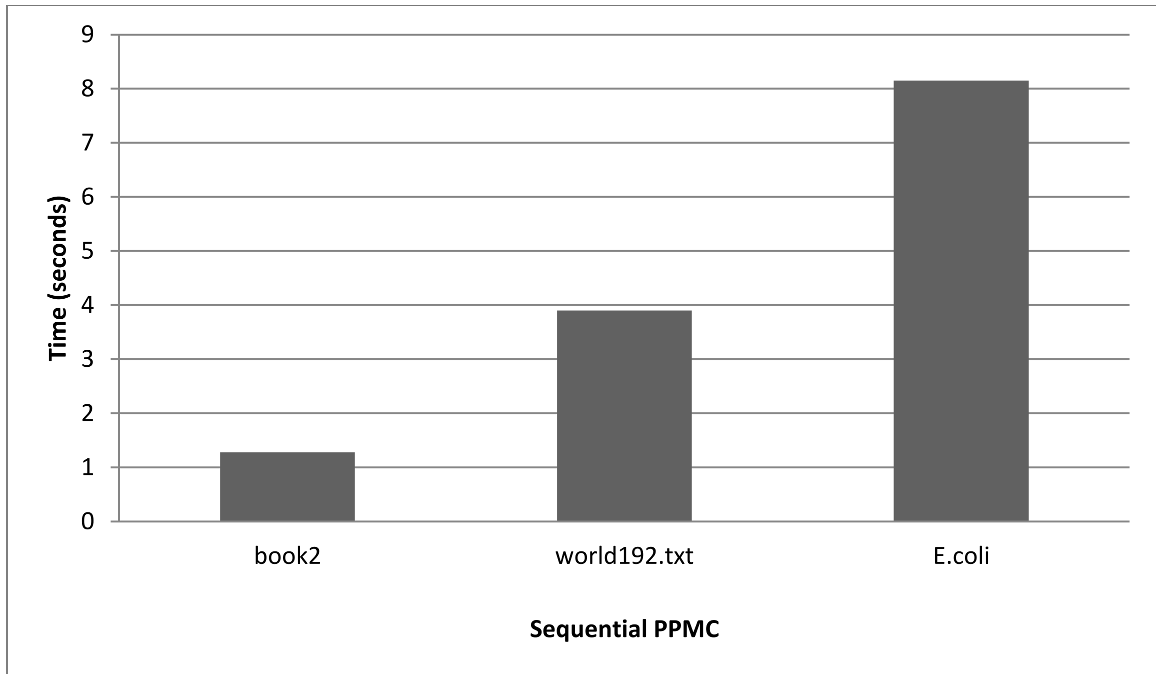
| | **Sequential** | **Parallel** |
|---|---|---|
| **Time (seconds)** | 0.3241 | 2.1946 |

## Parallel PPMCs test

The actual test, analysis results for the PPMCs are as follows. Large Calgary corpus and Canterbury corpus which offers larger size files is used during all PPMC’s test. The results drawn are based on time and compression efficiency offered by each implementation.

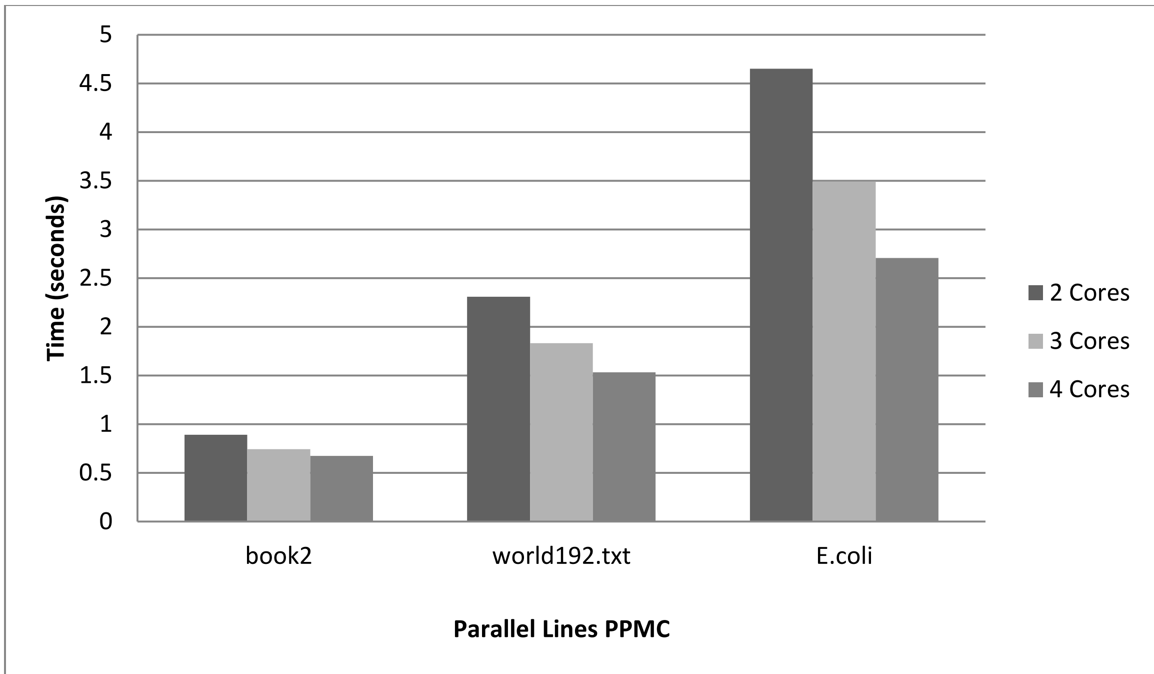
i. **Sequential PPMC**

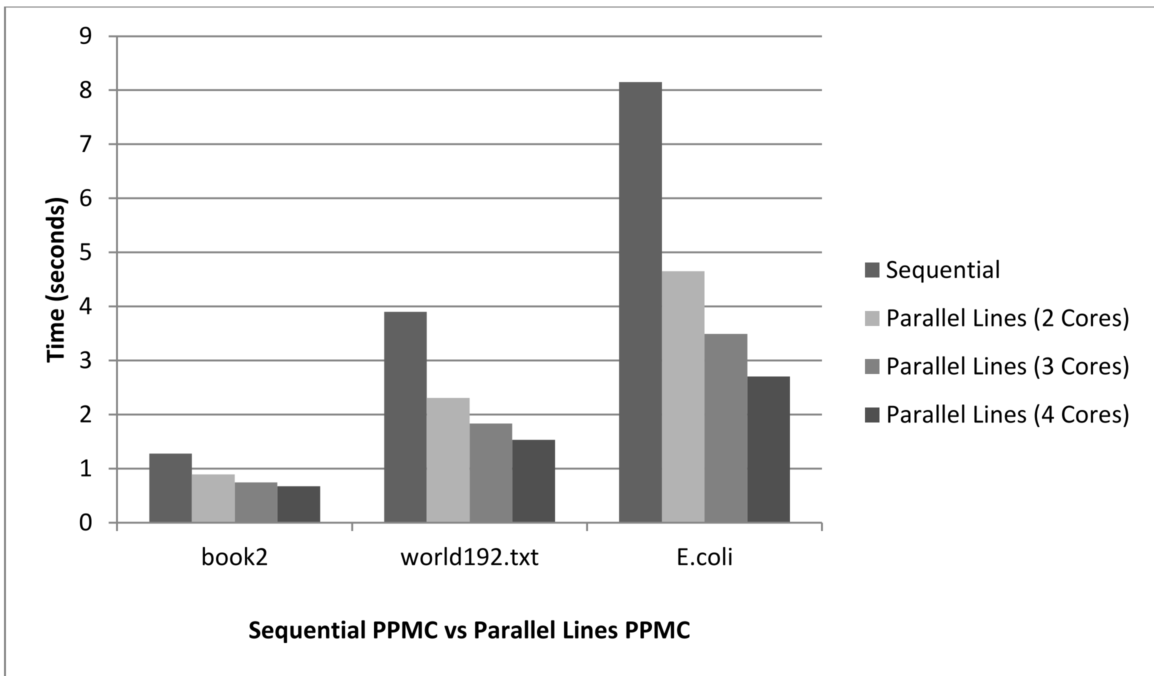
| File name | File size (KB) | Time (seconds) | Compression (KB) |
|---|---|---|---|
| book2 | 597 | 1.2801761 | 283 |
| world192.txt | 2,416 | 3.8991613 | 985 |
| E.coli | 4,530 | 8.1521292 | 1,118 |




### ii. Parallel Lines PPMC

| File name | File size (KB) | Number of Cores | Time (seconds) | Compression (KB) |
|---|---|---|---|---|
| book2 | 597 | 2 | 0.8910034 | 308 |
| book2 | 597 | 3 | 0.7439570 | 329 |
| book2 | 597 | 4 | 0.6733790 | 343 |
| | | | | |
| world192.txt | 2,416 | 2 | 2.3077443 | 1,051 |
| world192.txt | 2,416 | 3 | 1.8322877 | 1,100 |
| world192.txt | 2,416 | 4 | 1.5327735 | 1,141 |
| | | | | |
| E.coli | 4,530 | 2 | 4.6493431 | 1,123 |
| E.coli | 4,530 | 3 | 3.4909464 | 1,127 |
| E.coli | 4,530 | 4 | 2.7048271 | 1,131 |

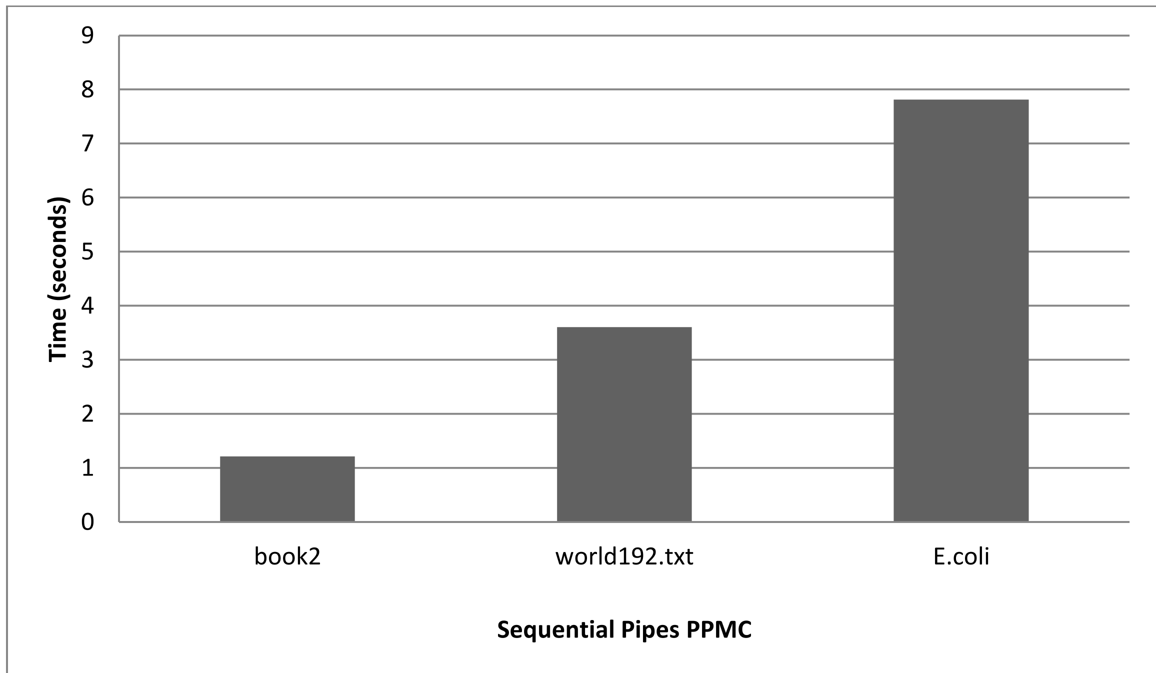


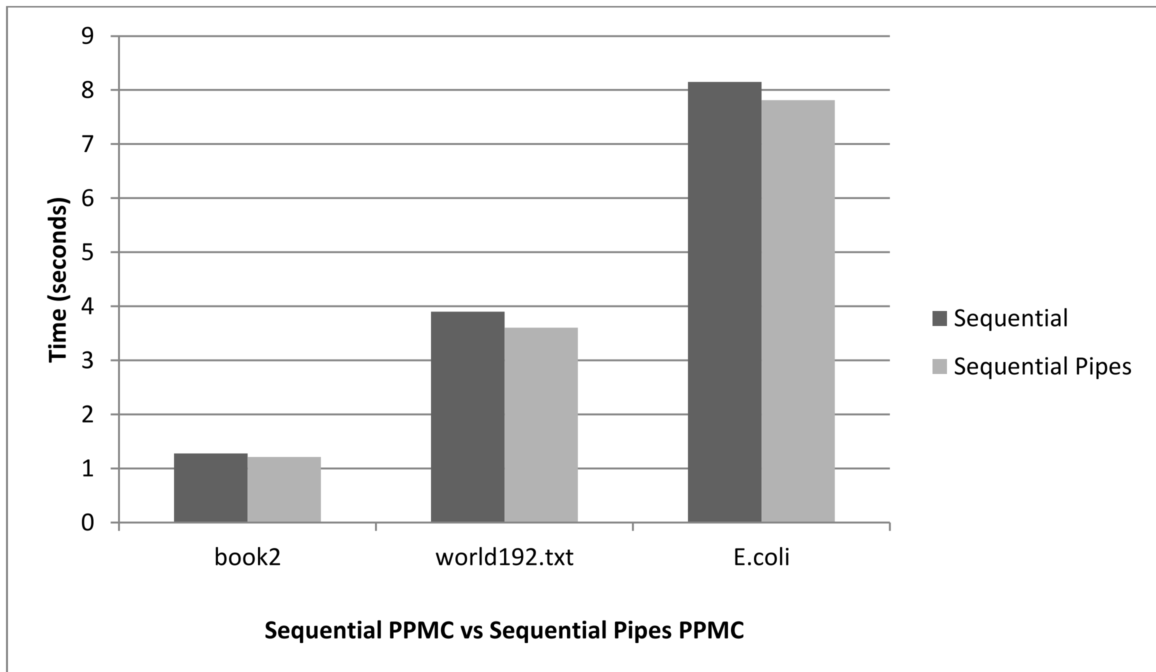




The above comparison graph shows that Parallel Lines PPMC cuts the processing time down to half for 2-Cores, however for more cores the time is not half but is rather cut down to quarter of the previous stage.

### iii. Sequential Pipes PPMC

| File name | File size (KB) | Time (seconds) | Compression (KB) |
|---|---|---|---|
| book2 | 597 | 1.2108961 | 283 |
| world192.txt | 2,416 | 3.6066420 | 985 |
| E.coli | 4,530 | 7.8152796 | 1,118 |

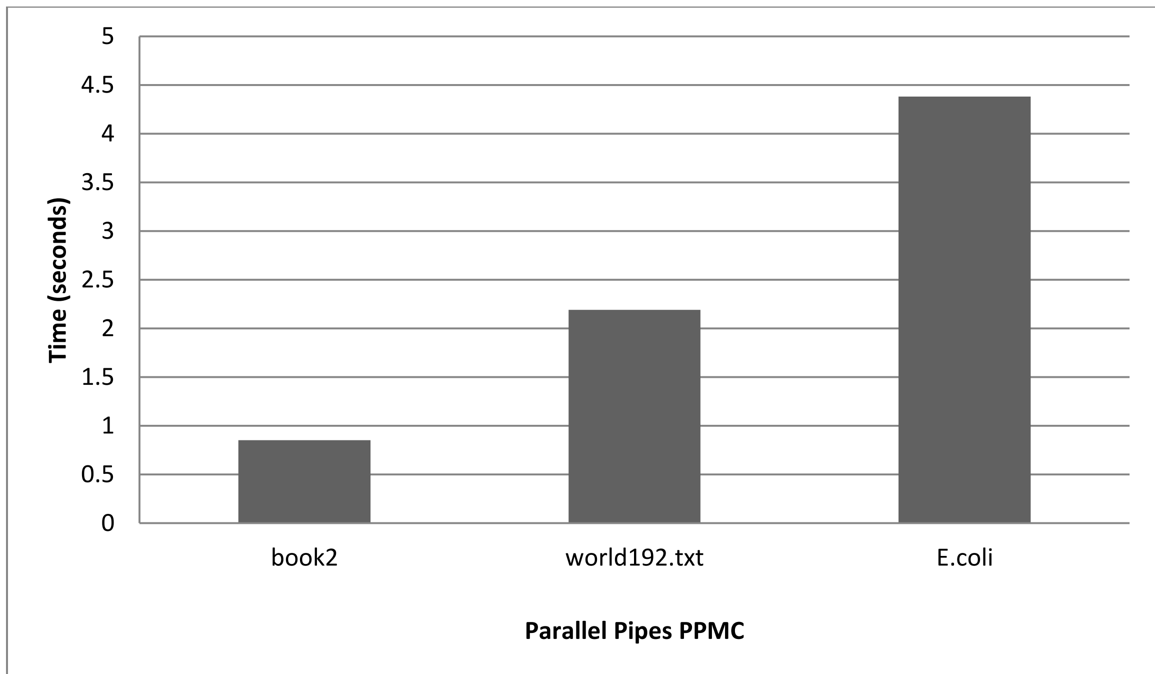


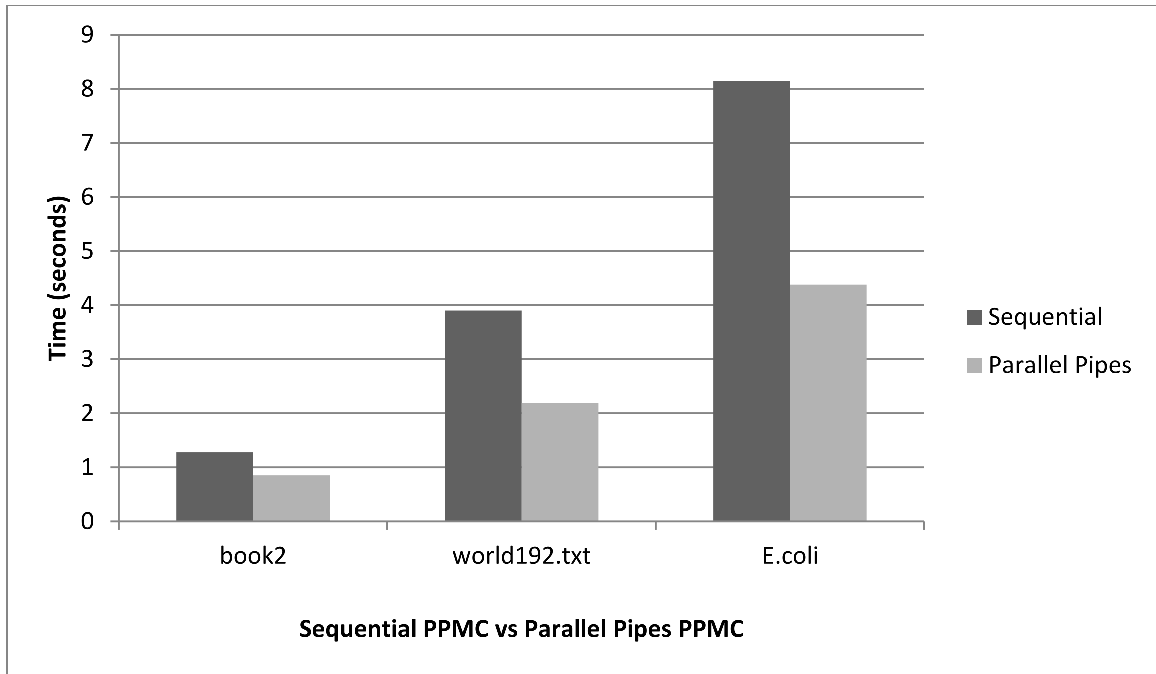



The test shows that Sequential Pipes PPMC does reduce time. This processing speed up increases gradually with the input file size.

### iv. Parallel Pipes PPMC

| File name | File size (KB) | Time (seconds) | Compression (KB) |
|---|---|---|---|
| book2 | 597 | 0.8505831 | 308 |
| world192.txt | 2,416 | 2.1899484 | 1,051 |
| E.coli | 4,530 | 4.3799399 | 1,123 |

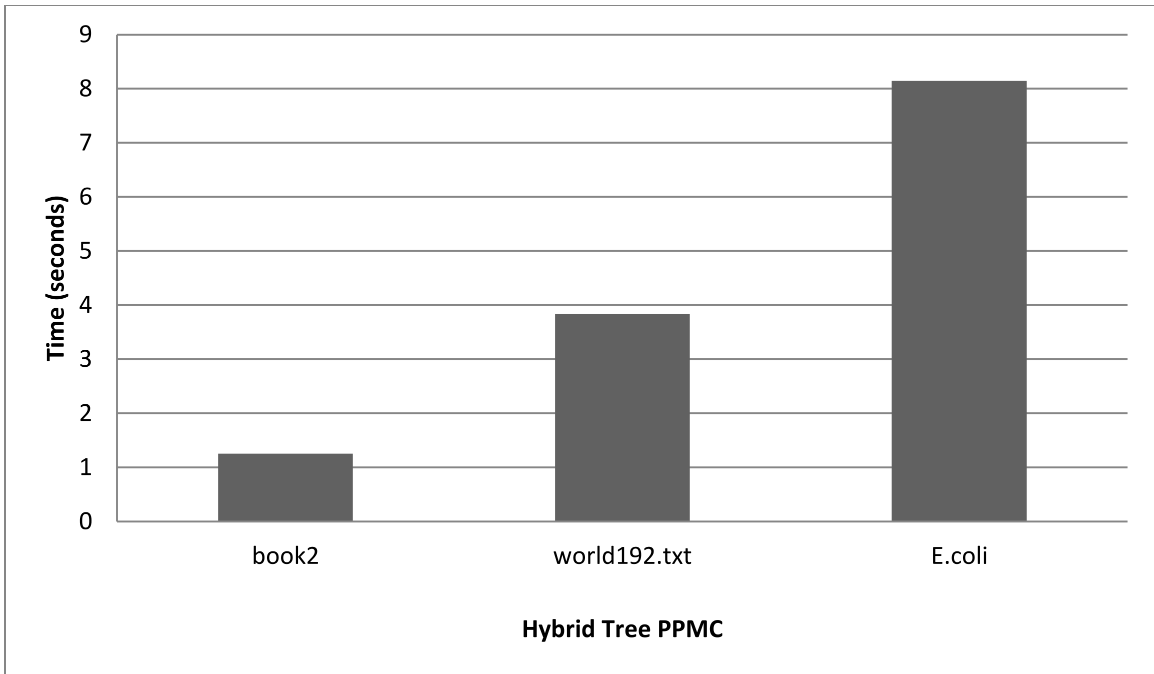


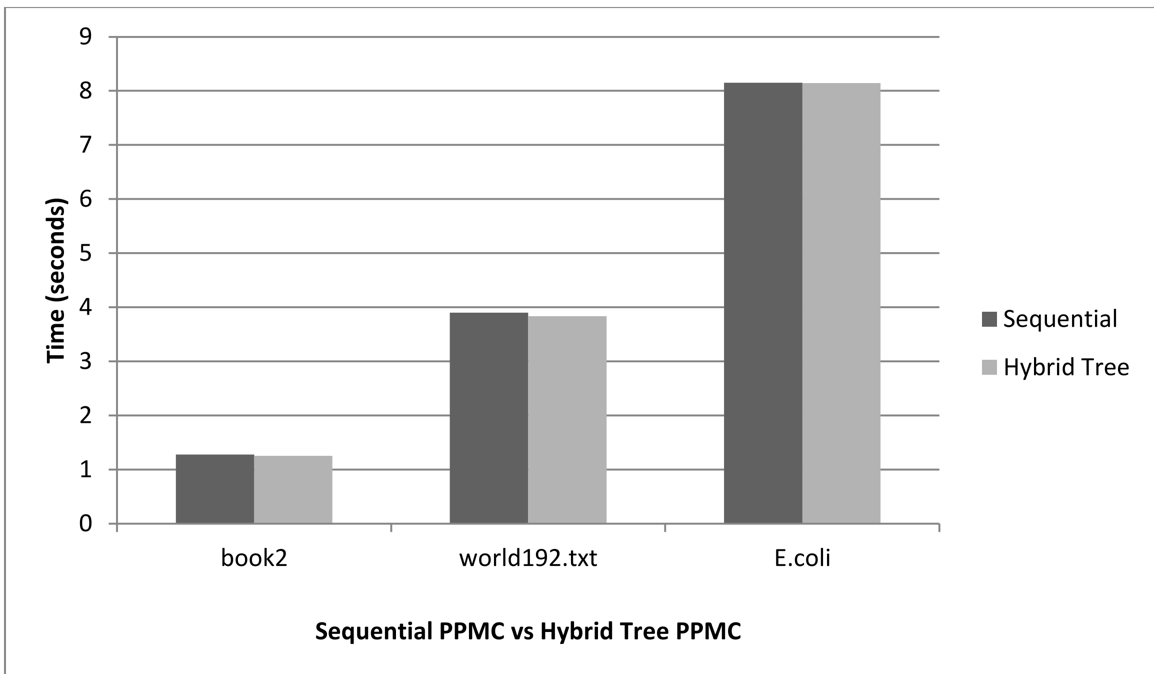



The Parallel Pipes PPMC reduces time to half, but this reduction comes at a very high hardware cost which in this case is 4-Cores. Comparatively 2-Cores based Parallel Lines PPMC shows also similar results.

**v. Hybrid Tree PPMC**

| File name | File size (KB) | Time (seconds) | Compression (KB) |
|---|---|---|---|
| book2 | 597 | 1.2542444 | 283 |
| world192.txt | 2,416 | 3.8353077 | 985 |
| E.coli | 4,530 | 8.1442850 | 1,118 |

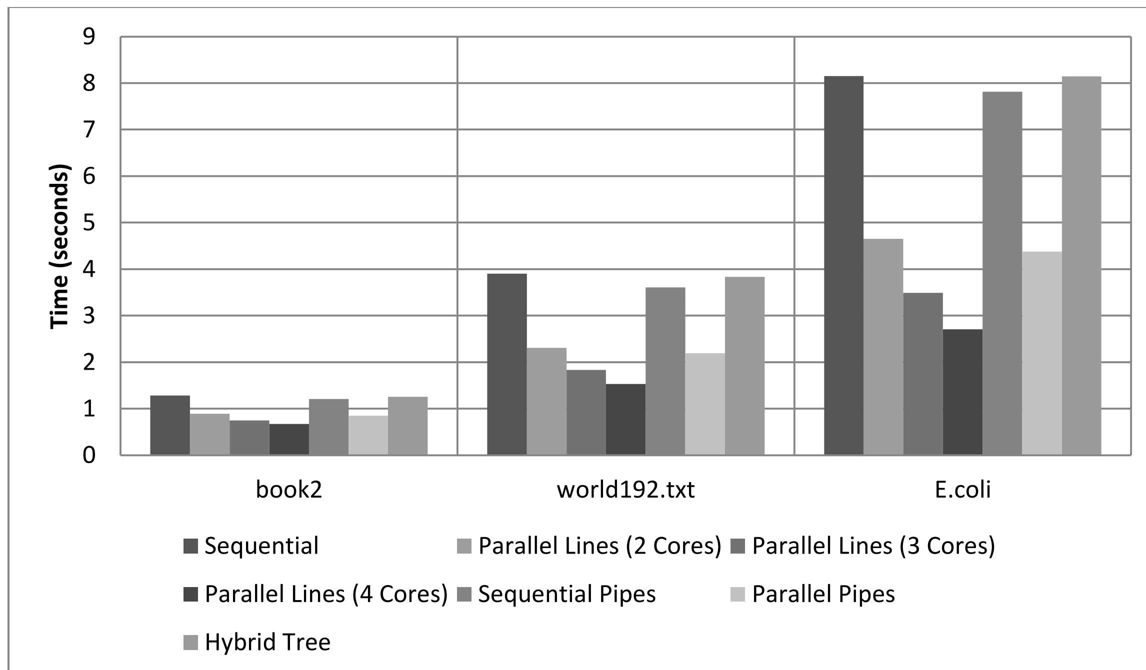



The model give constant speed gain no matter what the input file size may be. Hybrid Tree PPMC also uses 4-Cores (in our case). The speed gain is very less compared to the amount of hardware in use.

## Results Comparisons:

| | File size (KB) | Sequential PPMC | | Parallel Lines PPMC | | Sequential Pipes PPMC | | Parallel Pipes PPMC | | Hybrid Tree PPMC | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Time | Size | Time | Size | Time | Size | Time | Size | Time | Size |
| book2 | 597 | 1.280 | 283 | 0.891 | 308 | 1.210 | 283 | 0.850 | 308 | 1.254 | 283 |
| | | | | 0.743 | 329 | | | | | | |
| | | | | 0.673 | 343 | | | | | | |
| | | | | | | | | | | | |
| world192 .txt | 2,416 | 3.899 | 985 | 2.307 | 1,051 | 3.606 | 985 | 2.189 | 1,051 | 3.835 | 985 |
| | | | | 1.832 | 1,100 | | | | | | |
| | | | | 1.492 | 1,141 | | | | | | |
| | | | | | | | | | | | |
| E.coli | 4,530 | 8.152 | 1,118 | 4.649 | 1,123 | 7.815 | 1,118 | 4.379 | 1,123 | 8.144 | 1,118 |
| | | | | 3.490 | 1,127 | | | | | | |
| | | | | 2.704 | 1,131 | | | | | | |



# Chapter 8: Conclusion

## 8.1 Concluding Remarks

All the PPMC implementations describe previously have been tested and compared. No two PPMC's implementation shows similar results. Test results comparison shows that usage of one or more parallel processing unit diminishes computational time. However each parallel implementation diminishes this time varyingly.

The Parallel Lines PPMC shows the most suitable results for time speed up. It utterly half the processing time of sequential implementation on first doubling. However each addition of parallel element after the first doubling doesn't reduces time to half rather it reduces time by "half time of the previous stage". Due to the nature of PPMC algorithm, Parallel Lines PPMC does not produce good compression compared to sequential implementation. With a header attached, in the worst case it nearly doubles the compression ratio achieved by the sequential PPMC.

The Sequential Pipes PPMC seems to be a good compromise between the sequential and Parallel Lines PPMC. It does reduce time but not as clearly as parallel lines implantation. However compared to Parallel Lines, it produces exact compression ratio as that of sequential PPMC. The processing speed up achieved by this implementation increases exponentially with the file size.

The Parallel Pipes PPMC also half the sequential implementation's time. But it gives this output at a very high hardware cost. Comparatively almost this same time and compression can be achieved with Parallel Lines PPMC for only two (2) processing elements.

The Hybrid Tree PPMC show similar results as that of Sequential Pipes PPMC in both timings and compression, but this applies only to smaller file size. Hybrid Tree PPMC also comes with this speed up at a very high hardware (memory usage) cost. The memory consumption at runtime by Hybrid Tree PPMC is nearly that of Parallel Pipes PPMC.

In order to keep a compromise between speed and hardware use; the Parallel Lines PPMC model is the best choice to consider.

## 8.2 Future Works

The subject Parallel Computing is quite vast. A lot of development is occurring in it with the easy multicore systems availability to end users. As mentioned before; this work is the first in line which applies parallel computing to a “Prediction by Partial Matching” (PPM) method therefor a lot of work can be done considering this as a guideline. The following is a list of some suggested possible future works that can benefit from this parallel implementation:

- This work uses context orders of length up to two (2). A parallel implementation and analysis of the same based on higher context orders should be considered.
- Parallel implementation, tests and analysis of others variants of PPM using the architectures define here.
- Increasing the number of cores (processing elements) and the effect of such on parallel PPMC and/or other variants of PPM.
- Using controlled environment (by imposing restrictions on memory usage etc.) testing parallel PPM schemes. This would also lead to efficient parallel PPM schemes design.
- Using C/C++ based software environment for parallel PPM schemes. C# code works at an average of five times slower than C/C++ code.
- For fast parallel processing using GPGPUs (General Purpose Graphics Processing Unit) along cores. Already a research [34] showing Arithmetic Coding implementation on this has been done. PPM can be considered just one step ahead.

# Appendix A

## Information theory behind compression

Understanding data compression must start with understanding information, because the theme of former is based on the latter. This appendix provides a brief introduction to information theory.

We constantly receive and send information; this information is in the form of verbal conversation, text, sound, and images. We also feel that information is an elusive nonmathematical quantity that cannot be precisely defined, captured, or measured [1]. Information is a stimulus that has meaning in some context for its receiver. Information in its most restricted technical sense is a message or collection of messages in an ordered sequence that consists of symbols. When information is entered into and stored in a computer, it is generally referred to as data. After processing (such as formatting and printing), output data can again be perceived as information.

Information theory forms the mathematical basis of both lossy and lossless compression. The theory of information is developed by Claude Shannon in the 1940s [22, 23]. This theory provides a statistical way to measure information. Information theory measures the quantities of all kinds of information in terms of bits (binary digit). Information theory is the foundation of many data compression techniques. The two most important terms in information theory are entropy and redundancy.

### A.1 Entropy

Shannon tried to develop means for measuring the amount of information stored in a symbol without considering the meaning of the information. He discovered the connection between the logarithm function and information, and showed that the information content (in bits) of a symbol with probability $p$ is $-\log 2\ p$ [3].

Shannon used the term entropy to encapsulate the measure of information. The term is used widely in physics to describe the amount of order or disorder in a system. In information theory, a system with a high degree of disorder is also one that contains a great deal of information. Entropy is a number which is small when there is a lot of order and large when there is a lot of disorder.

Mathematically:

The *entropy* is defined as

$$H = -\sum_{i=1}^{n} P_i \log_2 P_i$$

Here H is for entropy which equals to the negative sum of the products of symbol's $P_i \log_2 P_i$, where *P* is the symbol's probability and *i* is the current symbol.

***Example***:

The entropy of "data_compression" is?

In above the symbol probabilities are,

$P[d] = {}^{1}/_{16}$ $P[a] = {}^{2}/_{16}$ $P[t] = {}^{1}/_{16}$ $P[_] = {}^{1}/_{16}$

$P[c] = {}^{1}/_{16}$ $P[o] = {}^{2}/_{16}$ $P[m] = {}^{1}/_{16}$ $P[p] = {}^{1}/_{16}$

$P[r] = {}^{1}/_{16}$ $P[e] = {}^{1}/_{16}$ $P[s] = {}^{2}/_{16}$ $P[i] = {}^{1}/_{16}$

$P[n] = {}^{1}/_{16}$

Hence entropy H is:

$H[data_{compression}] =$

$$-\left(\frac{1}{16}\log_2\frac{1}{16} + \frac{2}{16}\log_2\frac{2}{16} + \frac{1}{16}\log_2\frac{1}{16} + \frac{1}{16}\log_2\frac{1}{16}\right.$$

$$+\frac{1}{16}\log_2\frac{1}{16} + \frac{2}{16}\log_2\frac{2}{16} + \frac{1}{16}\log_2\frac{1}{16} + \frac{1}{16}\log_2\frac{1}{16}$$

$$\left.+\frac{1}{16}\log_2\frac{1}{16} + \frac{1}{16}\log_2\frac{1}{16} + \frac{2}{16}\log_2\frac{2}{16} + \frac{1}{16}\log_2\frac{1}{16}\frac{1}{16} + \log_2\frac{1}{16}\right)$$

$H[data_compression] = 3.625\ bits/symbol.$

## A.2 Redundancy

Redundancy is another concept which has arisen from information theory. Redundancy is the opposite of information. Something that is redundant adds little, if any, information to a message. Redundancy in information theory is the number of bits used to transmit a message minus the number of bits of actual information in the message. Informally, it is the amount of wasted "space" used to transmit certain data. Data compression is a way to reduce or eliminate unwanted redundancy.

***Example***:

Identify redundancies in the following string that contains consecutive repeating characters:

“BAAAAAAAAAC”?

Here the redundancy is the 9 repeating symbols ‘A’ which can be replaced by a shorter string such as ($r_9$)A.

## A.3 Relation between entropy and redundancy

The two qualities “Entropy” and “Redundancy” has an inversely proportional relation with each another which is reflected by the fact that when the entropy is at its maximum, the redundancy is zero and the data cannot be compressed any further.

# Appendix B

## Prediction and Probability theory concepts

This appendix serves as a brief introduction to Prediction and Probability theory concepts. Both prediction and probability concepts are fundamentals to the understanding of PPM (prediction by partial matching) and Arithmetic coding data compression schemes.

### B.1 Prediction

A prediction is a statement about the way things will happen in the future. Predictions are the connecting links between prior knowledge and new information [33]. It's the interaction between these two processes that make readers (both human and machine) predict. Although all proficient readers can make predictions, some readers seem to predict more explicitly than others which is mainly concerned with their prior knowledge experience.

Prediction is closely related to uncertainty. We predict outcomes, events or actions that are confirmed or contradicted; the first is based on authentic information whereas in the latter case one can say that guaranteed information about the information is also impossible. The whole intuition of probability theory is based on predictions.

### B.2 Probability

Probability is a mathematical prediction about the likelihood of an event occurring. It simply is a measure of the frequency of outcomes (events) that is assigning "density" to each possible event within some interval. The word 'event' refers to the predicted outcome. These events or prediction outcomes can be categorized in the following four states:

- Certain
- Likely

- Unlikely
- Impossible

Probability is used to mean the chance that a particular event (or set of events) will occur expressed on a linear scale from 0 (impossibility) to 1 (certainty), also expressed as a percentage between 0 and 100%. The analysis of events governed by probability is called statistics.

## B.3 Number line (probability line) concept in probability

Probabilities are expressed on a number line having range from 0 to 1. This number line often referred to as probability line acts as a graphical representation for expressing probabilistic events. The figure below shows a probability number line:

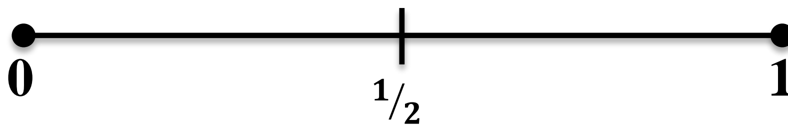



*Fig. Probability number line*

On this number line:

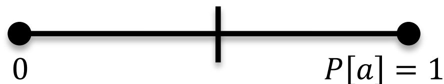
- Events that are impossible have a probability of 0.
- Events that are certain have a probability of 1.
- Events having even chances of occurrence have a probability of $^1/_2$.

Expressed another way:

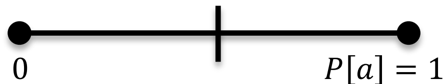
- A probability closer to 1 express higher frequency or occurrence.
- A probability closer to 0 express lower frequency or occurrence.
- A probability of 1 expresses maximum frequency or occurrence.
- A probability of 0 expresses no occurrence or frequency.

***Example*****:** The following shows occurrence of symbols 'a' and 'b' on the number line while reading the text "aabba".

- Read first symbol 'a' and assign it a probability of ${}^{1}/{}_{1} = 1$ as it is the first and only symbol occurred so far.

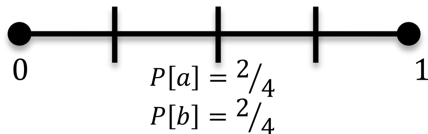



- Read second symbol 'a', probability assigned is ${}^{2}/{}_{2} = 1$ as frequency of 'a' is now two and only two symbols has occurred so far.

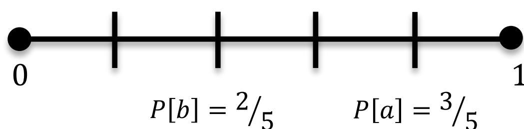



- Read third symbol 'b'. Now the total symbols are three, frequency of 'a' is two and frequency of 'b' is one. Hence new probabilities are $a = {}^{2}/{}_{3}$ $and$ $b = {}^{1}/{}_{3}$

- Read fourth symbol 'b', now the total is four, frequency of 'b' is two whereas frequency of 'a' is still two. Hence new probabilities are $a = {}^{2}/{}_{4}$ $and$ $b = {}^{2}/{}_{4}$




- Read fifth symbol 'a', new total is five and new probabilities are $a = {}^{3}/{}_{5}$ $and$ $b = {}^{2}/{}_{5}$



## References


[1] Salomon D, Motta G, Bryant DCON. Handbook of data compression: Springer-Verlag New York Inc; 2009.

[2] http://www.data-compression.info/Corpora/index.htm

[3] ftp://ftp.cpsc.ucalgary.ca/pub/projects/text.compression.corpus.

[4] http://corpus.canterbury.ac.nz/

[5] http://www.juergen-abel.info/

[6] http://sun.aei.polsl.pl/~sdeor/index.php?page=silesia

[7] Witten IH, Moffat A, Bell TC. Managing gigabytes: compressing and indexing documents and images: Morgan Kaufmann; 1999.

[8] Ian H. Witten, Radford M. Neal, and John G. Cleary. 1987. Arithmetic coding for data compression. Commun. ACM 30, 6 (June 1987), 520-540.

[9] Alistair Moffat, Radford M. Neal, and Ian H. Witten. 1998. Arithmetic coding revisited. ACM Trans. Inf. Syst. 16, 3 (July 1998), 256-294.

[10] Bell T, Cleary J, Witten I. Data compression using adaptive coding and partial string matching. IEEE Transactions on Communications. 1984;32(4):396-402.

[11] Moffat A. Implementing the PPM data compression scheme. Communications, IEEE Transactions on. 1990;38(11):1917-21.

[16] J. G. Cleary, W. J. Teahan, and I. H. Witten. 1995. Unbounded length contexts for PPM. In Proceedings of the Conference on Data Compression (DCC '95). IEEE Computer Society, Washington, DC, USA, 52-.

[17] Charles Bloom "Solving the Problems of Context Modeling",(March, 1998). http://www.cbloom.com

[18] G P Howard and Scott J Vitter. 1994. Design and Analysis of Fast Text Compression Based on Quasi-Arithmetic Coding. Technical Report. Duke University, Durham, NC, USA.

[19] Almasi, G.S. and A. Gottlieb. Highly Parallel Computing: Benjamin-Cummings publishers, Redwood City, CA.; 1989.

[20] Patterson, David A. and John L. Hennessy. Computer Organization and Design, Second Edition, Morgan Kaufmann Publishers.; 1998.

[21] A. Grama, G. Karypis, V. Kumar. Introduction to parallel computing: Addison-Wesley Longman Publishing Co., Inc.; 2002.

[22] Sergio De Agostino, A parallel decoding algorithm for LZ2 data compression, Parallel Computing, Volume 21, Issue 12, December 1995, Pages 1957-1961, ISSN 0167-8191, 10.1016/0167-8191(95)01030-0.

[23] Gilchrist J, editor. Parallel data compression with bzip2. 2004.

[24] Cinque L, Agostino S, Lombardi L, editors. Speeding up lossless image compression: Experimental results on a parallel machine. 2008.

[25] Martinovic G, Livada C, Zagar D. Analysis of parallelization effects on textual data compression. Proceedings of the 11th WSEAS international conference on Automation \&\#38; information; Iasi, Romania. 1863304: World Scientific and Engineering Academy and Society (WSEAS); 2010. p. 128-32.

[26] Franaszek P, Robinson J, Thomas J, editors. Parallel compression with cooperative dictionary construction. 1996: IEEE.

[27] Kitzman J, Fujiwara G. Parallel file compression. Technical Report 18.337 J, 2005.

[28] Gilchrist, J. and Cuhadar, A. (2008) ‘Parallel lossless data compression using the Burrows-Wheeler Transform’, Int. J. Web and Grid Services, Vol. 4, No. 1, pp.117–135.

[29] Schildt H. C# 4.0 The Complete Reference: McGraw-Hill Osborne Media; 2010.

[30] Watson K, Nagel C, Pedersen JH, Reid JD, Skinner M. Beginning Visual C# 2010: Wrox Press Ltd.; 2010.

[31] Shannon, Claude E. (1948), "A Mathematical Theory of Communication," *Bell System Technical Journal*, **27**:379–423 and 623–656, July and October.

[32] Shannon, Claude (1951) "Prediction and Entropy of Printed English," *Bell System Technical Journal*, **30**(1):50–64, January.

[33] Gillet, J. and C. Temple (1990). Understanding Reading Problems: Assessment and Instruction, Boston: Allyn and Bacon.

[34] Ana Balevic, Lars Rockstroh, Marek Wroblewski, and Sven Simon. 2008. Using Arithmetic Coding for Reduction of Resulting Simulation Data Size on Massively Parallel GPGPUs. In Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface, Alexey Lastovetsky, Tahar Kechadi, and Jack Dongarra (Eds.). Springer-Verlag, Berlin, Heidelberg, 295-302.