University of the Witwatersrand

Masters Dissertation

Model Propagation for High-Parallelism in Data Compression

Author:
Shaw Chian Lin

A Dissertation submitted in fulfillment of the requirements for the degree of Master of Science

in the

October 2023

Declaration

I, **Shaw Chian Lin**, declare that this Dissertation titled, "**Model Propagation for High-Parallelism in Data Compression**" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this dissertation is entirely my own work.
- I have acknowledged all main sources of help.

Signed: 

Date: 06/10/2023

UNIVERSITY OF THE WITWATERSRAND

Abstract

Engineering and the Built Environment
School of Electrical and Information Engineering

Master of Science

Model Propagation for High-Parallelism in Data Compression

by Shaw Chian Lin

Supervisor: Ling Cheng

Recent data compression research focuses on the parallelisation of existing algorithms (LZ77, BZIP2 etc.) by exploiting their inherent parallelism. Little work has been performed on parallelising highly sequential algorithms, whose slow compression speeds would benefit the most from parallelism. This dissertation presents a generalised parallelisation approach that can be potentially adopted by any compression algorithms, with model sequentiality in mind. The scheme presents a novel divide-and-conquer approach when dividing the data stream into smaller data blocks for parallelisation. The scheme, branching propagation, is implemented with prediction by partial matching (PPM), an algorithm of the statistical-modelling family known for their serial nature, which is shown to suffer from compression ratio increases when parallelised. A speedup of 5.2-7x has been achieved at 16 threads, with at most a 6.5% increase in size relative to serial performance, while the conventional approach showed up to a 7.5x speedup with an 8.0% increase. The branching propagation approach has been shown to offer better compression ratios over conventional approaches with increasing parallelism (a difference of 11% increase at 256 threads), albeit at slightly slower speeds. To quantify the speedup over ratio penalty, an alternate metric called speedup-to-ratio increase (SRI) is used. This shows that when serial dependency is maintained, branching propagation is superior in standard configurations, which offers substantial speed while minimising the compression ratio penalty relative to the speedup. However, at lower serial dependency, the conventional approach is generally preferable, with 9-16x speedup per 1% increase in compression ratio at maximal speed compared to branching propagation's 6-13x speedup per 1%.

Acknowledgements

Firstly, I would like to express utmost gratitude towards my supervisor, Ling Cheng, who never failed to support me in the hardest of times. His supervision and mentorship have also given me many great insights and motivations. If it was not for him, I would not have had the will to see this through.

Utmost gratitude to my friends, Jiyaad Lai and Talha Tayob, who always showed support and optimism for my work. When I was formulating concepts, they were often the first people for me to do a "does this make sense?" idea pass.

Contents

List of Figures

List of Tables

Abbreviations

ANS	**A**symmetic **N**umeral **S**ystem
BWT	**B**urrows-**W**heeler **T**ransform
CPU	**C**entral **P**rocessing **U**nits
GPU	**G**raphical **P**rocessing **U**nits
GPGPU	**G**eneral **P**urpose **G**raphical **P**rocessing **U**nits
LZ	**L**empel-**Z**iv
LZMA	**L**empel-**Z**iv **M**arkov-chain **A**lgorithm
LZSS	**L**empel-**Z**iv-**S**torer-**S**zymanski
LZW	**L**empel-**Z**iv-**W**elch
MTF	**M**ove-**T**o-**F**ront
PNG	**P**ortable **N**etwork **G**raphics
PPM	**P**rediction by **P**artial-**M**atching
RLE	**R**un-**L**ength-**E**ncoding
SRI	**S**peedup-to-**R**atio **I**ncrease
TIFF	**T**ag **I**mage **F**ile **F**ormat

Publications

Overlapping chapter	**Submitted journal papers directly related to this Dissertation**
[3]	**Shaw Chian Lin** and Ling Cheng, "Model Propagation for High-Parallelism in Data Compression", submitted to *IEEE Access* for review.

CHAPTER 1

Introduction

The use of data compression is near-ubiquitous in the present information age. Whether for storage or transfer purposes, various types of compression algorithms are employed to reduce said data storage and transfer requirements. A compression algorithm can be either lossy - where unnecessary information is removed and cannot be recovered upon decompression, or lossless - where all information is retained upon decompression. This dissertation concerns and assumes lossless algorithms unless otherwise stated. Most of today's algorithms are combinations and/or derivations of techniques developed before the 2000s. For example, the widely used ZIP file archive format and the PNG image format both use the DEFLATE compression algorithm[2] from 1996. DEFLATE itself uses Huffman codes [3] from 1952 and the LZ77 algorithm [4] from 1977. As computing resources become more powerful and network infrastructures more established, existing algorithms such as DEFLATE are more than good enough for typical applications.

Over the past decade, computational resources have become increasingly parallel and more widely available. The increasing popularity of general-purpose graphical processing units (GPGPUs) on a large scale also garnered the attention of parallelism in data compression, with a resurgence in data compression research in recent years. Emphasis has been on the parallelisation of previously developed algorithms, mostly for GPU-acceleration, some earlier work has been done for CPU-based parallelisation as well. However, not all compression algorithms can be treated equally in this endeavour. Algorithms with the best compression ratio performance, in that they are the best choice in terms of compressed file size, those (compression) ratio-focused are generally also the worst in terms of

compression speed. Recent research progress on parallelisation made the faster algorithms even faster, while those slower algorithms are generally ignored due to the ever-decreasing cost of digital storage and data transfer. Yet it is those ratio-focused compression algorithms that need parallel speedup the most, as they are the slowest algorithms and could benefit greatly from said speedup.

1.1 Problem Statement

The approach taken by conventional algorithms for parallelisation is simple, split the data source into a corresponding number of chunks and perform said compression algorithms on each block in parallel, independent of each other. This is the pitfall of ratio-focused algorithms, whose excellent ratio performance comes from their ability to "learn" the data and thus improve compression ratio over time as it serially processes the data. Those algorithms are mostly based on statistical modelling and perform best when sequentially processing data. As one might expect, parallelising a highly serial algorithm yields poor results. Splitting said data into smaller chunks effectively means less data for the statistical model(s) to learn and will often result in worse ratio performance. It is also likewise difficult to quantify said ratio penalty in relation to the speedup benefits.

1.2 Aim & Research Objectives

The aim of the research can be described with a primary research question:

Is it possible to parallelise highly serial statistical algorithms, while offering viable space-time trade-offs?

Since one can always parallelise any algorithm by the conventional approach of splitting data. The goal is a parallel approach for statistical algorithms which will perform better than the conventional approach. For evaluating performance, space-time trade-off refers to compression ratio versus speed, where favouring one would come at a cost of the other. Due to this trade-off, it is harder to quantify

one approach over the other. As mentioned before, this trade-off is also heavily influenced by an algorithm's serial dependency, in that algorithms which benefit the most from serial processing will also be most affected by parallelism. From our primary question, several objectives can be presented in sub-questions:

1. *Is there a viable alternative to the conventional approach of splitting data for parallel compression?*
2. *If so, how would this hypothetical approach be affected by an algorithm's serial dependency?*
3. *Finally, how do we quantify the parallel performance increase for comparison?*

1.3 Contribution

The contribution of this work presents itself in the form of responses to the sub-questions:

1. *Is there a viable alternative to the conventional approach of splitting data for parallel compression?*

   This dissertation presents a solution to this: branching propagation. While the solution is not outright better regarding the space-time trade-offs in all cases (an aspect relating to problem 2), the compression ratio alone is always better than the conventional approach. The branching propagation scheme is designed such that one can generalise the approach without being specific about the algorithm, as the scheme can be described without specifying the algorithm beforehand and likewise, be applied to various algorithms. The selected algorithm does, however, have to be implemented to conform to certain restrictions.

2. *If so, how would this hypothetical approach be affected by an algorithm's serial dependency?*

The serial dependency is first shown to be relative to the data source itself, along with the algorithm's capacity. For example, an algorithm with limited model capacity will perform worse on sources with long stretches of alternating patterns, as it will forget certain sequences that have not been seen in a long time. The greater the serial dependency of the algorithm, the better branching propagation performs when considering space-time trade-offs. On the contrary, low serial dependency will perform better with the conventional approach.

3. *Finally, how do we quantify the parallel performance increase?*

   To aid in answering the previous two questions. A metric is proposed to specifically quantify parallel (or any sort of modification) speedup relative to the increase in the compressed size of a serial algorithm. This metric, speedup-to-ratio increase (SRI), allows for easier comparison between two parallel approaches. One could compare the two by considering the increase in speed per increase in size. For example, the proposed scheme has an average SRI of around 7×10^2 for longer sources, corresponding to around 700% increase in speed per 1% increase in size, at 16 parallel threads.

1.4 Organization of Dissertation

The rest of the dissertation is organized as follows. Chapter 2 gives a preliminary on lossless data compression algorithms. Chapter 3 presents the proposed scheme in a paper format and details the answers to the aforementioned questions. Chapter 4 then concludes the paper with possibilities for future work.

1.5 Conclusion

The simplest approach to parallelising compression algorithms is to divide the data source into individual sources, each to be compressed separately. However, this becomes far less viable with increasing serial dependency, as will be shown later on in Chapter 3. With the problem described, this dissertation will then present a potential solution to the problem: branching propagation.

CHAPTER 2

Background and Literature

2.1 Introduction

This chapter provides the basis of preliminary information regarding common lossless data compression techniques. Emphasis will be placed on algorithms that became popular candidates in the recent parallelisation trend, as well as components in the work of this dissertation, followed by a review of the literature on said trend.

2.2 Background

Data compression algorithms can be divided into two main types: lossy compression and lossless compression. Our interest lies in lossless compression, which retains data correctness upon decompression, as such, lossless compression can be assumed for the remainder of this dissertation. In lossless compression, most algorithms can be categorised into sub-types. The first group are entropy codes, including Huffman codes [3], arithmetic codes [6] and asymmetric numeral systems (ANS) [7]. Entropy coders achieves compression by translating symbol probabilities into codewords, the three techniques mentioned are optimal to Shannon's entropy [5]. Complementing entropy codes, are statistical modelling techniques[8, 9, 29], which produces more accurate probability estimations and consequently better compression ratios. Third group are dictionary codes[4, 10–13], which achieves compression by mapping input sequences to shorter codeword sequences from a dictionary. The dictionary is constructed based on previously observed data. All dictionary coders are based on the initial Lempel-Ziv77 (LZ77) algorithm [4] and

generally have the LZ prefix. The remaining techniques are more diverse: the basic run-length encoding (RLE); transforms such as the Burrows-Wheeler transform (BWT) [14] and move-to-front (MTF) makes data easier to compress for other techniques. Practical compression algorithms generally combines the compression subtypes to attain better compression. Common examples includes the commonly used DEFLATE format [30] (used in ZIP archives) which combines LZ77 [4] and Huffman coding [3], and the BZIP2 algorithm , which was described in the original BWT paper [14] and uses BWT, RLE and Huffman coding [3].

2.2.1 Entropy Codes

The field of information theory began with Shannon's paper, A Mathematical Theory of Communication [5], from 1948. In the paper, the lossless data compression technique called Shannon coding became the precursor of an entire family of compression techniques known as entropy coders. An entropy coder performs mapping of symbols to codewords, with the optimal length of binary codewords being [5]

$$-\log_2 P \; bits \tag{2.1}$$

where P is the probability of a particular symbol occurring. Huffman codes [3], a type of prefix code like the Shannon codes, offered optimal (minimum-redundancy per-symbol) representation to achieve compression. Both methods generate codewords by constructing a binary tree with symbols as leaf nodes. The corresponding sequence of 0s and 1s corresponds to the left and right traversal from the root node to said symbol node. Huffman code [3] achieves optimal coding by constructing the binary tree from the bottom up, by repeatedly connecting two parent-less nodes with the lowest probability to a single parent node (which itself is parent-less). Huffman coding is still one of the most prominent techniques used in many compression algorithms today.

A flaw with Huffman codes is that codewords are always rounded to the nearest whole bit. If a symbol has a 0.9 chance to occur, it should only require roughly

0.15 bit to represent. However, Huffman code will map it to one bit. Two solutions to this are arithmetic coding [6] and the asymmetric numeral systems (ANS) [7]. Arithmetic coder achieves compression by encoding symbol probabilities in a single decimal number of arbitrary length (every symbol encoded adds to the length). This is achieved by repeatedly shrinking an interval (starting with [0, 1)) in proportion to the symbol probabilities. As an example, suppose we have the symbols a and b with corresponding probabilities 0.9 and 0.1. encoding the sequence ab will result in the interval steps of $[0, 1) \rightarrow [0, 0.9) \rightarrow [0.81, 0.9)$. ANS [7] is a much newer addition which encodes and decodes data in a first-in-last-out fashion. It also achieves optimal coding, but is much faster than arithmetic coders, at a speed comparable to Huffman coders.

2.2.2 Statistical Modelling

To attain a better compression ratio, entropy coders are often used in conjunction with statistic models. One algorithm of concern is the prediction by partial-matching algorithm (PPM), which uses a variable-order Markov model [8] to produce better probability estimates and consequently shorter codewords. The variable-order Markov model keeps track of symbol occurrences after a specific sequence. For example, in English text it is highly probable that the symbol that comes after "th" is a vowel, therefore those symbols will have a higher probability assigned to them by the model. Context mixing [9] is a technique which extends upon this idea, by utilising multiple models to account for various data patterns (e.g. 2-D matching for images), one can achieve even better compression ratios. Statistical modelling algorithms are, however, relatively slower and more memory-intensive.

2.2.3 Dictionary Algorithms

Alternate to entropy coders, is the Lempel-Ziv (LZ) family of dictionary coders. Instead of using an entropy coder, those algorithms achieve compression by replacing recurring sequences with shorter representations from a dictionary. The nature of the dictionary varies based on the algorithm. The original LZ77 keeps the last few kilobytes of data as its dictionary and replaces sequences with positions of occurrences within said "sliding window" of data. LZ78 [10] instead constructs a dictionary, which maps symbol sequences to codewords which are simply their position within the dictionary. Lempel-Ziv-Welch (LZW) [11] is a commonly used algorithm based on LZ78. Its primary difference is that it uses a pre-initialised dictionary with all possible symbols. Lempel-Ziv-Storer-Szymanski (LZSS) [12] is instead a derivation of LZ77, the primary difference being the ability to directly encode the sequence with the codeword may require more bits than the sequence itself. Lempel-Ziv-Markov chain algorithm (LZMA) [13] is the algorithm used in the 7Z file archive format. It is also based on LZ77, but is capable of much larger dictionary sizes (up to 4 GB), along with many other enhancements, including an entropy coder for the final encoding process.

2.2.4 Transforms

Transform algorithms do not compress the data. As the name suggests, they reversibly transform the data such that other algorithms can achieve better compression ratios with the transformed data. Move-to-front transform keeps a list of the possible symbols, it then outputs the position of the symbol on the list (as a symbol). When a symbol is observed, it is then moved to the front of the list. The idea is that more frequently occurring symbols will be found closer to the front of the list, and consequently increase the probability of output symbols (positions) closer to the front. Another transform algorithm, the Burrows-Wheeler transform (BWT) [14] transforms the data block-by-block. The block of symbols is first rotated (i.e. move the first symbol to the back and repeat) to produce

all possible arrangements. After which the arrangements are sorted into lexical order. The output sequence is formed by the last symbol of each arrangement, in the order of the sort. BWT is used in the BZIP2 algorithm [14], along with run-length encoding (RLE) and Huffman coding.

2.3 Related Work

The above summarises all of the commonly used compression algorithms. As one may have noticed, much of the initial work was done before the 21st century. Those techniques, entropy coders, statistical models, dictionary algorithms, and transforms, forms the basis of all of the algorithm we use today. Recent research (discussed in Section 2.3) has also been concerned with their parallelisation, mostly for the application of GPU-acceleration.

To avoid confusion, we will properly define a few terms which may be used interchangeably. Compression speed, compression rate and performance all refer to the throughput of data by an algorithm. Speedup/slowdown refers to the ratio between the speed of two algorithms. For compression ratio, the two commonly used measures are 1) *output size/input size* and 2) *input size/output size*. While the first metric generally presents a value from 0 to 1.0, the latter value is a value usually greater than 1. Smaller/larger refers to 1), while higher/lower respectively corresponds to 2). For simplicity, this dissertation will only make use of 1), as it shows a linear scale relative to the original input. Ratios mentioned by other literature will also be represented in 1). In most cases, a difference in compression ratio (from serial) is used, referred to as ratio penalty.

The first parallelisation of compression algorithm concerned the Lempel–Ziv–Storer–Szymanski (LZSS) algorithm [12] by Gonzalez Smith and Storer [32] in 1985 and Franaszek[33] in 1996. Their paper described a parallel hardware implementation of LZSS with complexity analysis, it does not, however present actual results other than a reference to the original LZSS [12]; [33] presents an "intermediate approach" to the parallel dictionary-modelling of the LZSS algorithm. It allows

for the joint construction of a dictionary by multiple processing threads, which the author has experimentally shown to have little ratio loss over the sequential variant. One other work done in CPU-based parallelisation is on BZIP2 in [34]. As mentioned before, BZIP2 is memory-less (between data blocks), therefore the only concern is the speedup. They have shown a near-linear speed-up proportional to the number of CPU cores.

The three works considered so far made use of parallelisation on multi-core CPUs as opposed to highly parallel GPUs, as the general-purpose application of early GPUs was not practical due to their poor performance. However, interest in parallelising compression algorithms saw a resurgence in the past decade with the ever-growing presence of big data and the growing applicability of GPGPUs.

A 2009 paper by Balevic [15] sets the first practical example of data compression on GPGPUs. The paper describes a novel method of parallelising Huffman code [3] on the GPU, where the codewords have their length (consequently position) reserved via parallel computations. In this design the main bulk of computation lies in the bit offset computation with a time complexity of O(log n) [15]. The speed improvement over conventional CPUs (of similar age) reaches up to 50x. A paper from 2011 by Cloud et al. [16] also experimented with implementing Huffman codes on the GPU. While the results are not as impressive as [15], with 2x the performance increase over multi-core CPUs and 5-8x the performance over single-core processors, it still demonstrates the viability of accelerating data compression with GPUs.

Following the work on GPU accelerated Huffman coding [15, 16], the majority of work had been on GPU acceleration of the LZSS algorithm [12]. Ozsoy et al presented the CULZSS algorithm with improving iterations [20–22]. The algorithm was implemented on CUDA [20] and displayed a speed-up (8-20x) over the serial version on the CPU and a slight speed-up (1.5x) over the multithreaded CPU implementation at a slight compression penalty. A pipeline version of CULZSS [21] was presented the following year with improved throughput and speed-up

over the parallel CPU implementation (2.2x). Ozsoy later presented the CULZSS-Bit algorithm [22] in 2014, which further parallelises the CULZSS algorithm by implementing bit-level parallelism. In the same year, a separate implementation, referred to as Parallel Matching LZSS (PMLZSS), was presented by Zhou et al. [27]. The performance improvements are similar to that of Ozsoy [21], with a speed-up of up to 16x compared to the serial implementation, but at a much lower compression ratio penalty (2%). In 2019 Stein et al. [26] presented an improved version of the CULZSS algorithm via stream parallelism. Additionally, they described a different method of finding the longest match with multi-GPU support. The result is not only better compression ratios over both [21, 22] and [27], but is also faster (albeit with better GPU hardware). It is worth noting that the scalability over multiple GPUs demonstrates the viability of GPU acceleration for data compression.

Work had also been done on other LZ-variants. Funasaka et al. [17] succeeded in implementing the LZW algorithm [11] used in TIFF images on the GPU. They achieved a 3x speed increase over CPU with no compression penalty while maintaining compatibility with the existing format. The widely used DEFLATE algorithm [2] for GPU acceleration is explored in [25]. Recently in 2019, Lu & Hua [28] described an algorithm called the G-match, which is based on Google's Snappy algorithm [35] (also of the LZ family). G-match presented a speed improvement of 7x over Snappy with a compression penalty of less than 0.1 and performed 1.4x faster than CULZSS with 22% better ratios.

Compared to the popular dictionary-based algorithm, little work has been done on GPU acceleration for other families of compression algorithms. In 2012 Patel et al.[23] implemented the BZIP2 algorithm, which is based on BWT. However, they noted a decrease in speed in all three stages of the BZIP2 algorithm (BWT, MTF followed by Huffman coding) compared to the CPU counterpart. Improvement of the BZIP2 algorithm was still possible, as demonstrated by Shastry et al. [24]. Unlike the implementation of [23], theirs yielded positive results, achieving speed-ups of up to 6.3x when the GPU is utilized in conjunction with the multicore CPU [24]. A less-known algorithm, the context tree weighting (CTW) method

[29], was implemented by Krishnan et al. [19] on the GPU. While it is comparable to other existing commercial algorithms when it comes to time-space complexity trade-offs, it does not present a compelling reason for GPU acceleration compared to the work done on BZIP2 [23, 24] and LZ-variants [17, 20–22, 26–28].

Although our scheme applies to the GPGPU architecture, this dissertation only evaluates the performance on CPU. It is worth noting that the speedup benefits are not directly comparable between ours and existing work. While existing literature considers the inherent parallelisation within an algorithm, our approach instead considers the scenario of running multiple instances of such algorithms (serial or parallel) in parallel. The generalised nature of our scheme is to not only enable parallelisation but also offer an alternative parallelisation approach, which can potentially be adopted by and benefit said existing work. Furthermore, no work has been done on the parallelisation of our example algorithm (PPM [8]), which is of the statistical model sub-type.

2.4 Conclusion

In this chapter, common compression algorithms were first described, followed by a review of the literature concerning the parallelisation of said algorithm. A re-emphasis that the work presented in this dissertation is unlike those of existing literature in that this work considers the application of running an algorithm in parallel, as opposed to parallelising the algorithm internally.

CHAPTER 3

Model Branching Propagation

The content of this chapter is adapted from the paper "Model Propagation for High-Parallelism in Data Compression", submitted to IEEE Access for review.

3.1 Introduction

With the application of general-purpose graphical processing units (GPGPUs) in various fields such as scientific simulations, data sciences and machine learning. Data compression is no exception, recent research has been focused on parallel variants of existing data lossless compression algorithms [15–28]. However, not all types of compression algorithm have their parallel nature explored. Most lossless compressors, [4, 8–13, 29] are online/adaptive, in that they develop a model to better compress the data based solely on previously observed information. Algorithms which favour compression ratio over compression speed [8, 9] are especially exploitative of processing serial data for context modelling. The downside, however, is that they are less popular due to their relatively slow compression speed. With the rise of the information age, the ever-decreasing cost of both storage and bandwidth further reduced the need for better compression ratios; when considering the slight improvement in compression ratio with a large decrease in speed over other types of algorithms, (compression) ratio-focused algorithms become the lesser choice. That is not to say that compression ratios no longer matter. More so, serial algorithms will prove to be more valuable if their speed can be greatly increased using modern computing resources. This begs the question:

can modelling-based compression algorithms, whose performances are highly dependent on serial processing, be modified to benefit from parallel speedup while minimizing compression ratio penalties?

Many parallel implementations of popular algorithms [2, 8, 13, 14] take the simple approach of running independent models on individual data blocks. Memoryless algorithms such as Huffman codes[3] and BWT[14] are capable of producing exact compressed outputs regardless of the degree of parallelism [15, 23, 24, 31]. However, algorithms that maintain an online/adaptive model, including all of the dictionary and statistical algorithms, will observe a penalty to compression ratio when configured for parallelisation. Thus, we present a generalised method, termed *branching propagation*, which takes an approach that exists in between serial and parallel compression. As with the base case, the method assumes the segmentation of the data source into smaller blocks of data. More specifically, the scheme is designed for algorithms that benefit from inter-block persistent memories and model data correlations/repetitions to achieve compression. Our contribution can be summarised as follows:

- We present a generic parallelisation scheme that can be applied to various types of serial model-based compression algorithms.
- We show significant speedup with an example serial algorithm, prediction by partial-matching (PPM) [8], with a ratio penalty much less than the aforementioned simple approach.
- Additionally, we will show with said example that the greater the serial dependency, the greater the compression ratio penalty.
- We show that our scheme can be future-scalable, a concept which will be discussed in Section 3.4.7.

We will first discuss related work in Section 2.3 The details of the branching propagation scheme and a practical implementation will be discussed in Section 3.2 and 3.3 respectively. Followed by a discussion on experimental results and

possible improvements/investigations in Sections 3.4 and 3.5, and concluding with Section 3.6.

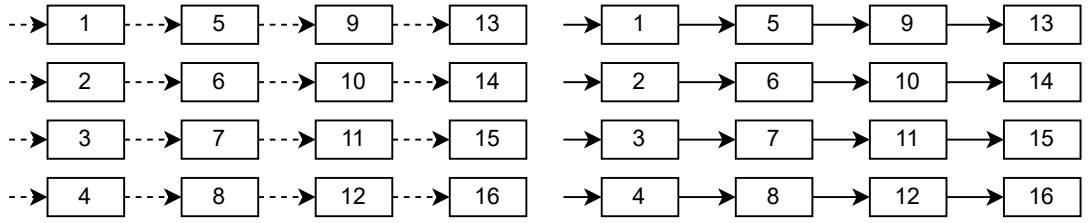

FIGURE 3.1: m-independent sources (left) and k-independent sources (right). Dashed lines indicate a thread's transition from one data block to another. While solid lines indicate thread *and* model transition. The block numbers are in order of appearance in the source.

3.2 Branching Propagation

Consider the conventional approach to parallelisation, the data stream of length n is divided into m data blocks . Each block is then processed serially by one of the K available processing threads of a machine, effectively achieving a $O(K)$ speedup in an ideal scenario. The flaw in this approach is the lack of model inter-dependencies, as most compression algorithms are online models in that they are updated as data is being processed. Thus, the algorithm splits the sources into m number of independent sources, that is, a m-independent algorithm. An alternate solution to this is using a common model per thread, in this case, we have K-independent sources. The remaining issue of this approach is that as K increases, the penalty to compression ratio also increases. We will refer to this method as the naive approach or the K-independent approach (Fig. 3.1). In a highly parallel system, K-independent approaches will become closer to m-independent approaches, as such, m-independent sources approaches are upper bounds of K-independent approaches both in compression ratio and speed.

Our proposed scheme, the branching propagation method (Fig. 3.2), offers a compromise between the serial end of the spectrum, with complete model inter-dependency, and the fully parallel end, where $K = m$ is a large number much less than n (effectively m-independent). The method takes a divide-and-conquer

approach to the assignments of data blocks to processors (Fig. 3.3). It consists of two main phases: the propagation phase and a saturation phase, the basics of which will be described below, followed by an overview of implementation.

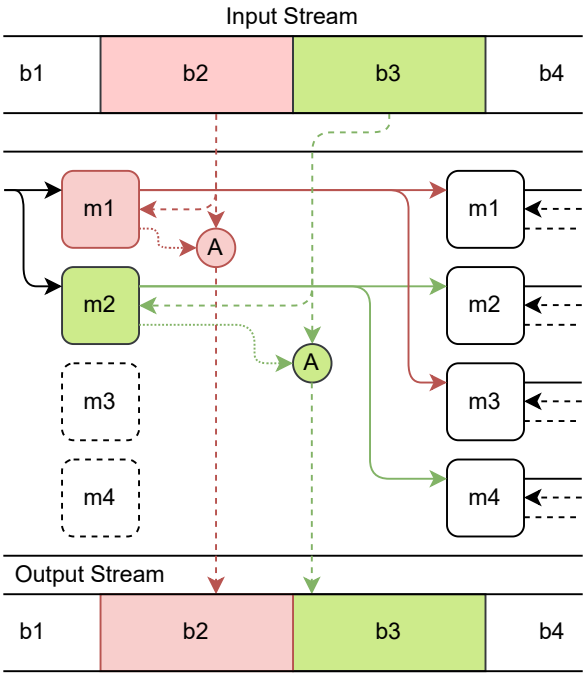

FIGURE 3.2: A single iteration of the branching propagation scheme. Note that the models' memory is preallocated. This current iteration is at propagation with $k = 2$, $K = 4$.

3.2.1 Propagation Phase

The scheme begins with the propagation phase. Given the source S of length n is divided into m data blocks $b_1, b_2, b_3, ...b_m$, K processing threads $t_1, t_2, ..., t_K$ are available, with a corresponding model set $M_1, M_2, ..., M_K$. Let k be the number of active threads, i.e. only threads $t_1, t_2, ..., t_k$ will be running at a given iteration. At the beginning we have $k = 1$ active thread (t_1), processing the 1^{st} data block b_1 using m_1. After b_1 is done, m_1 is copied into m_2, the number of active threads increases to $k = 2$, the active threads process the next two corresponding blocks b_2 and b_3. After which m_1 and m_2 are copied into m_3 and m_4 respectively, k is doubled to 4, and the next four data blocks are processed. This doubling process is repeated until all of the threads are active ($k = K$), upon which the scheme

will enter the saturation phase. Note that the block assignments are in-order and predetermined.

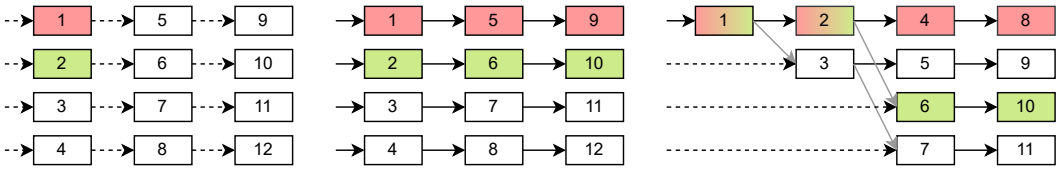

FIGURE 3.3: Inter-block correlation of models for m-independent (left), K-independent (middle) and branching propagation (right). In this example K = 4.

Consider Fig. 3.3, compared to the m-independent sources or the K-independent sources, we see greater model correlation in our branching approach. This is especially important with online compressors, where a model begins with zero knowledge of the source. Thus, more time should be spent learning the initial blocks of data before distributing them, as is the case with the branching propagation. While the difference of branching versus naive may seem minimal in Fig. 3.3, for large values of K, the performance of k-independent source will tend towards that of a m-independent source approach. This makes the branching approach more suitable for environments that offer a large number of logical processors, such as GPGPUs.

The branching factor β is defined as the additional number of processing threads/models to be activated in the next iteration i.e. $k_{t+1} = k_t + \beta$. In the standard case $\beta = k$, however, β can be set to any integer value ≥ 0 or an integer multiple of k. Setting β to a constant corresponds to a linear model propagation and a polynomial speedup w.r.t. K. $\beta = 0$ corresponds to a serial compression algorithm. $k_{initial}$ can also affect the overall scheme, setting $k_{initial} = K$ corresponds to a typical K-independent sources scheme.

3.2.2 Saturation Phase

In the saturation phase, the scheme behaves like a standard K-independent sources algorithm, like the propagation phase the block assignments are in-order and predetermined. Depending on the block size n/m and total number of processing

threads K. A compressor using branching propagation might not even reach the saturation phase before it is done with the data stream.

In the case of saturation, consider the application of the branching techniques used in the propagation phase. While there are no benefits to setting K beyond the number of logical processors available on a machine, one can still consider replacing one model with another. The worst-performing (ratio-wise) models can be combined or replaced with the top-performing models without exceeding K.

3.2.3 Complexity Analysis

The time complexity of the resulting algorithm can be estimated based on the proportion of propagation and saturation phases. For cases of propagation, we have a logarithmic time complexity $O(\log_2 m_{prop})$, based on our divide-and-conquer strategy. For the saturation phase, the time complexity is $O(m_{sat}/K)$. Where m_{prop} and m_{sat} are the block count within the propagation phase and saturation phase respectively. Correspondingly, the ideal speedup is $O(m_{prop}/\log_2 m_{prop})$ and $O(K)$, note that $m_{prop} = 2K - 1$ for sufficiently large $m \geq m_{prop}$. Before considering Amdahl's law:

$$\frac{1}{1 - p + p/s} \tag{3.1}$$

where $1 - p$ is the serial portion and p/s is the parallel portion. Even with an ideal (fully parallelisable) scenario, the logarithmic time of the propagation phase will contribute to a less-than-ideal speedup. Furthermore, propagation will have a higher associated coefficient due to the additional time required for model copying. We can approximate the overall speedup as

$$Speedup \approx \frac{m}{S + (1 + P)(\log_2 m_{prop} + 1 - 1) + (m_{sat} + K)/K} \tag{3.2}$$

where P is the time required in model copy relative to the average time required to process a data block, and S is the relative time of serial operations. While S and

$(m_{sat}+K)/K$ are correspondingly congruent to the serial and parallel denominator terms of Amdahl's law, the central log term is introduced by the proposed scheme.

3.3 Implementation

The scheme is implemented using the C++ language. The algorithm of interest is from the statistical modelling family called prediction by partial matching (PPM) [8]. A brief overview of the algorithm is given, along with high-level implementation.

3.3.1 Preliminary

Given a source S of arbitrary length n and a sequence of symbols $s_1, s_2, s_3, \ldots, s_n$ from an alphabet X with N unique symbols $x_1, x_2, x_3, \ldots, x_N$. A model is required to model the source and achieve compression. The requirements of the "black-box" model are generalised as such:

1. The model must provide a probability mass function (PMF) estimate of the next unseen symbol s_i for the entropy coder.
2. The model does not know what the next symbol is until it has been encoded.
3. If $P(s_i = x_j) = 0$ for any x_j, then the probability of an additional symbol known as the escape symbol [8] must be provided and greater than 0.
4. The model can be informed that the symbol s_i has "escaped"(i.e. $P(s_i = x_j) = 0$ [8]). Note that it does not know what s_i is. It then needs to provide an alternative PMF. This process will repeat until a distribution where $P(s_i = x_j) > 0$ is provided by the model.

In the case of PPM, the above model is a variable-order Markov tree [8]. The technique uses arithmetic coding [6] as its entropy coder. Our arithmetic coder

is based on the implementation described in [36]. The coder will be encoding the probability near-optimal to Shannon's entropy [5]

$$H(X) = -\sum_{j=1}^{N} P(x_j) \log_2 P(x_j) \tag{3.3}$$

where $P(x_j)$ denote estimated probability of all of the symbols of X. Note that (3.3) applies to an independent and identically distributed (i.i.d) source with an alphabet of fixed distribution. In algorithms such as PPM[8], the model does not present an i.i.d. source. We can instead approximate the information density. Given by [5] that the information of a single symbol is $-\log_2 P$ bits. Taking the average over all symbols $s_1, s_2, s_3, \ldots, s_n$, we have

$$-\frac{1}{n}\sum_{i=1}^{n} \log_2 P(s_i) \tag{3.4}$$

where $P(s_i)$ is the taken probability from the model, i.e. $P(s_i) = P(s_i = x_j)$ if $s_i = x_j$. Note that escape symbols are additionally generated for the sole purpose of encoding/decoding in [8] $P(s_i = x_j)$ is based on previously observed symbols, the degree of which (i.e. how many previous symbols are considered) is dependent on the model itself. While there are also errors from using practical entropy coders, (3.4) is useful in fast approximation, bypassing the computation cost from arithmetic coders.

3.3.2 Memory Structure

The core aspect of the scheme is model duplication. The instantaneous throughput of the algorithm becomes zero during actual propagation when models are being copied. A copy-efficient structure is thus required for the models in use. Two important properties must be considered: contiguity and dynamic.

Contiguous/sequential memory allocation was observed to be far more copy-efficient compared to random-access with dynamic memory allocation. Therefore we need to assign contiguous memory blocks for each model. This effectively reduces model

copying to standard array copies. On the other hand, dynamic allocation is preferred due to the dynamic growth and memory requirement of many models as they observe more and more data. A high-entropy source often requires more model memory compared to a low-entropy source. Pre-allocating memory for contexts that may never occur is next-to-wasteful in a memory-limited environment. A context is a set of prediction-related data that corresponds to a set of previously observed symbols, usually limited to the last few symbols in a variable-order Markov model. For example, given a+ prior sequence "abc", we have the corresponding contexts, "", "c", "bc" and "abc", for order-0, order-1, order-2 and order-3.

The proposed structure is a contiguous array with its internal dynamic allocation. A maximal number of contexts is designated for the array, which behaves like a smaller memory with its allocation scheme. The allocation is sequential, with its index counter for allocating new context data structures. When the memory is full, the array undergoes garbage collection. This is achieved by removing all memory references to the contexts and resetting the index counter.

This particular implementation of PPM is based on a trie-structure [37] up to the 5^{th} order, thus, we can simplify the reference removal. The root node, corresponding to an order-0 Markov model, is preserved, while its references to any child nodes are removed. Consequently, the rest of the trie will not be referenced as resetting the index treats all preexisting entries as empty for reallocation. A more complete memory allocation system may be better, for example, if we wish to keep both order-0 and order-1 (root and the first layer of the trie). The issue is that it would in turn result in much greater overhead in memory allocation and deallocation (note that the current approach used does not require deallocation).

With the memory structure defined, the propagation phase can be implemented efficiently. Given t maximum number of threads, K blocks of the memory blocks will be preallocated for our models M. Starting with $k = 1$, the stage alternates between model updating/compression and model copying. The k active threads will perform compression on the next k data blocks. When all threads are done,

they proceed to copy their models into the next k available memory blocks. Finally, we double k (unless $k = t$) and start over with compression of the next k data blocks.

3.3.3 Decompression & Metadata

As is the case with many online compressors, the propagation scheme is designed for symmetrical algorithms (in this case it refers to similar compression and decompression algorithms, not speed). It is a requirement that the decompression models can diverge at the same time the compression models do. This way the decompression threads will have matching compression/decompression model pairs and corresponding data blocks. This requires additional information to distinguish between the individual compressed blocks. While it is easy to determine the corresponding models of the data blocks, the nature of data compression means that the blocks have varying lengths. The simple solution to this is to write the compressed length at the beginning of each data block. The number of bits per block length can be predetermined (and written to the file header) based on the uncompressed block length, which the compressed length should never exceed.

3.4 Results & Discussion

The primary testing setup uses a Ryzen 7 5800X, with 8 physical cores and 16 logical cores, the total memory on the machine is 32 GB. Consequently, we assign 1.1 GB (2^{22} contexts) for each model's memory block. The effects of varying the maximum number of threads K and number of data blocks m, will be investigated. The branching scheme is compared to the naive/K-independent approach to determine its effectiveness, the serial algorithm is used as the control. The same PPM algorithm based on [37] will be used across the schemes. This is to maintain fairness, along with the fact that performances of the PPM algorithm [8] vary slightly by implementation. For comparison against the serial performance, some of the results are also shown as normalised values, i.e. measured relative to

serial performance, and are displayed on the right-vertical axes of relevant figures. The importance of normalised results will be discussed further in this section.

The test data used is the *enwik8* and *enwik9* dataset from the *Large Text Compression Benchmark* [1], which corresponds to the first 10^8 and 10^9 bytes of the English Wikipedia. It is the largest text compression dataset available [38]. The main concern of the branching propagation is speed-up and size increase, and not the performance of the algorithms themselves. Furthermore, we will see in the following results that parameters such as the length of the source stream n, maximum thread count t and block size n/m. Data samples are taken after each iteration described in Fig. 3.2.

3.4.1 Transient Performance

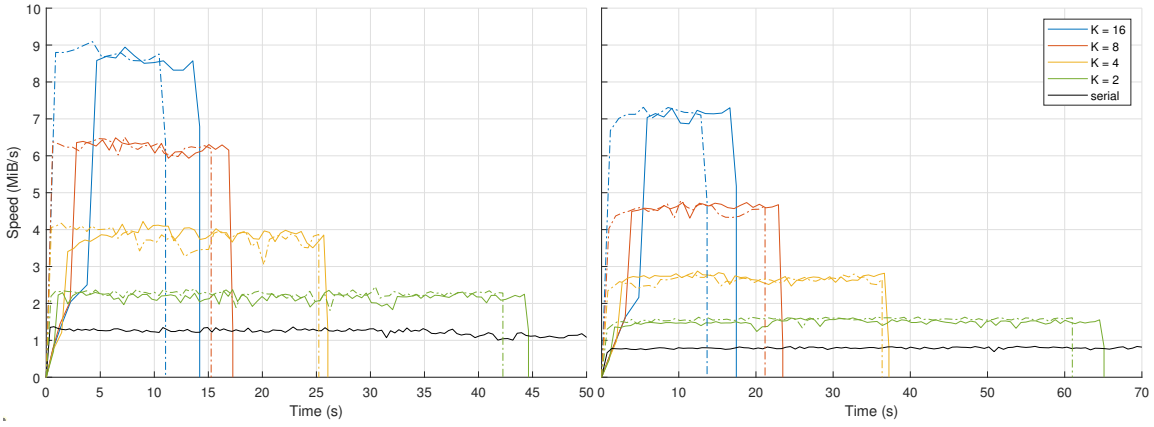

FIGURE 3.4: Compression (left) and decompression (right) speed over time of branching propagation with varying maximum number of threads. Dashed lines represent the naive performance for comparison. The intersections on the x-axis mark the total time taken for compression/decompression. The (omitted) serial compression/decompression finished at 78.5 and 120.0 seconds respectively. Block size is fixed at 500KB.

The characteristics of the scheme can be seen with compression/decompression speed over time (Fig. 3.4). Specifically, the raw data rates are measured, i.e. compression input speed and decompression output speed. The propagation method takes time to reach maximum speed, which is in line with the branching characteristic. Upon saturation, the performance behaves like a naive approach. The slightly slower decompression is expected since the PPM decoder needs the same

state calculation as the encoder in addition to the decoding calculations [8]. Decompression runs at 70-80% of the compression speed. Due to the near-linear proportionality, the proceeding results will only show compression speed to avoid cluttering the plots.

3.4.2 Block Size

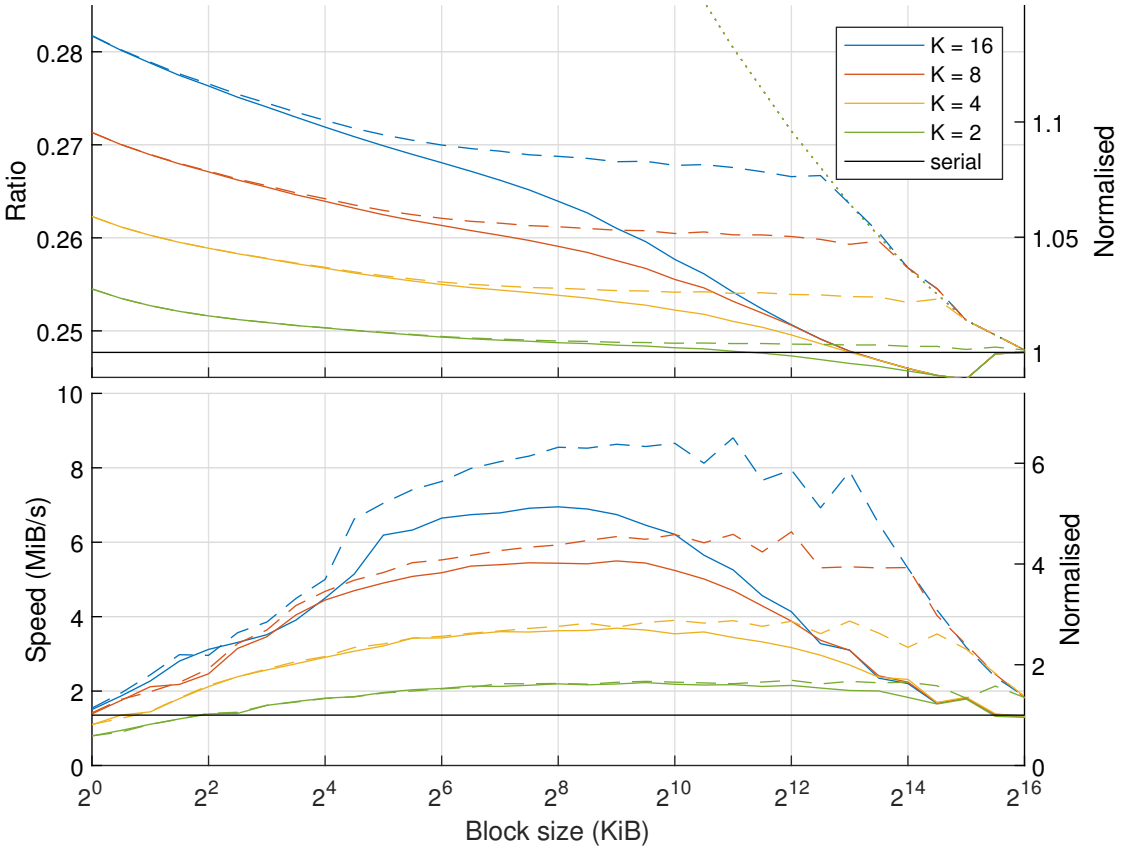

FIGURE 3.5: Effects of block size/propagation intervals on compression ratio (top) and speed (bottom) of *enwik8*, with varying maximum thread count. The branching method is represented by solid lines and naive is by dashed lines. The dotted line in the top plot corresponds to a m-independent sources approach, marking the maximum ratio penalty of a completely parallel scheme.

The next set of results concerns the size of the data blocks n/m. This is a crucial aspect as it affects the propagation rate and consequently how fast the scheme reaches saturation. A higher propagation rate is desirable for faster speed, but in contrast, smaller blocks would mean less data for the models to learn between propagation.

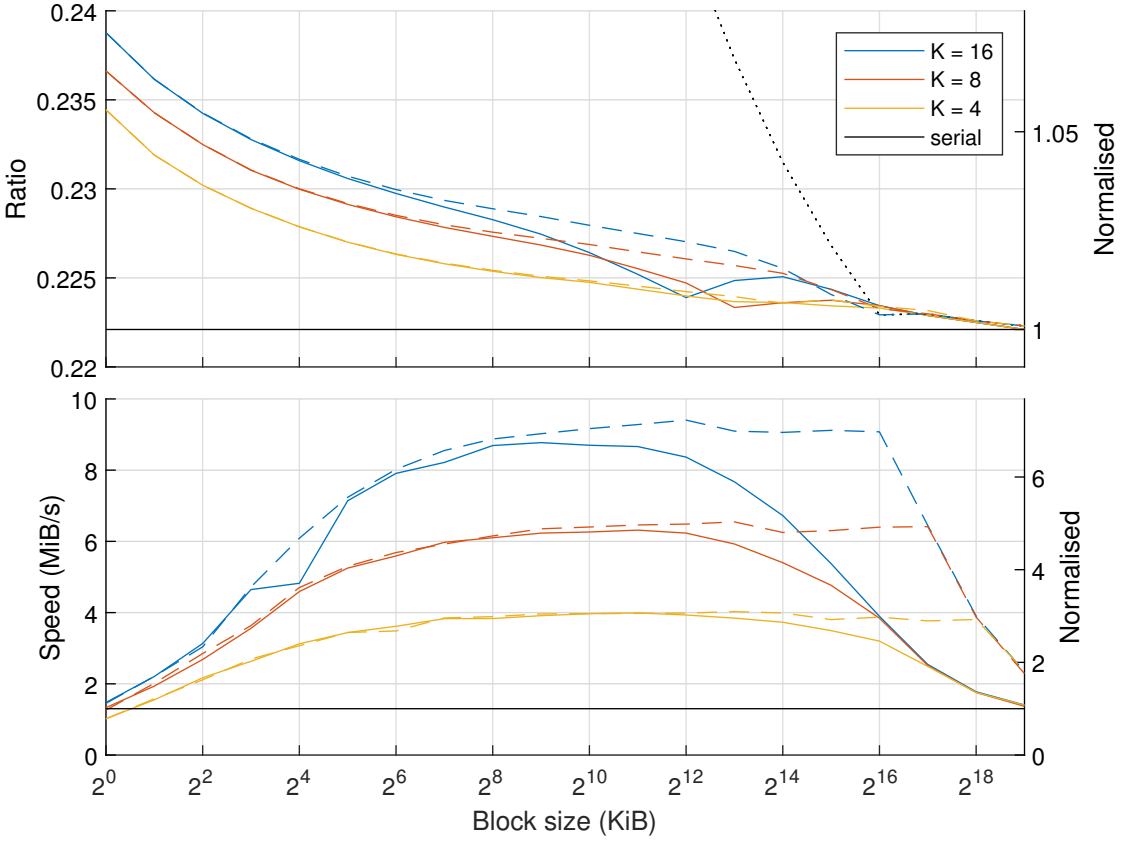

FIGURE 3.6: Effects of block size/propagation intervals on compression ratio (top) and speed (bottom) for *enwik9*, with varying maximum thread count. The branching method is represented by solid lines and naive is by dashed lines. The dotted line in the top-left plot corresponds to a m-independent sources approach, marking the maximum ratio penalty of a completely parallel scheme.

Fig.3.5 shows the effects of block size on the compression ratio and average speed for *enwik8*. The first observation is the detrimental effect m-independent sources have on the compression ratio. As our machines do not have m processor cores, the speed of m-independent is omitted. We can expect it to follow a similar trend shown with compression ratio, where it behaves like an upper bound in larger block sizes. Further on, the compression ratio improves with increasing block length, from being more parallel-like to being more serial-like. Speed performance is poor when the block sizes are either too small or too big. The expected reason for small block sizes hindering compression speed is memory I/O and multi-threading overheads. Therefore, the relevant region of interest is the right side of the plot (the point of maximum speed to the point of serial performance). Towards the right end, slower compression speed (along with better ratios) was expected for large

block sizes, reducing the number of data blocks for the scheme to reach saturation and making the scheme converge to a serial performance. The proposed scheme performed as expected, with a slower, yet scaling speed. The compression ratio also follows a similar pattern, as its performance tends towards that of the naive propagation with decreasing block size (increasing propagation rate).

The ratio difference is not as distinct for *enwik9* (Fig. 3.6), while the speed performance follows a similar pattern of *enwik8*. This alone suggests that our scheme is not more beneficial for larger input data, which contradicts the purpose of using branching propagation over the K-independent approach. A counterargument is that at best branching propagation can still achieve results similar to K-independent schemes. In Section 3.4.5, we will provide a metric that takes both speed and ratio into account, which will allow for a better comparison in cases of similar performance such as this.

3.4.3 Block Ratio

One observation is the oscillatory pattern in speed, consider the plot of the same data against block count or block ratio m (Fig. 3.7 This is a point of consideration, especially with low block ratios ($m < 5K$, 2 MiB or 2^{11} KiB for *enwik8*). Since the compression ratios follow a smooth trend, it is ideal to maintain the highest compression speed possible. The oscillatory pattern is due to the nature of the propagation scheme, the slowdown is caused by idle threads during the compression of the last batch of data blocks. The relationship between the block ratio m and the maximum number of threads K is

$$m = iK - 1 \tag{3.5}$$

for $i \in \mathbb{N}$ the scheme will attain local maximum compression speed, for other $i \in \mathbb{R}$ there will be idling threads when processing the last batch of data blocks. A solution to this is to modify the scheme to evenly assign the last batch of data

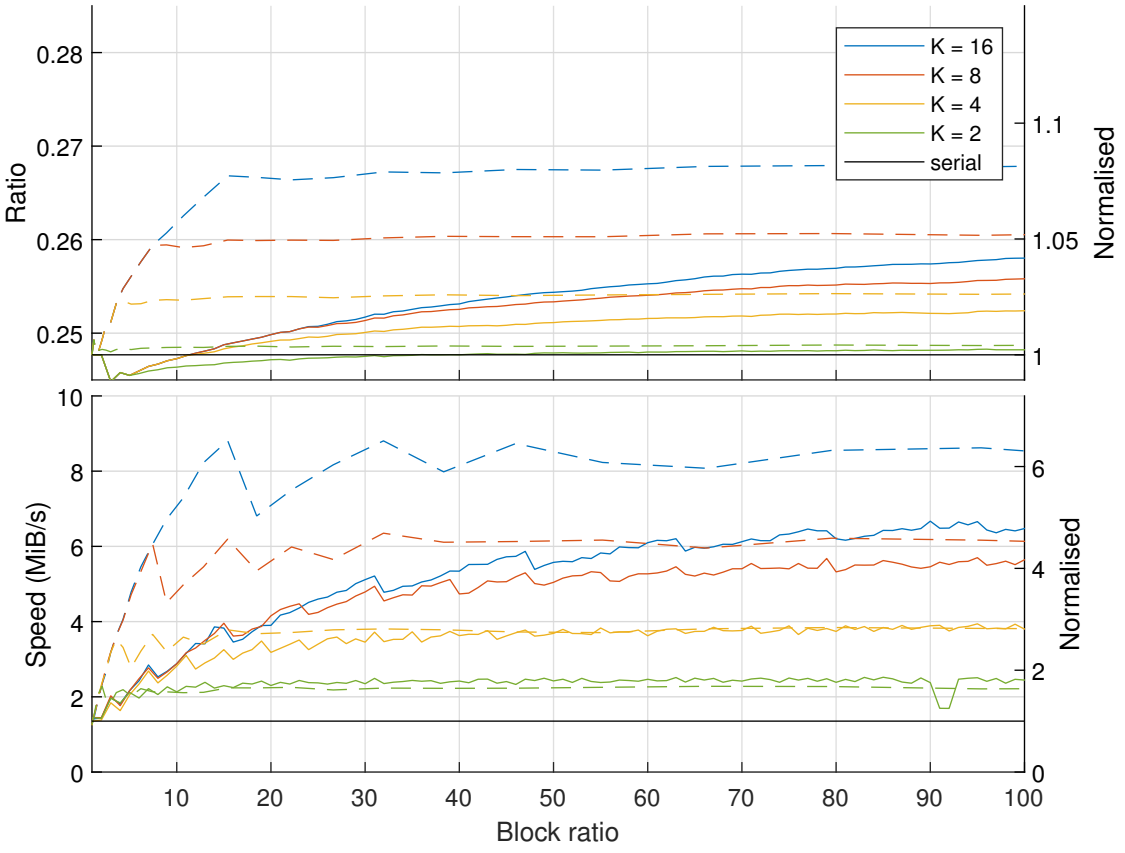

FIGURE 3.7: Effects of block ratio on compression ratio (top) and speed (bottom) of *enwik8*, with varying maximum thread count. The branching method is represented by solid lines and naive is represented by dashed lines. The right axes show the relative performances to the serial version of the algorithm.

with a different block length such that all of the threads are given the same length of data to process (i.e. n_{left}/k)

The value of i in (3.5) can be used to differentiate the scheme into different types. For $i \leq 2$, k never reaches the saturation phase, this will be referred to as *propagation-pure*. For $2 < i < 4$, where more data blocks are processed during propagation than saturation, we have propagation-dominant. Where $i > 4$ and most data blocks are processed in saturation, it is *saturation-dominant*. In between, where $i = 4$, the scheme is *balanced* and an equal amount of data blocks processed in the propagation and saturation phase.

saturation-dominant is chosen when higher speed is preferred, while *propagation-pure* or *propagation-dominant* favors better compression ratio. With a *balanced* configuration, we can achieve roughly 80% of the maximum speed, while having

TABLE 3.1: Branching propagation at maximum speed compared to a *balanced* configuration on *enwik8*.

K	Speedup			Ratio Increase		
	Max	Balanced	Max/Bal.	Max	Balanced	Max/Bal.
2	1.67	1.48	0.88	0.002	-	-
4	2.74	2.19	0.80	0.018	0.004	0.22
8	4.06	3.17	0.78	0.040	0.017	0.43
16	5.13	4.17	0.81	0.066	0.034	0.52

TABLE 3.2: Branching propagation at maximum speed compared to a *balanced* configuration on *enwik9*.

K	Speedup			Ratio Increase		
	Max	Balanced	Max/Bal.	Max	Balanced	Max/Bal.
4	3.06	2.46	0.80	0.010	0.005	0.50
8	4.79	3.66	0.76	0.015	0.007	0.47
16	6.65	5.17	0.78	0.024	0.013	0.54

a ratio increase that is at most 55% of the increase from maximum speed(Table 3.1 & 3.2). Note that the comparison is between different branching propagation configurations, not against the K-independent approach.

From 3.1 we can validate (3.2) described in Section 3.2.3. By assuming a *balanced* configuration, (3.2) can be simplified with (3.5) to

$$Speedup \approx \frac{m}{S + (1 + P)(\log_2 K) + (m - K + 1)/K} \tag{3.6}$$

Using (3.6), we can find constant values of S and P/m (since P is proportional to m) that exist for all values of the *balanced* configuration. 3.3 shows that at $S = 0.5$, P/m remains mostly constant.

3.4.4 Model Resets

One may have noticed from Fig. 3.5 that at one point, branching propagation performed better than the serial implementation. To explain this phenomenon,

TABLE 3.3: Values of P/m for fixed S.

S	enwik8				enwik9		
	K=2	K=4	K=8	K=16	K=4	K=8	K=16
0.4	0.04714	0.04833	0.03632	0.03058	0.02326	0.02225	0.01898
0.5	0.03286	0.04500	0.03526	0.03019	0.01999	0.02141	0.01858
0.6	0.01857	0.04166	0.03419	0.02979	0.01667	0.02009	0.01819

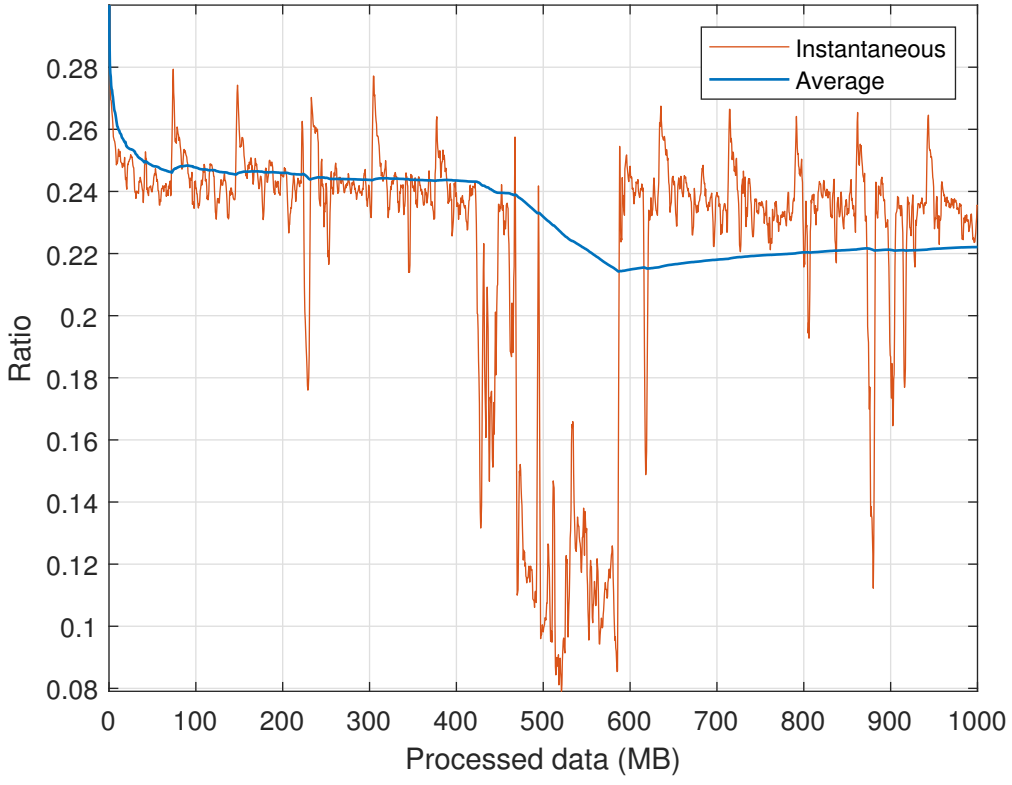

FIGURE 3.8: Serial compression ratio over the entire *enwik9* data. The drop at 400-600MB position is due to highly repetitive sequences [1]. The periodic spikes roughly every 74MB correspond to points where the model reset, as described in Section 3.3.2.

consider the block count or block ratio m. Fig. 3.7, plots the data of Fig. 3.5 but with finer steps of m as the x-axis. From Fig. 3.7, the best compression ratio is at $m = 3$. When Fig. 3.8 is also taken into account, we observe a flaw in our adaptation of the PPM algorithm to suit the required memory structure. The periodic spike increase in compression ratio corresponds to the model reset described in Section 3.3.2. At $m = 3$ there is no model reset, which corresponds to m_1 processing 33MB of data, branching into m_1 and m_2, each of which will process the remaining two blocks of 33MB. The total data processed by each model is 66MB, which is below the observed model reset threshold of 74 MB. As the propagation interval and thread counts increase, the decreasing model correlation overwhelms the ratio improvement of not having a model reset. Most curiously, is that even with the model reset, which removes all but the 0-order context (similar to a simple frequency counter), branching propagation still offers a significantly better compression ratio over a K-independent approach, even while losing a majority of the memory from the previous data.

Fig. 3.6

3.4.5 Speed-to-Ratio Increase

When comparing Fig. 3.5 and Fig. 3.6, we see that the benefit of branching over naive becomes less with larger sources. This is expected, since an online model's *average* compression ratio averages out with larger source lengths (and consequently block lengths), as seen in Fig. 3.8. The benchmark is reiterated for the first 100MB, 200MB, 300MB, ..., 1000 MB of the *enwik9* dataset (recall that *enwik8* corresponds to the first 100MB) and the results shown in 3.9. We observe that an increase in source length corresponds to more model resets. This reduces the models' serial dependency on the source, where there is insufficient memory to hold older information. As the serial dependency decreases, the ratio differences between branching and naive also decrease (Fig. 3.9). It is important to consider the relation between speed increase and size increase since the main goal is parallelisation with minimal penalty. To unify our metric of speed increase and

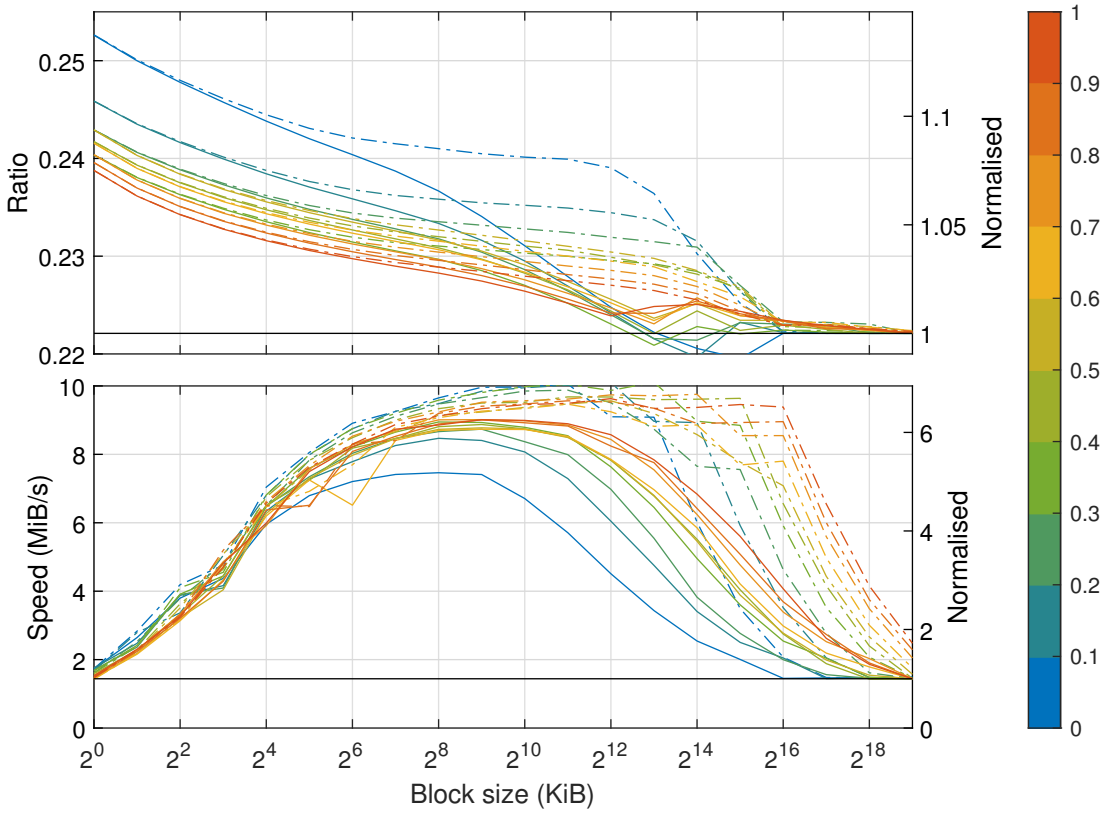

FIGURE 3.9: Effects of block size/propagation intervals on compression ratio (top) and speed (bottom) of varying fractional lengths of *enwik9*, i.e. 0.2 corresponds to the first 200MB of the 1000MB file. The solid and dashed lines correspond to the branching method and naive method respectively. The maximum thread count is fixed at 16. Note that the difference in compression ratio becomes smaller with increasing source length.

size increase, an alternative measure of parallelisation improvement is proposed, called the speed-ratio increase factor:

$$SRI = \frac{R(speed_{norm} - 1)}{R(ratio_{norm} - 1) + \epsilon}, \qquad R(x) = \begin{cases} x, x \geq 0 \\ 0, x < 0 \end{cases} \tag{3.7}$$

where the speed and ratio parameters are normalised against serial performance, $R(x)$ is the ramp function, and ϵ is a small, positive value, set to be 10^{-5} in this case, used to prevent division-by-zero errors. The rationale behind this metric is that it measures the speed gain per size gain, therefore the higher the speed increase for the same size increase, the better the parallelised algorithm.

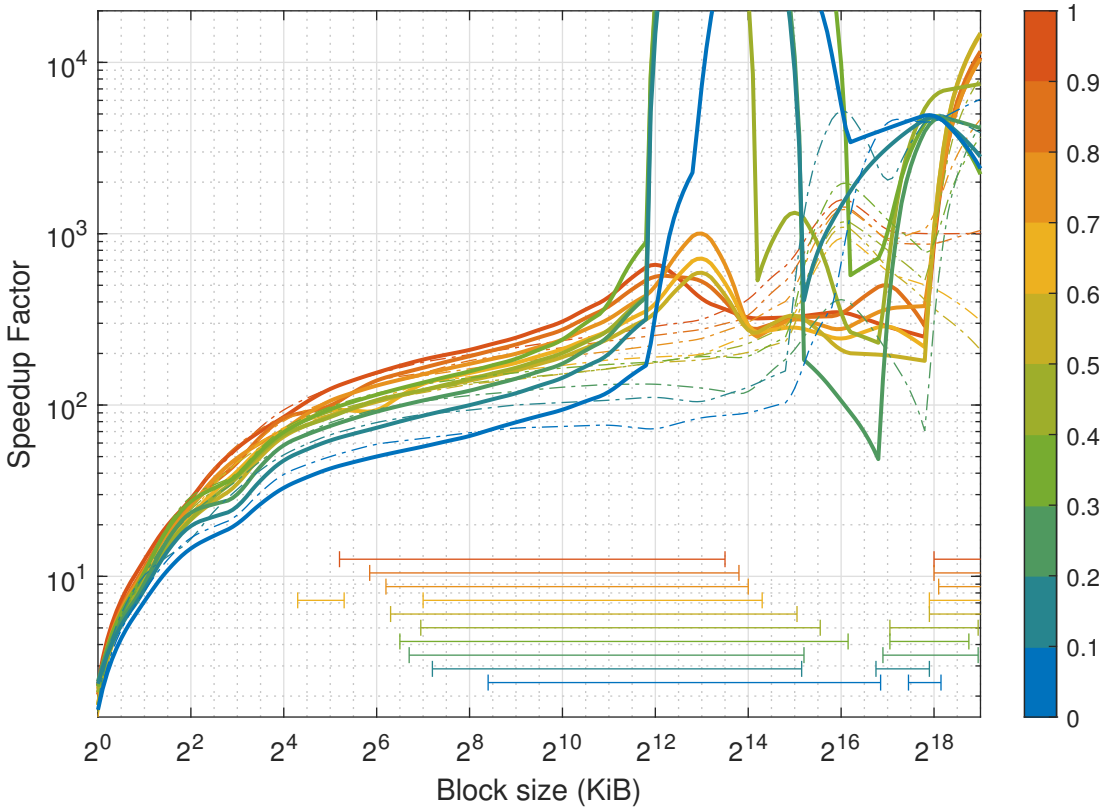

FIGURE 3.10: SRI of varying fractional lengths of *enwik9*, i.e. 0.2 corresponds to the first 200MB of the 1000MB file. The solid and dashed lines correspond to the branching method and naive method respectively. Horizontal bars at the bottom indicate the ranges where branching has higher SRI than naive. The maximum thread count is fixed at 16.

For a sense of scale, a SRI of 10^2 means that for every 1% increase in compressed size, the speed is increased by a factor of 1 (100%), likewise, an SRI of 10^3 corresponds to a factor of 10 (1000%) per 1% increase. The SRI plot (Fig. 3.10 shows that for smaller source lengths (¡500 MB), the branching method will perform better than the naive method at reasonable block sizes. Beyond 500 MB, however, the effectiveness of branching propagation is lessened, only performing better than naive propagation within a range of block sizes. The suspected reason for this is the major drop in compression ratio of the source at the 400-600 MB position (Fig. 3.8). The change in data patterns, while beneficial as it offers a better compression ratio at the given position, is detrimental as the pattern returns to "normal" past the 600 MB mark. For branching propagation, it effectively loses the benefits of branched learning at the beginning, since it has to re-adjust the models the

same way as the naive approach to account for the sudden change in redundancy pattern. Due to naive being inherently faster, it yields a better SRI while having ratio increases similar to that of branching for block sizes greater than 2^{13} KiB (8 MiB), while also being much faster. Branching does still present better SRI at the block range of 2^{8} KiB (256 KiB) to 2^{13} KiB (8 MiB) which is also the region in where we have a range of *balanced* to *saturation-dominant* propagation scheme, which ranges from 2^{10} KiB (1 MiB) to 2^{14} KiB (16 MiB). Note that branching propagation will perform better than the naive approach for source lengths below 500 MB, where there is a greater serial dependency. While the asymptotic effect from the better-than-serial performance does make the comparison more difficult, one could also point out that the K-independent scheme never performs better than serial in terms of ratios. This is despite both approaches using the same algorithmic implementation with the same model reset frequency. This suggests that the propagation scheme does transfer sufficient information in propagation which yields a better compression ratio, enough to offset the effect of model resets. Regarding the m-independent sources approach, since they are in essence upper bounds of K-independent approaches, we can expect that branching propagation will likely yield better SRI when compared to m-independent schemes, while providing better compression ratios.

3.4.6 Encoder-Decoder Mismatch

An important aspect of the branching propagation scheme is what can be termed an encoder-decoder mismatch, whereby the system specifications at the compression end and decompression end are different. This is critical, particularly in cases where the decompression machine is not as capable as the machine performing compression. Those scenarios will be illustrated with the use of three machines. The first one is an Intel Pentium Gold 6500Y mobile processor, with 4 logical cores with 8 GB of memory. The second setup is an Intel Core i5-10400F, with 12 logical cores and 16 GB of memory. The final machine is the primary setup used in previous benchmarks, with 16 logical cores and 32 GB of memory.

TABLE 3.4: Compression (encode) and decompression (decode) speed on different setups of *enwik8*, in MiB/s. The configuration (logical core counts, maximum threads) and compression speed of each setup are under the encode column. While decompression of *each* data is performed on all three setups with their respective logical core counts (max thread persists from the encoder). The dashed entries under decode (4) are due to insufficient memory. Block size is fixed at 500KB.

Encode		Decode Speed (MiB/s)		
(Cores, K)	Speed (MiB/s)	(4)	(12)	(16)
(4,4)	1.42	0.98	2.40	2.67
(12,16)	4.62	-	4.29	5.87
(16,16)	7.61	-	4.29	5.87

Table 3.4 shows the compression of data with the ideal maximum thread of each setup (i.e. K is set to the smallest power of two that is greater than or equal to the number of logical cores) followed by decompression of said data on all three setups. It is important to note that compression and decompression require the same parameters, i.e. same K and m. Thus, the decoding machine may be limited by the parameters determined by the encoder. With 1.1 GB per memory, the mobile machine is incapable of decoding any data with $K > 4$, as it has insufficient memory. On the other hand, the 12-core and 16-core machines are unable to reach maximum decompression speed when decoding with $K = 4$.

3.4.7 Unbounded Propagation

One might have observed this in the previous sections. A curious aspect of the proposed propagation scheme is scenarios primarily dominated by the propagation phase. Consider an arbitrarily large value of K such that $i \leq 2$ in 3.5, k will appear to be constantly doubling throughout (de)compression. This is referred to as unbounded model propagation, where K is indeterminable by the observer. While the number of logical threads available determines the actual speed and point of saturation in compression, it is preferable to set K to be arbitrarily large for future scalability. This refers to the storage of data with unbounded propagation. Supposedly, given a machine with only 4 logical cores, compressing the data at

$K = 4$ and $K = 64$ will take approximately the same amount of time (albeit one would require larger memory overhead, which may affect latency). However, with ever-improving hardware (both thread counts and memory size), it is undesirable to bind the stored data by $K = 4$ when the new machine has 64 logical threads. As we have seen in 3.4, given sufficiently large K (greater than available logical threads), the encoding and decoding machine's performances are independent of each other. This property of being able to fully utilise the computational power of any configuration for decompression is what we will refer to as future scalability. Future scalability is the primary advantage offered by the commonly used m-independent sources approach, as each block is independently modelled.

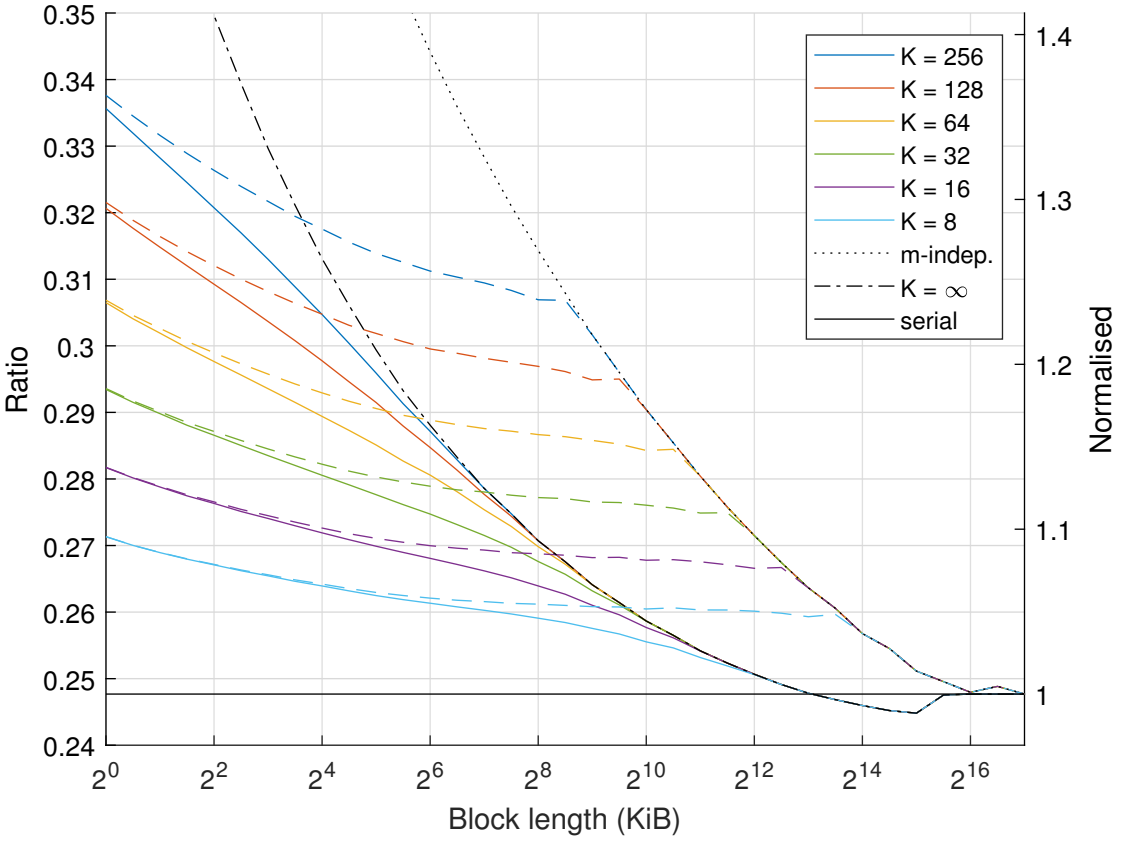

FIGURE 3.11: Effects of block size/propagation intervals on compression ratio for *enwik8*. The branching method is represented by solid lines and naive (i.e. K-independent) is represented by dashed lines.

Fig. 3.11 shows the performance of branching propagation compared to K-independent and m-independent sources at larger K. The results are emulated with a single thread at the cost of time for higher values of K due to the unavailability of a machine with sufficient logical thread count and memory capacity, as such, speed

TABLE 3.5: A comparison of various compression schemes.

	m-indep.	K-indep.	Branching	Serial
Dependency	No	Yes	Yes	Yes
Parallel	Yes	Yes	Yes	No
Scalable	Yes	No	Unbound	No

TABLE 3.6: Expected (normalised) compression ratio at higher thread counts, using a balanced configuration.

K	Balanced Ratio (normalised)		
	K-indep.	Branching	Δ
16	1.08	1.03	0.05
32	1.11	1.04	0.07
64	1.16	1.08	0.08
128	1.20	1.11	0.09
256	1.25	1.14	0.11

performance is omitted. Observe that the branching method is bound by both the corresponding K-independent curve and the branching curve of a higher K. We can see that the adjoining points of the branching curves form the bound for unbounded propagation $K = \infty$. Furthermore, note that an unbounded K-independent scheme will simply result in an m-independent sources scheme, also shown in Fig. 3.11, Table 3.5 summarises the comparison.

Assuming a balanced configuration will maintain around 80% maximum speed at higher thread counts, the corresponding differences in improvement-over-serial (normalised) are shown in Table 3.6. If the purpose of the compression scheme is more inclined towards compression size. It would be preferable to either limit K directly or indirectly via the block size. However, it would be more beneficial to make use of unbounded propagation, especially for long-term storage, as it offers a middle ground between the future scalability of the m-independent-sources scheme and the compression sizes of a serial scheme. The only major concern is that of model memory space. Where there is insufficient memory, one would need to either: 1) compute the blocks in separate passes, similar to how the results were emulated.

3.4.8 Model Selection in Saturation

We will briefly discuss the model selection idea described in Section 3.2.2, a point of investigation is the selection of a better-performing model to replace lesser competition. The approach taken takes the top-half best performers (in terms of ratios) and replaces the lower half with them in a one-to-one fashion. While there may be other possible methods to model selection, this dissertation would not be investigating the alternatives. Since the initial results (Fig. 3.12) are shown to be detrimental to not only speed (from additional model copies) but also compression ratio.

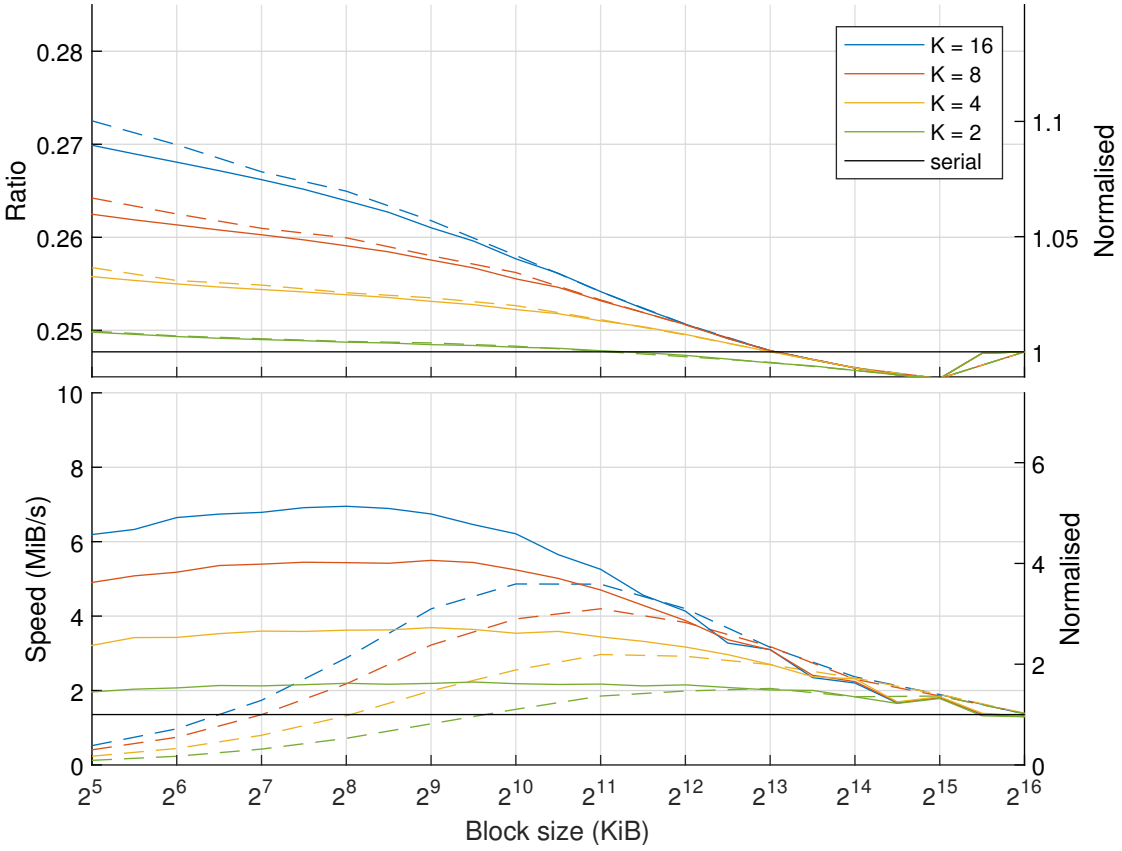

FIGURE 3.12: Effects of block size/propagation intervals on compression ratio (top) and speed (bottom) of *enwik8*, with varying maximum thread count. The solid and dashed lines correspond to the branching method without and with model selection.

3.5 Future Work

Much of the related work (Section 2.3) is based on the premise of GPU-acceleration. Since ranching propagation is already designed for models to be self-contained in a contiguous memory space, consequently, we can also adapt our scheme to GPU-based architectures with heterogeneous setups. This will allow for more experimentation at higher thread counts. However, the serial nature of our example algorithm will be much less suited for a GPGPU's kernel parallelisation. It would be better to test our scheme on alternative algorithms mentioned in Section 2.3, most of which are designed for in-kernel parallelism.

There is also the unexplored aspect of branching propagation that was mentioned in this paper. Effects of different branching factors β can be considered. Alternatives to model selection are unlikely, as copying does incur a significant slowdown. A compromised approach, where minor combination/information-sharing between models could be worth considering. To reduce memory usage, one could consider using a single model to compress multiple data blocks, race conditions will have to be negated in such implementations.

3.6 Conclusion

We presented a novel approach to parallelising any generic algorithm, called branching propagation. By adjusting the parameters of the scheme, a varying balance between compression speed and compression ratio can be achieved. While some compromises have been made in implementing a viable memory model, our scheme has still been shown to perform better than conventional approaches in standard, balanced configurations. This is done using the speedup-to-ratio increase (SRI) which is used as a unifying metric for speed and ratio trade-off that allows for better interpretation results in parallelisation of compression algorithms.

CHAPTER 4

Conclusion

The conventional approach of parallel compression algorithm, while simple and effective when speed is the only point of concern, can be rather detrimental when concerning compression ratios, as we have noted in the case of both m-independent and K-independent schemes. This is, however, also dependent on the serial dependency between the algorithm and the data source. As been shown, greater serial dependency results in far greater ratio penalties as does the number of parallel threads. Furthermore, algorithms with high serial dependencies are generally also highly sequential, making it difficult for tailor-made parallelisation of their functions.

This dissertation presents a novel solution to reduce the impact of parallelisation, called branching propagation. The algorithm itself assumes a generic divide-and-conquer approach and persists the algorithm's memory between data blocks. This can be applied to various types of algorithms. The branching propagation scheme provided some answers to our initial research questions:

1. *Is there a viable alternative to the conventional approaches of splitting data for parallel compression?*

   This is directly answered by the approach itself, the work described is a novel divide-and-conquer approach to improving the overall data assignment in a parallel compression environment. It offers not only better compression ratios but also better space-time trade-offs at sufficient serial dependency. A direct relation to Problem 2.

2. *If so, how would this hypothetical approach be affected by an algorithm's serial dependency?*

   It is first shown that serial dependency (i.e. how well it compresses data in serial compared to parallel) is dependent on not only the algorithm but also the source, greater source lengths will reduce the serial dependency the algorithm has on the source itself. As shown with the results, greater serial dependency yields better SRIs, while still providing adequate parallel speedups. However, the performance of branching propagation is heavily tied to serial dependency, to a point where insufficient serial dependency yielded worse SRI than the conventional. Branching propagation still offers better compression ratios, although with decreasing serial dependency the differences decrease as well.

3. *Finally, how do we quantify the parallel performance increase?*

   The solution to this is SRIs, which allow for one to take both speed and compression ratio into account. One can intuitively interpret an SRI value of 700 as a 700% increase in speed per 1% increase in size. While this metric is not perfect, as cases with better compression ratios than serial can result in an undefined SRI, it does, however, also make an argument that if the compression ratio is better than serial, there is no reason not to use it.

4.1 Recommendation & Future Direction

Much of the work presented in this dissertation covers only the basics. As pointed out, there are still many unexplored potentials of branching propagation. Some examples include aspects relating to the branching factor and sharing models between multiple threads to reduce memory usage. The memory architecture could do with additional refinements, as illustrated, model resets do affect the compression ratio. The generality and future scalability of branching propagation give it high relevance in modern data compression research, this is further reinforced by

its ability to parallelise highly sequential algorithms with adequate ratio penalties. One would not expect difficulties in adapting such an approach to other, potentially GPU-accelerated algorithms. It is worth reiterating that algorithms with high serial dependency are inherently more difficult to parallelise on a GPU, hence the lack of relevant work. It is, however, still a potential direction to explore the degree to which statistical algorithms can be parallelised. The greatest limiting factor is still the hardware limitations of branching propagation, memory and computing threads. But this will only increase and become more widely available with ever-improving technology. One could widely speculate that it may become widely adopted like its many predecessors.

Appendix A

Separate SRI plots

This appendix splits the SRI plot for *enwik9* in Fig. 3.10 into individual plots for legibility.

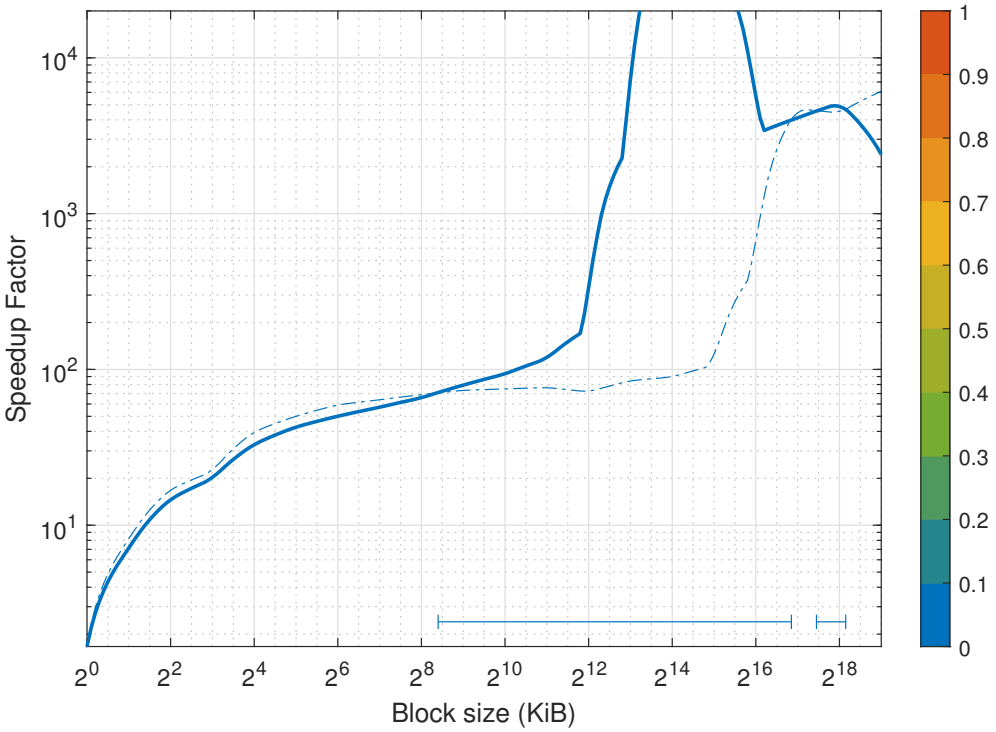

FIGURE A.1: SRI for source length = 100 MB

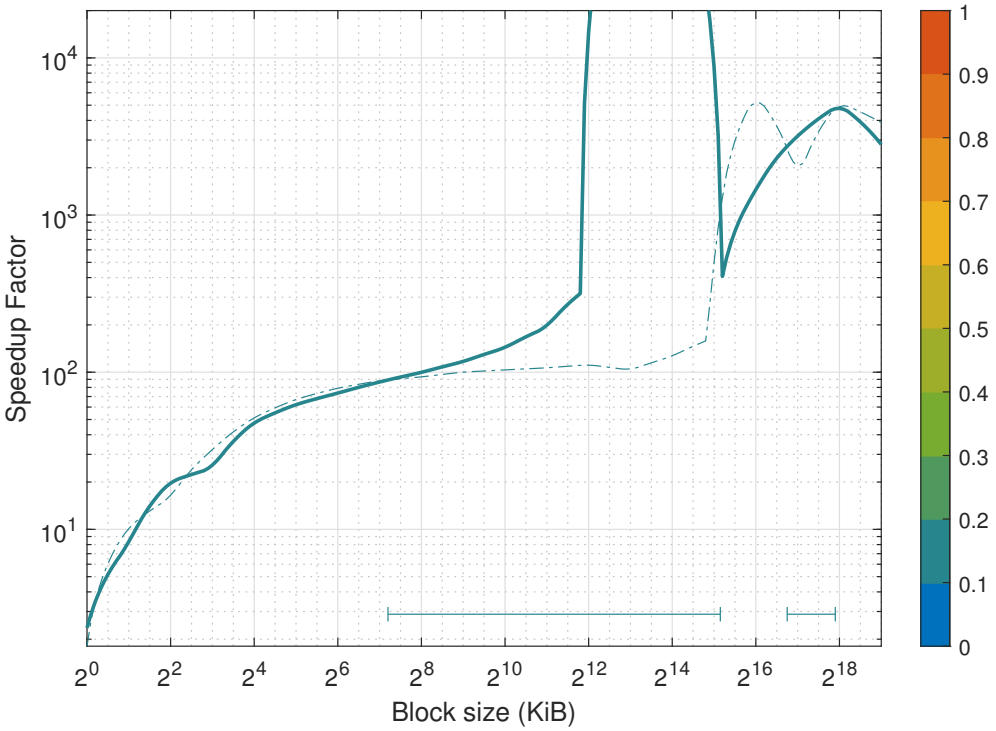

FIGURE A.2: SRI for source length = 200 MB

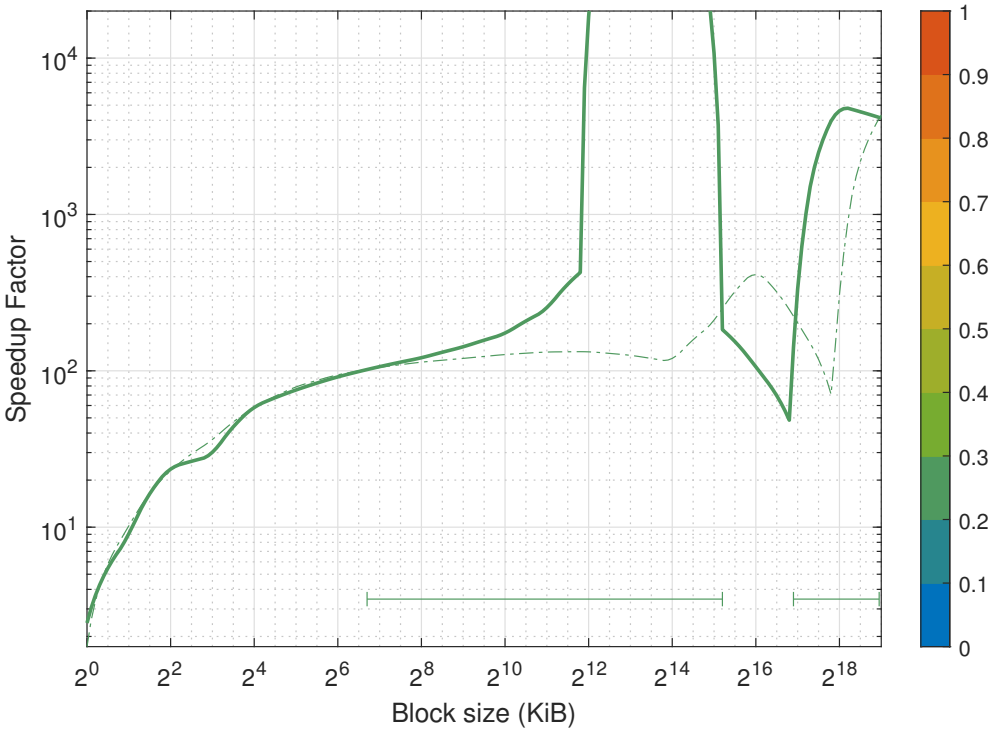

FIGURE A.3: SRI for source length = 300 MB

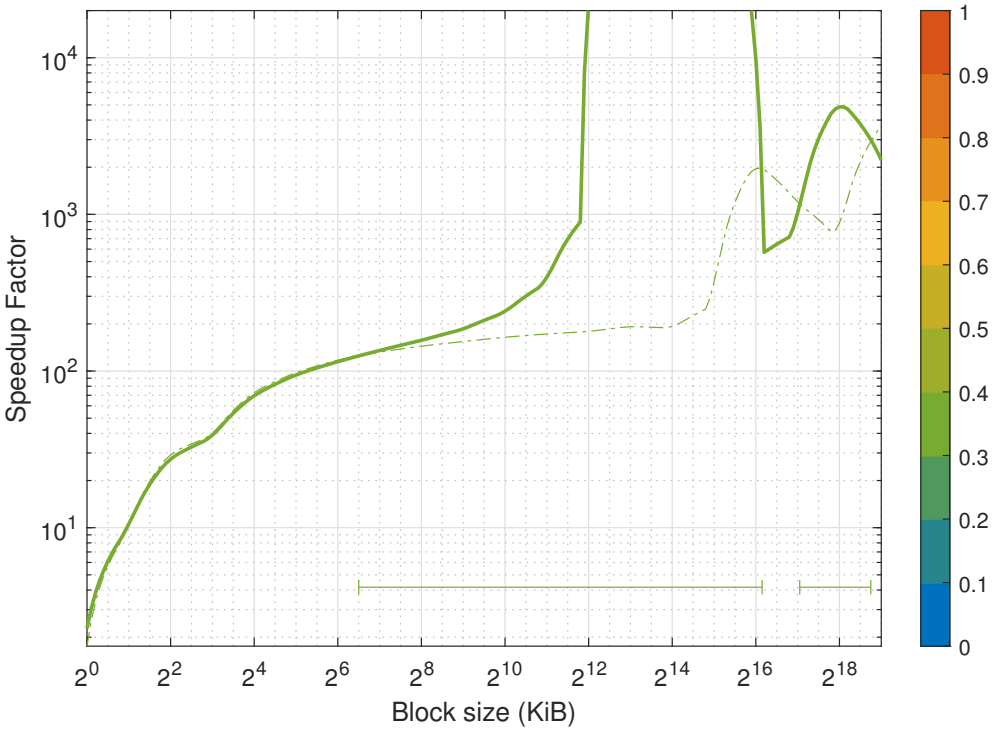

FIGURE A.4: SRI for source length = 400 MB

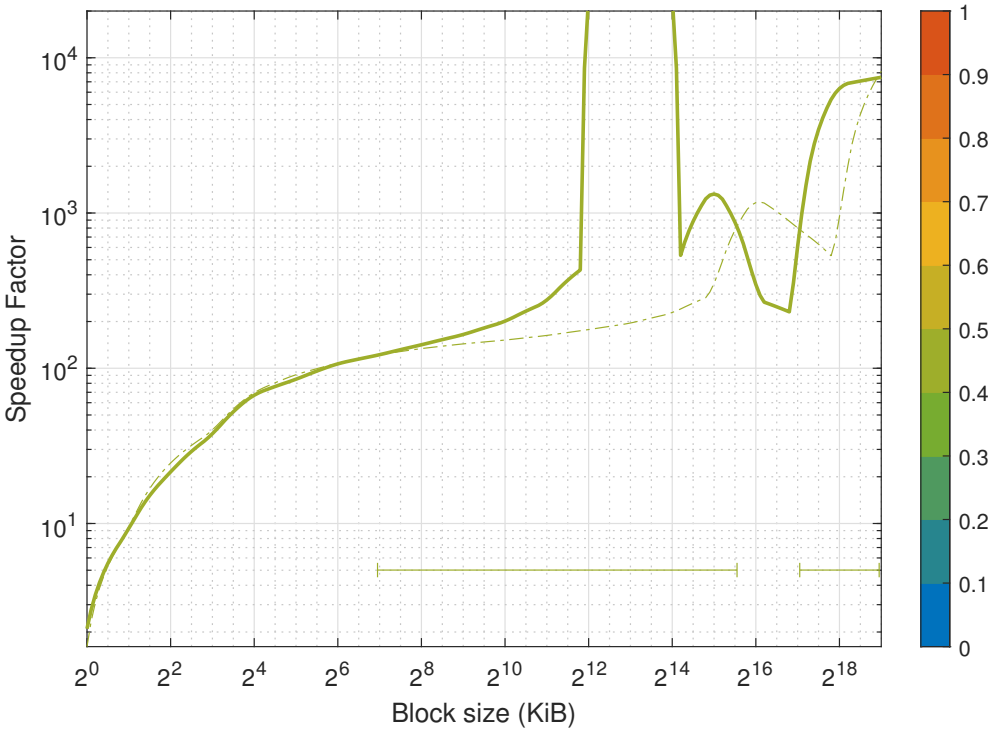

FIGURE A.5: SRI for source length = 500 MB

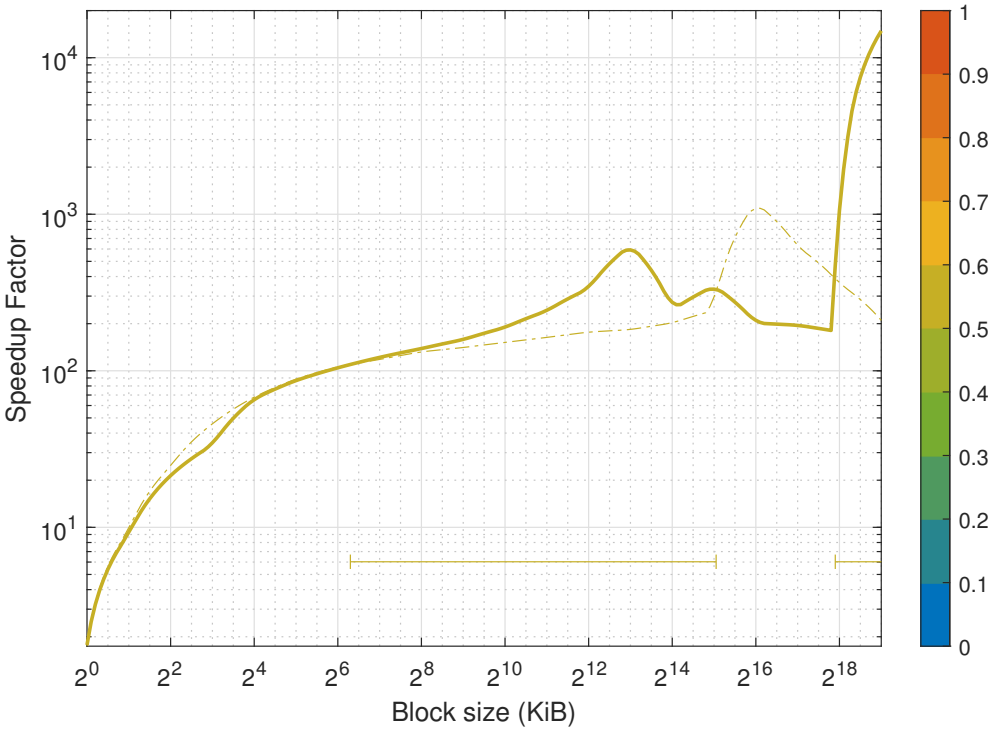

FIGURE A.6: SRI for source length = 600 MB

FIGURE A.7: SRI for source length = 700 MB

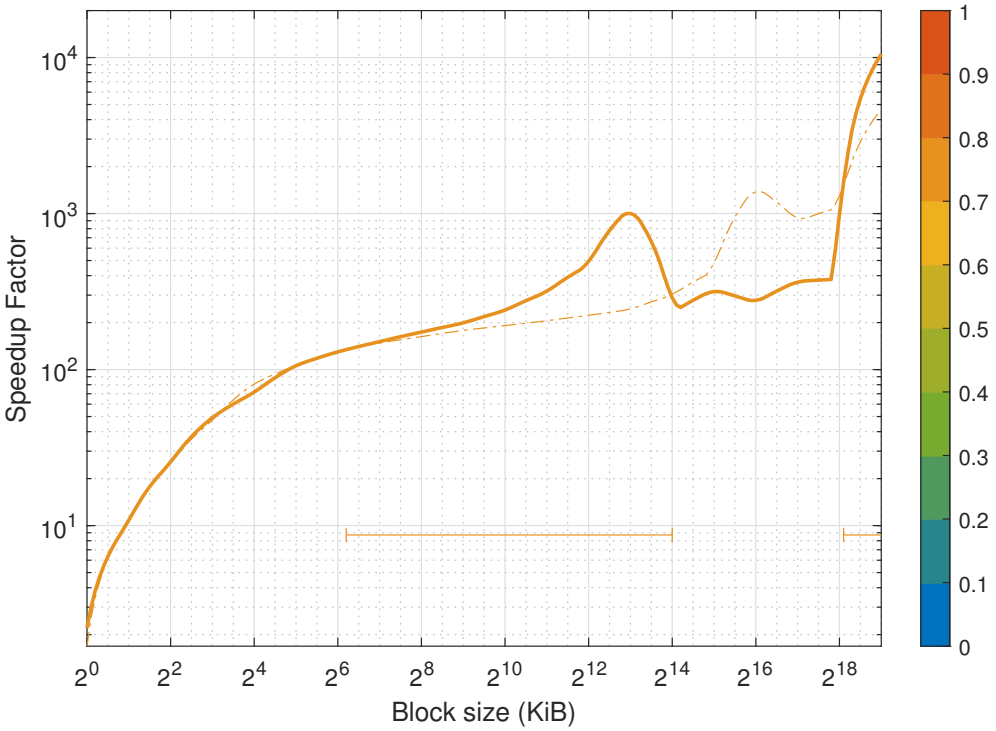

FIGURE A.8: SRI for source length = 800 MB

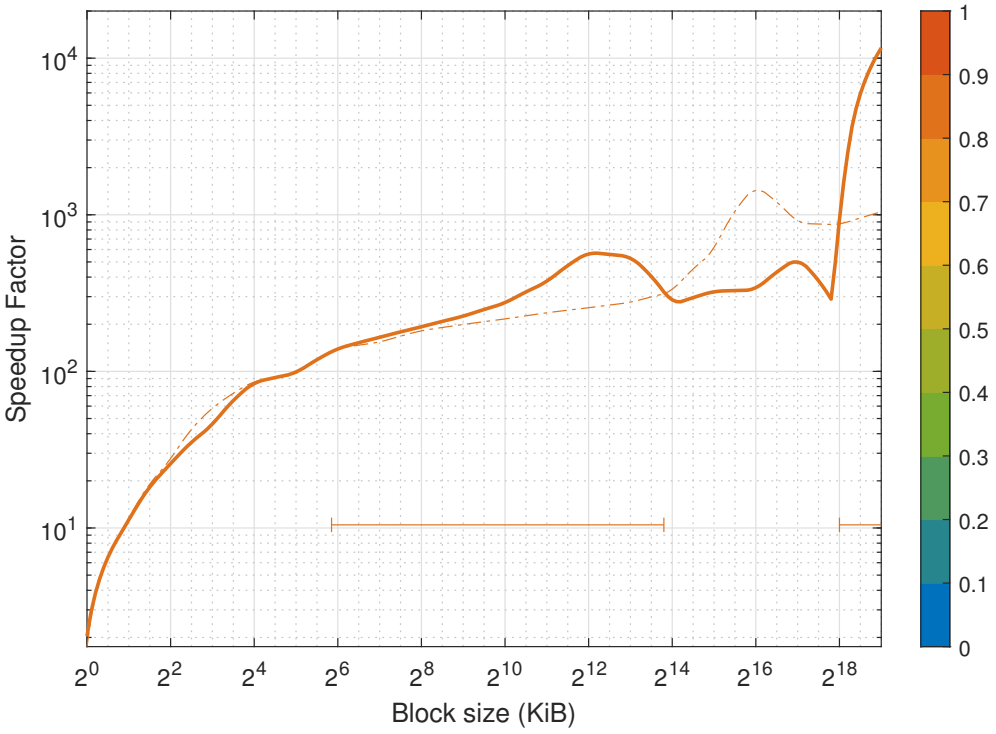

FIGURE A.9: SRI for source length = 900 MB

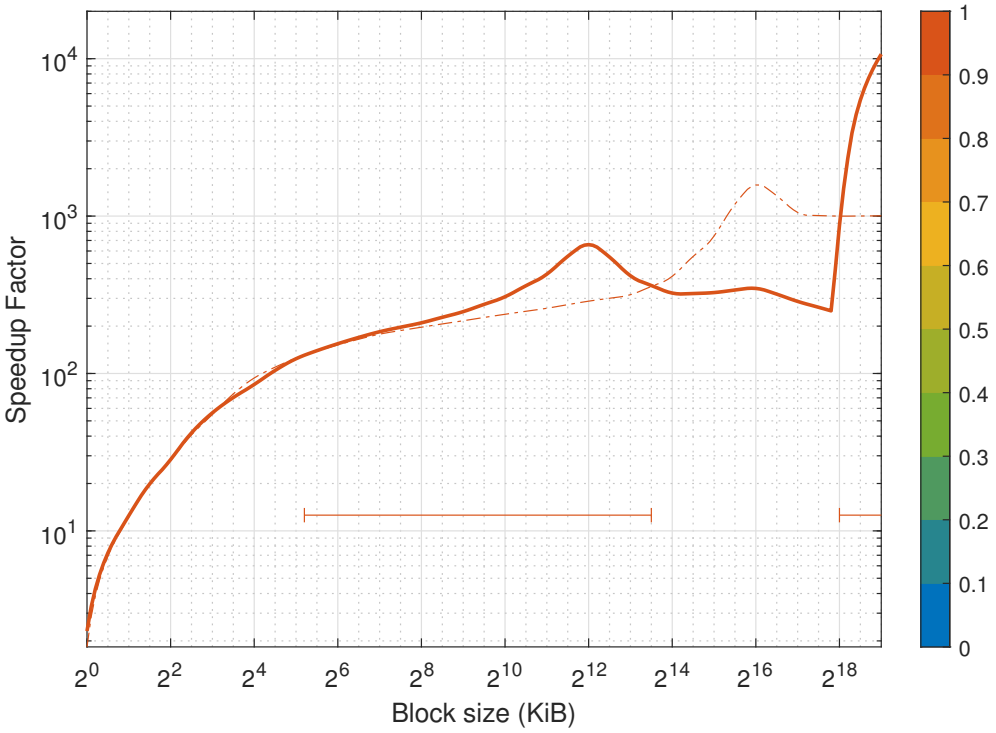

FIGURE A.10: SRI for source length = 1000 MB

References

[1] M. Mahoney, "About the test data," Sep 2011. [Online]. Available: http://mattmahoney.net/dc/textdata.html

[2] P. W. Katz, "String searcher, and compressor using same," Sep. 24 1991, uS Patent 5,051,745.

[3] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.

[4] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on information theory*, vol. 23, no. 3, pp. 337–343, 1977.

[5] C. E. Shannon, "A mathematical theory of communication," *The Bell system technical journal*, vol. 27, no. 3, pp. 379–423, 1948.

[6] J. Rissanen and G. G. Langdon, "Arithmetic coding," *IBM Journal of research and development*, vol. 23, no. 2, pp. 149–162, 1979.

[7] J. Duda, "Asymmetric numeral systems," *arXiv preprint arXiv:0902.0271*, 2009.

[8] J. Cleary and I. Witten, "Data compression using adaptive coding and partial string matching," *IEEE transactions on Communications*, vol. 32, no. 4, pp. 396–402, 1984.

[9] M. V. Mahoney, "Adaptive weighing of context models for lossless data compression," Tech. Rep., 2005.

[10] J. Ziv and A. Lempel, "Compression of individual sequences via variable-rate coding," *IEEE transactions on Information Theory*, vol. 24, no. 5, pp. 530–536, 1978.

[11] T. A. Welch, "A technique for high-performance data compression," *Computer*, vol. 17, no. 06, pp. 8–19, 1984.

[12] J. A. Storer and T. G. Szymanski, "Data compression via textual substitution," *Journal of the ACM (JACM)*, vol. 29, no. 4, pp. 928–951, 1982.

[13] I. Pavlov, "LZMA SDK (software development kit) - 7-zip." [Online]. Available: https://www.7-zip.org/sdk.html

[14] M. Burrows and D. Wheeler, "A block-sorting lossless data compression algorithm," in *Digital SRC Research Report*. Citeseer, 1994.

[15] A. Balevic, "Parallel variable-length encoding on GPGPUs," in *European Conference on Parallel Processing*. Springer, 2009, pp. 26–35.

[16] R. L. Cloud, M. L. Curry, H. L. Ward, A. Skjellum, and P. Bangalore, "Accelerating lossless data compression with GPUs," *arXiv preprint arXiv:1107.1525*, 2011.

[17] S. Funasaka, K. Nakano, and Y. Ito, "Fast LZW compression using a GPU," in *2015 Third International Symposium on Computing and Networking (CANDAR)*. IEEE, 2015, pp. 303–308.

[18] ——, "Light loss-less data compression, with GPU implementation," in *International Conference on Algorithms and Architectures for Parallel Processing*. Springer, 2016, pp. 281–294.

[19] N. Krishnan, D. Baron, and M. K. Mıhçak, "A parallel two-pass MDL context tree algorithm for universal source coding," in *2014 IEEE International Symposium on Information Theory*. IEEE, 2014, pp. 1862–1866.

[20] A. Ozsoy and M. Swany, "CULZSS: LZSS lossless data compression on CUDA," in *2011 IEEE International Conference on Cluster Computing*. IEEE, 2011, pp. 403–411.

[21] A. Ozsoy, M. Swany, and A. Chauhan, "Pipelined parallel LZSS for streaming data compression on gpgpus," in *2012 IEEE 18th International Conference on Parallel and Distributed Systems*. IEEE, 2012, pp. 37–44.

[22] A. Ozsoy, "CULZSS-Bit: A bit-vector algorithm for lossless data compression on GPGPUs," in *2014 International Workshop on Data Intensive Scalable Computing Systems*. IEEE, 2014, pp. 57–64.

[23] R. A. Patel, Y. Zhang, J. Mak, A. Davidson, and J. D. Owens, *Parallel lossless data compression on the GPU*. IEEE, 2012.

[24] K. Shastry, A. Pandey, A. Agrawal, and R. Sarveswara, "Compression acceleration using GPGPU," in *2016 IEEE 23rd international conference on high performance computing workshops (HiPCW)*. IEEE, 2016, pp. 70–78.

[25] E. Sitaridi, R. Mueller, T. Kaldewey, G. Lohman, and K. A. Ross, "Massively-parallel lossless data decompression," in *2016 45th International Conference on Parallel Processing (ICPP)*. IEEE, 2016, pp. 242–247.

[26] C. M. Stein, D. Griebler, M. Danelutto, and L. G. Fernandes, "Stream parallelism on the LZSS data compression application for multi-cores with GPUs," in *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. IEEE, 2019, pp. 247–251.

[27] B. Zhou, H. Jin, and R. Zheng, "A high speed lossless compression algorithm based on CPU and GPU hybrid platform," in *2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications*. IEEE, 2014, pp. 693–698.

[28] L. Lu and B. Hua, "G-Match: a fast GPU-friendly data compression algorithm," in *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. IEEE, 2019, pp. 788–795.

[29] F. M. Willems, Y. M. Shtarkov, and T. J. Tjalkens, "The context-tree weighting method: Basic properties," *IEEE transactions on information theory*, vol. 41, no. 3, pp. 653–664, 1995.

[30] L. P. Deutsch, "DEFLATE Compressed Data Format Specification version 1.3," RFC 1951, May 1996. [Online]. Available: https://www.rfc-editor.org/info/rfc1951

[31] J. A. Edwards and U. Vishkin, "Empirical speedup study of truly parallel data compression," Tech. Rep., 2013.

[32] M. Gonzalez Smith and J. A. Storer, "Parallel algorithms for data compression," *Journal of the ACM (JACM)*, vol. 32, no. 2, pp. 344–373, 1985.

[33] P. Franaszek, J. Robinson, and J. Thomas, "Parallel compression with cooperative dictionary construction," in *Proceedings of Data Compression Conference-DCC'96*. IEEE, 1996, pp. 200–209.

[34] J. Gilchrist, "Parallel data compression with bzip2," 01 2004.

[35] "Google/snappy: A fast compressor/decompressor." [Online]. Available: https://github.com/google/snappy

[36] I. H. Witten, R. M. Neal, and J. G. Cleary, "Arithmetic coding for data compression," *Communications of the ACM*, vol. 30, no. 6, pp. 520–540, 1987.

[37] A. Moffat, "Implementing the PPM data compression scheme," *IEEE Transactions on communications*, vol. 38, no. 11, pp. 1917–1921, 1990.

[38] "Squash compression benchmark." [Online]. Available: https://quixdb.github.io/squash-benchmark/