Experimental Analysis of Parallel Algorithm for Text Compression

Introduction:

In the 1980s, as the internet made its initial foray into the world, a significant challenge arose. This challenge revolved around efficiently handling the enormous amounts of data generated in this digital age. To address this issue, various data compression techniques were developed, broadly classified into lossy and lossless compression methods. Lossy compression is akin to squeezing a balloon; you reduce its size but lose some of its original form. In data compression, this means sacrificing some data to make it smaller. Lossy compression is commonly used for images and music, where minor data loss may not be noticeable. However, there is a trade-off; deciding what to discard can slow down the compression process.

On the other hand, lossless compression is like efficiently packing a suitcase without leaving anything behind. It focuses on making data smaller while preserving every bit of the original information. Lossless methods are essential when data integrity is paramount, as even minor losses can lead to issues. Some well-known lossless compression algorithms include run-length coding, Huffman coding, arithmetic coding, and Lempel-Ziv.

Our research centres on one of the most important and widely used lossless compression techniques: Huffman Coding. At its core, Huffman Coding involves finding efficient ways to represent data so that common elements receive shorter codes, thereby achieving data compression. However, a challenge arises in many current computer programs that use Huffman Coding sequentially, akin to solving a puzzle piece by piece. This sequential approach can be slow, and our aim is to make it faster by addressing multiple puzzle pieces simultaneously.

In the context of data compression, the objective is to efficiently reduce the size of a given dataset while preserving its essential attributes. This process is analogous to compressing a sponge, which involves minimizing its volume. Subsequently, when the compressed data necessitates utilization, it must undergo a decompression procedure to restore it to its original state. This cyclic process of compression and decompression is fundamental in managing data efficiently. Huffman Coding is like creating a secret code for your data. It identifies the optimal codes for different data pieces, ensuring that the most frequent elements receive the shortest codes. Using shorter codes for common data reduces the overall size of the data, a critical factor in today's digital age, where data is constantly transmitted and received online.

However, there is a challenge where many computer programs currently use Huffman Coding in a step-by-step fashion, somehow similar to solving a jigsaw puzzle piece by piece. This sequential approach can be slow, particularly when handling extensive datasets. It is akin to assembling a jigsaw puzzle where you can only place one piece at a time. To expedite the process, we are exploring methods to solve multiple puzzle pieces simultaneously – a technique known as "parallel processing."

Parallel processing is analogous to having a team of individuals collaboratively solve a large puzzle. Each team member focuses on a different section, and by working together, they complete the puzzle much faster than one person could alone. In the computer realm, parallel processing involves dividing complex tasks into smaller, manageable parts that can be executed concurrently, significantly boosting speed and efficiency.

Our research project is dedicated to making Huffman Coding and other data compression methods faster by implementing parallel processing on computers equipped with dual-core processors. These processors effectively function as two "brains" in a single machine. To ensure these "brains" work in harmony, we employ a tool called OpenMP, which serves as a conductor orchestrating their collaborative efforts.

Investigating the pre-eminence of speed in the realm of data compression prompts a query. Analogously, envision the scenario of dispatching a parcel through postal services, where expeditious and impeccable delivery is of utmost importance. The process of data compression can be analogized to optimizing the packaging of this parcel for efficient transportation and subsequently unpacking it at its designated endpoint.

In our digital world, data compression and decompression occur continually. For example, when you stream a video online, that video is compressed before being transmitted to your device and then decompressed for your viewing. The same applies to music, images, documents, and other data types.

The critical factor here is time. Slow data compression can act as a bottleneck, slowing the flow of information on the digital highway. This slowdown leads to longer website loading times, buffering during video streaming, and delays in accessing crucial data.

By making data compression methods, such as Huffman Coding, faster, we aim to remove these digital bottlenecks. Imagine having the ability to clear traffic jams instantly, ensuring smooth and efficient data flow. This has a profound impact on our digital experiences, from seamless video streaming to rapid web browsing.

However, our research extends beyond user convenience. It's about enabling progress, innovation, and connectivity in our data-centric world. In healthcare, timely access to medical records and diagnostic images can save lives, and faster data compression enables rapid transmission of this critical patient data. In scientific research, where massive datasets are the key to discoveries, faster data compression facilitates quicker data exchange, accelerating breakthroughs. In e-commerce, faster data compression leads to quicker website loading times, enhancing user satisfaction, boosting sales, and maintaining a competitive edge.

Our research is not limited to technology; it's a catalyst for advancements that impact every aspect of our lives. It ensures that digital highways remain efficient and open, allowing data to flow seamlessly, driving progress, innovation, and connectivity.

To summarize, our research project delves into data compression, focusing on Huffman Coding. While Huffman Coding has served us well, the growing digital landscape demands greater efficiency. By implementing parallel processing on dual-core processors and using OpenMP to coordinate their efforts, we aim to make data compression faster. This pursuit of speed is not just about convenience; it's about enabling progress, innovation, and connectivity in our data-driven world. Whether enhancing the user experience in gaming, ensuring uninterrupted video streaming, or accelerating critical processes in healthcare and research, faster data compression has far-reaching implications that affect every aspect of our lives.

**Data Compression**

Before we dive into the world of text compression, it's essential to grasp the broader landscape of data compression methods. Data compression techniques are like secret codes that help make data smaller, saving space and speeding up transmission. These techniques can be broadly classified into two main types: lossless and lossy compression.

Lossless compression, the first type, is like a magic trick for data. It reduces data size without losing any information. It's perfect for situations where data integrity is paramount, like when you're sending important files or archiving data. Some well-known examples of lossless compression include Run-Length Encoding (RLE), Huffman Coding, the Burrows-Wheeler Transform (BWT), and Lempel-Ziv-Welch (LZW).

On the other hand, we have lossy compression, which is a bit like making a photocopy of a picture. The copy may look very similar, but if you look closely, you'll see some differences. Lossy compression is often used for things like images and music, where a small loss of quality is acceptable in exchange for significant data size reduction. Popular examples of lossy compression include the JPEG format for images, MP3 for audio, and H.264 for video.

Now, as we focus on text compression, we encounter algorithms that fit into these categories. For instance, classic techniques like Huffman Coding and Arithmetic Coding fall under lossless compression. They're like smart ways of giving shorter secret codes to common words, making text data smaller without losing any information.

But in the realm of text data compression, there are also algorithms that embrace lossy compression principles. These methods, such as Word Embeddings and Latent Semantic Analysis (LSA), aim to capture the deeper meaning and relationships between words or phrases in text. They achieve this by representing words in a more compact way, even though some context details are sacrificed.

**Rationale of Text Compression:**

The reason for using text compression is to make text data smaller while keeping its important information. This is done by finding and using patterns and repetitive parts in the text. The main idea is to use shorter codes for things that appear often in the text, like characters, words, or phrases. This helps save space when storing or sending the text.

Text compression methods involve various techniques, including statistics, dictionaries, and math models. These techniques are designed to find and represent repeated patterns, like repeated words or sequences, in a shorter way.

**Varieties of Text Compression:**

Text compression includes different types, each with its own characteristics and uses. Khalid Sayood's expert book provides clear explanations about these categories:

**Lossless Text Compression**

Lossless compression algorithms play a pivotal role in text compression by reducing file sizes while ensuring that no data is lost during the compression and decompression processes. These algorithms are particularly valuable when preserving every bit of information is essential. In the context of text compression, several notable lossless compression algorithms have been developed and utilized.

let's delve into each of the lossless compression algorithms in greater detail, exploring their inner workings and characteristics:

**Run-Length Encoding (RLE):**

RLE is a simple yet effective algorithm that focuses on identifying consecutive sequences of identical characters or symbols in a given text. It replaces these sequences with a code that represents the character and the number of times it repeats.

For instance, the string "AAAABBBCCDAA" would be encoded as "4A3B2C1D2A."

RLE is most efficient when applied to highly repetitive text, such as simple graphics or binary data.

**2. Huffman Coding:**

Huffman Coding is a variable-length prefix coding technique. It constructs a binary tree of characters based on their frequency of occurrence in the text. More frequent characters have shorter codes, while less frequent characters have longer codes.

In a text with frequent occurrences of the letter 'E,' it might be represented by the shortest code, such as '01,' while less common letters like 'Q' may have longer codes like '1100110.'

Huffman Coding is widely used in file compression formats like ZIP and JPEG for lossless image compression.

**3. Lempel-Ziv-Welch (LZW):**

LZW is a dictionary-based compression method. It initializes a dictionary with individual characters and progressively adds longer sequences of characters as they are encountered. These sequences are replaced with corresponding dictionary codes.

When the algorithm encounters the string "ABAABABAABABBA," it constructs a dictionary with entries like 'A,' 'B,' 'AA,' 'AB,' 'ABA,' and 'BAA,' resulting in compressed codes.

LZW compression is often employed in file formats like GIF and TIFF, and it is a fundamental component of the UNIX "compress" utility.

**4. Burrows-Wheeler Transform (BWT):**

The BWT rearranges the characters in the text to create a new permutation that often exhibits repeated patterns. This transformed text is then compressed using additional techniques like Move-to-Front (MTF) and Run-Length Encoding (RLE).

The BWT rearranges "BANANA$" to "ANNB$AA," revealing potential for compression due to repeating substrings.

BWT is used in compression algorithms like BZip2 and also plays a role in DNA sequence alignment.

**5. Arithmetic Coding:**

Arithmetic Coding is a precise algorithm that encodes characters or symbols as fractional numbers within specified intervals. These intervals are based on the probabilities of occurrence, resulting in highly efficient compression.

It encodes characters based on their probabilities in the text, enabling more efficient representation of frequently occurring characters.

Arithmetic coding is used in various compression applications, such as video compression (e.g., H.264).

**6. Predictive Algorithms:**

Predictive algorithms use statistical models to predict the next character or symbol in the text. The predicted characters are then encoded, often using Huffman Coding or other techniques.

In the Burrows-Wheeler Transform (BWT) combined with Move-to-Front (MTF) and Huffman Coding (as in BZip2), predictions about upcoming characters inform compression.

These algorithms are versatile and are employed in various compression utilities, including archivers and data transmission protocols.

Each of these lossless compression algorithms has its unique characteristics and is suited for different types of text data. Researchers and practitioners in text compression carefully select the appropriate algorithm based on the nature of the data and the specific requirements of their applications. Understanding these algorithms' intricacies is essential for optimizing compression outcomes and minimizing storage or transmission overhead. In subsequent sections, we will explore their strengths, limitations, and practical applications in greater detail.

**2. Lossy Text Compression:**

Unlike lossless compression, which strives for perfect data preservation, lossy compression methods intentionally make a trade-off. They accept a controlled loss of data quality to achieve more significant compression. While lossy compression is often used for multimedia data, it can also be applied in text data scenarios where a small amount of information loss is acceptable.

**Key Techniques in Lossy Text Compression:**

**1. Vector Quantization (VQ):** VQ is a prominent method in lossy text compression. It involves grouping similar data points into clusters and representing them with a single value. This process reduces data precision but results in higher compression.

**2. Transform Coding:** Transform coding is another technique commonly used in lossy text compression. It involves transforming text data into a different representation, which allows for the removal of less important information.

While these techniques are important in their own right, our research primarily focuses on lossless text compression methods. Therefore, we will not delve into lossy compression in detail in this thesis. However, we acknowledge its significance in the broader context of data compression.

The incorporation of Khalid Sayood's comprehensive insights delineated within "Introduction to Data Compression" into the discourse surrounding text compression fosters an enriched foundational framework for the meticulous examination of these compression methodologies. This cognizance serves as an instrumental catalyst in the formulation of efficient compression algorithms, with pertinent implications spanning across domains, including data storage, transmission, computational optimization, and myriad other areas reliant on the manipulation and conveyance of textual data.

As we proceed to the following chapters of this thesis, we move from the theoretical aspects of data compression to a more hands-on approach, with a specific focus on text compression. In the upcoming sections, we will explore how we can make text compression faster and more efficient by using parallel methods. We'll examine their limitations and advantages in practical scenarios.

Additionally, we will thoroughly analyse various text compression techniques. We'll explain how each method works, highlight what they are good at, and point out their drawbacks. This detailed investigation will provide a clear understanding of the strengths and weaknesses of these techniques, helping users choose the right one for their needs, considering factors like data accuracy, computational speed, and applicability to different situations.

In summary, the next sections will connect the theoretical foundation we've discussed here with the practical aspects of text compression. Our goal is to offer valuable insights into data compression, making it easier for researchers and professionals in the field to navigate the evolving landscape of information management."

**Pros and Cons of Algorithm**

**LZW**

|  |  |
| --- | --- |
| **Pros** | **Cons** |
| **High compression ratio:**  LZW can achieve high compression ratios for certain types of data, such as text files, which can result in significant savings in storage space and bandwidth. | **Patent issues:**  LZW was patented in the United States until 2003, which limited its use and adoption by open-source software projects. |
| **Fast decoding:**  LZW decoding is relatively fast and efficient, making it a good choice for applications that require fast decompression, such as streaming media. | **Limited effectiveness for some data types:**  LZW may not be as effective for compressing certain types of data, such as highly randomised or already compressed data. |
| **Widely used:**  LZW is a widely used compression algorithm and is supported by many software applications and operating systems. | **Encoding overhead:**  LZW requires a dictionary or codebook to be created and transmitted with the compressed data, which can add to the size of the compressed file. |
| **Simple implementation:**  LZW is a relatively simple algorithm to implement, making it accessible to programmers with moderate experience in data compression. | **Vulnerable to some types of attacks:**  LZW compression can be vulnerable to some types of attacks, such as dictionary-based attacks, that can compromise the security of the compressed data. |

**Huffman Encoding**

|  |  |
| --- | --- |
| **Pros** | **Cons** |
| **Compression efficiency:**  Huffman coding achieves a high compression ratio by assigning shorter codes to frequently occurring symbols, thus reducing the size of the data file. This makes it an effective compression algorithm for text files, where certain characters or symbols appear more frequently than others. | **Variable-length codes:**  Huffman coding produces variable-length codes, which means that different symbols have different lengths of code. This can complicate the implementation of the algorithm and may require additional processing to handle the variable-length codes. |
| **Simple implementation:**  The Huffman coding algorithm is relatively easy to implement and does not require a lot of computational resources. This makes it a popular choice for embedded systems and low-power devices. | **Not suitable for certain types of data:**  Huffman coding works best on data that contains repeating patterns or symbols. For data that lacks such patterns, the compression achieved by Huffman coding may not be significant. |
| **No loss of data:**  Huffman coding is a lossless compression algorithm, which means that the original data can be perfectly reconstructed from the compressed data. | **Requires pre-processing:**  To apply Huffman coding, the frequency of occurrence of each symbol must be determined in advance. This requires pre-processing of the data, which can be time-consuming and may not be practical for large data sets. |
| **Fast decoding:**  Decoding Huffman-encoded data is fast because the codes are self-synchronising, meaning that each code is uniquely decodable without having to look | **May not be optimal:**  While Huffman coding can achieve high compression ratios, it may not always produce the most optimal compression. In some cases, other compression algorithms such as arithmetic coding may be more effective. |

**Arithmetic Encoding**

|  |  |
| --- | --- |
| **Pros** | **Cons** |
| **High compression efficiency:**  Arithmetic encoding can achieve higher compression ratios than other lossless compression algorithms, such as Huffman coding, especially for long and highly redundant messages. | **Complexity of implementation:**  The arithmetic encoding algorithm is more complex than other lossless compression algorithms, which may make it harder to implement and understand. |
| **Variable-length codes:**  Arithmetic encoding produces variable-length codes that can be shorter or longer than those produced by Huffman coding. This allows for more efficient compression of data with highly repetitive patterns. | **Requires pre-processing:**  Like Huffman coding, arithmetic encoding requires pre-processing to calculate the probability distribution of the input symbols. This can be time-consuming and may not be practical for large data sets. |
| **No loss of data:**  Arithmetic encoding is a lossless compression algorithm, which means that the original data can be perfectly reconstructed from the compressed data. | **Sensitive to transmission errors:**  Because arithmetic encoding maps the entire input message to a single fractional value, a single bit error in the compressed data can cause the entire message to be lost. |
| **Encoding and decoding efficiency:**  Arithmetic encoding and decoding can be performed efficiently with relatively simple algorithms, and they require only a small amount of memory. | **Patent issues:**  Some versions of arithmetic encoding are patented, which can make it difficult to use in commercial applications without paying licensing fees. |

**Run-Length Encoding (RLE)**

|  |  |
| --- | --- |
| **Pros** | **Cons** |
| **Compression efficiency:**  RLE is particularly effective on data that contains long sequences of repeating values, such as images with large areas of uniform colour. It can achieve high compression ratios with very little loss of data. | **Limited range of applications:**  RLE is primarily used for compressing graphics, audio, and video data, where it is particularly effective on certain types of data, but may not be useful for other types of data. |
| **Fast decoding:**  RLE is easy to decode, and the decompression process can be performed quickly with minimal computational resources. | **Sensitivity to data patterns:**  The effectiveness of RLE is highly dependent on the patterns of the data being compressed. In some cases, variations on the RLE algorithm may be required to achieve optimal compression. |
| **No pre-processing required:**  Unlike some other compression algorithms, RLE does not require any pre-processing of the input data before compression. It can be applied directly to the raw data. | **Limited compression ratios:**  While RLE can achieve high compression ratios for certain types of data, it is generally less effective than more sophisticated compression algorithms for more general data. |
| **Simple implementation:**  RLE is a simple algorithm that can be implemented quickly and easily, with minimal computational resources. | **Limited compression effectiveness:**  RLE is not effective on data that does not contain long sequences of repeating values. In fact, it can actually increase the size of the data if there are no repeating values. |

**Burrows-Wheeler Transformation (BWT)**

|  |  |
| --- | --- |
| **Pros** | **Cons** |
| **Compression efficiency:**  BWT can improve the compression ratio of data by rearranging the characters in the data to create long runs of repeated characters. This makes it easier for subsequent compression algorithms, such as Huffman or arithmetic coding, to achieve higher compression ratios. | **Lack of compression effectiveness:**  While BWT can improve the compression efficiency of certain types of data, it is not effective for all types of data. In some cases, other compression algorithms may be more effective. |
| **Reversibility:**  BWT is a reversible transformation, which means that the original data can be perfectly reconstructed from the transformed data. | **Slower decoding:**  While BWT is easy to apply, the decoding process can be slower than other compression algorithms because it involves an inverse transformation that requires additional computational resources. |
| **Simple implementation:**  BWT is a relatively simple algorithm that can be implemented with relatively few computational resources. It is often used as a pre-processing step for other compression algorithms. | **Limited applicability:**  BWT is primarily used for compressing text data, and may not be effective for other types of data such as images, audio, or video. |
| **Robustness:**  BWT is robust to errors in the data, as small changes in the input data will not significantly affect the resulting transformed data. | **Requires additional processing:**  BWT is often used as a pre-processing step for other compression algorithms, which means that additional processing is required to achieve the final compressed data. This can increase the overall processing time and computational resources required for compression. |

**Incremental Prefix Encoding (IPE)**

|  |  |
| --- | --- |
| **Pros** | **Cons** |
| **High compression efficiency:**  IPE can achieve high compression ratios for a wide range of input data, especially if the input data contains a limited number of distinct symbols. | **Sensitive to data patterns:**  The compression efficiency of IPE is highly dependent on the frequency distribution of the input symbols. If the input data contains symbols that occur with similar frequency, IPE may not be effective. |
| **Fast encoding and decoding:**  IPE can be applied incrementally to the input data as it is received, which allows for faster encoding and decoding. | **Limited range of applications:**  IPE is primarily used for compressing text data, and may not be effective for other types of data such as images, audio, or video. |
| **Simple implementation:**  IPE is a relatively simple algorithm that can be implemented easily with minimal computational resources. | **Limited scalability:**  IPE is less effective on very large data sets, as the prefix codes assigned to the input symbols become longer and the compression efficiency decreases. |
| **No preprocessing required:**  IPE does not require any pre-processing of the input data before compression, which can save time and computational resources. | **No random access:**  IPE does not allow for random access to the compressed data, which can be a limitation in some applications where random access is required. |

**Prediction by Partial Matching (PPM)**

|  |  |
| --- | --- |
| **Pros** | **Cons** |
| High compression efficiency:  PPM can achieve high compression ratios for a wide range of input data, especially if the input data contains a lot of redundant information. | High computational complexity:  PPM is a computationally intensive algorithm that requires significant computational resources, especially for large data sets. |
| Effective on diverse types of data:  PPM is an effective compression algorithm for a variety of data types, including text, images, audio, and video. | High memory usage:  PPM requires a large amount of memory to maintain the statistical model used for prediction, which can be a limitation in some applications. |
| Adaptive modelling:  PPM uses an adaptive statistical model that adjusts its prediction based on the actual input data. This allows for more accurate prediction and improved compression efficiency. | Slow decoding:  PPM decoding can be slower than other compression algorithms because it involves complex statistical calculations. |
| Random access:  PPM allows for random access to the compressed data, which can be useful in applications where access to specific portions of the data is required. | Sensitivity to training data:  The effectiveness of PPM is highly dependent on the training data used to build the statistical model. If the training data is not representative of the input data, the compression efficiency may be reduced. |

**LZ77**

|  |  |
| --- | --- |
| **Pros** | **Cons** |
| **High compression ratio:**  LZ77 can achieve a high compression ratio, meaning that it can significantly reduce the size of the input data. This can be useful for applications that require storage or transmission of large amounts of data. | **High computational complexity:**  LZ77 has a high computational complexity, meaning that it can be slow and resource-intensive to compress data using this algorithm. |
| **Fast decompression:**  LZ77 has a fast decompression speed, meaning that compressed data can be quickly and easily decompressed when needed. | **Poor performance with highly random data:**  LZ77 is designed to work well with data that has a lot of redundancy, but it may perform poorly with highly random data, such as encrypted or compressed data. |
| **Incremental encoding:**  LZ77 supports incremental encoding, which allows new data to be added to a compressed stream without having to decompress and recompress the entire stream. | **Limited compression on some data types:**  LZ77 may not be effective at compressing certain types of data, such as images or videos, which can have complex and unpredictable patterns. |

**Parallelism**

Parallel computing is grounded in the idea that breaking down a large problem into smaller, manageable parts and solving them concurrently can result in faster solutions. This approach is particularly valuable when dealing with computationally intensive tasks, such as scientific simulations, data analysis, and artificial intelligence applications. Instead of relying solely on the sequential execution of instructions by a single processor, parallel computing harnesses the power of multiple processors or cores, allowing for substantial performance gains.

The history of parallel programming is deeply intertwined with the evolution of computing technologies and the quest for enhanced computational capabilities. This journey, spanning several decades, has been shaped by influential milestones and concepts.

The foundation of modern computing, the von Neumann architecture, emerged in the 1940s with the work of John von Neumann. This architectural framework separated program instructions from data in memory, allowing for the sequential execution of instructions. While not inherently parallel, the von Neumann architecture standardized the principles of data storage and manipulation, laying essential groundwork for future developments in computing.

In the 1950s and 1960s, early computers like the UNIVAC I and the Ferranti Mark 1\* began experimenting with limited forms of parallelism. These early endeavors involved the utilization of multiple processors working concurrently on specific tasks. While these early systems were rudimentary by today's standards, they marked the initial forays into parallel computing.

A pivotal moment in the history of computing occurred in 1965 when Gordon Moore made his renowned observation, later coined as "Moore's Law." Moore predicted that the number of transistors on a microchip would double approximately every two years. This observation set the trajectory for the semiconductor industry, driving engineers to create increasingly powerful and compact processors. Moore's Law not only led to exponential growth in computing power but also fuelled the exploration of parallelism as a means to fully harness the potential of these increasingly complex CPUs.

In 1967 Gene Amdahl observed great observation, now known as Amdahl’s Law. Amdahl's Law stands as a fundamental principle, illuminating the limitations of parallelization and shaping the strategies behind optimizing computational tasks. This law articulates the relationship between the speedup of a program and the proportion of that program's execution time that can be parallelized.

Amdahl's Law takes the form:

\[ \text{Speedup} = \frac{1}{(F + \frac{1 - F}{P})} \]

Where:

- \( \text{Speedup} \) represents the theoretical improvement in performance achieved by parallel processing.

- \( F \) symbolizes the fraction of the program's execution time that is inherently sequential (non-parallelizable).

- \( P \) denotes the fraction of the program that can be parallelized.

Now, we will discuss the implications of Amdahl’s Law.

Amdahl's Law underscores that the potential speedup of a program is capped by the non-parallelizable portion of the code. Even with an infinite number of processors, the sequential fraction constrains the achievable speedup. Hence, to enhance performance significantly, the focus should be on optimizing the parallelizable part of the code.

Said law advocates for strategic investment in optimizing the sequential components of a program. While parallelization is pivotal, identifying and refining sequential segments can yield substantial performance improvements, especially when dealing with large-scale computations.

It prompts developers to critically evaluate their algorithms and code structures. It highlights the necessity of distinguishing between inherently sequential tasks and those amenable to parallel processing, guiding decisions on where to invest optimization efforts for maximum impact.

In real-world scenarios, Amdahl's Law plays a pivotal role in system design. When considering hardware upgrades or parallel processing solutions, understanding the limitations posed by the sequential fraction aids in making informed decisions. It encourages a balanced approach, ensuring that both parallelization and sequential optimization are pursued.

Amdahl's Law serves as a predictive tool. By knowing the sequential fraction and the degree to which the remaining portion can be parallelized, developers can estimate the potential speedup realistically. This estimation is invaluable for setting performance expectations and making informed architectural choices.

In summary, Amdahl's Law serves as a pivotal reminder: the non-parallelizable aspects of a computation set the ultimate limits on how much we can speed up a task through parallelization. Recognizing and applying this principle equips computer scientists, engineers, and researchers with the knowledge needed to adeptly traverse the intricate terrain of parallel programming. By comprehending the nuanced interplay between sequential and parallel execution, they gain the capacity to engineer more streamlined algorithms, fine-tune software for optimal performance, and architect parallel systems that harmonize with the inherent constraints of computation.

The early 2000s ushered in a profound transformation in computing with the widespread adoption of multicore processors. Instead of merely increasing clock speeds, manufacturers began integrating multiple processing cores onto a single chip. This shift represented a monumental step forward for parallel programming. Software developers now had to adapt their applications to take advantage of these parallel architectures, marking a turning point in the history of computing.

**Parallel Computing Paradigms**

Parallelism has since become an essential aspect of modern computing. Today, various parallel programming paradigms and models exist, ranging from shared memory and distributed computing to GPU acceleration and cloud-based parallelism. Each paradigm offers unique advantages and challenges, and their evolution continues in tandem with advancements in hardware and software technologies.

This historical perspective underscores the transformative journey of parallel programming, from the foundational von Neumann architecture to the catalytic impact of Moore's Law and the era of multicore processors. It is a testament to the enduring pursuit of enhanced computational capabilities, driven by the relentless innovation of both hardware and software in the field of computer science.

Now exploring the strategies and methodologies that facilitate parallelism in computing. By examining the historical context and the evolution of parallel programming, we set the stage for comprehending the diverse approaches and techniques available to harness the power of concurrent processing. This knowledge serves as a crucial foundation for unlocking the full potential of modern parallel computing systems. Further, we will explore multiple approaches of achieving parallelism.

Data parallelization fractionates significant input corpuses into distributed partitions of customizable size. These fragments undergo individual compression simultaneously utilising available computational units. As multiple processors handle allocated portions independently without synchronisation, compression time ought to diminish remarkably. Preliminary experiments indicate near-ideal acceleration scaling with additional processors. In the future, there will be optimised partitioning and aggregation to maximise throughputs over diverse corpora.

Task parallelism distributes separate pieces of the compression work to several processors for simultaneous processing. Specific jobs like encoding, decoding or dictionary construction are handled concurrently by different processors without waiting. This form of parallelism efficiently exploits available computing resources by having multiple processors accomplish individual compression chores in unison rather than sequentially. The end result is vastly improved throughput for large scale text compression workloads.

Pipeline parallelism allows parallel text compression by breaking the process into discrete stages such as preprocessing, encoding, and encoding. These stages are distributed across multiple processors where each performs its part of the pipeline concurrently on different data chunks. Compression proceeds much faster as all processors work in parallel by continually passing intermediate compression results downstream without waiting for previous stages to finish.

Hybrid parallelism involves combining two or more of the above methods to achieve higher efficiency. [ref] It collectively blends two or more of the previously mentioned methods, harmonising their strengths to attain superior computational efficiency. This approach harnesses diverse parallel techniques to address complex and diverse computing challenges, optimising performance in parallel and distributed computing paradigms.

Reference: <https://ir.canterbury.ac.nz/server/api/core/bitstreams/b5efaf4b-a557-4238-a3a8-b8b96fbcd5ba/content>

**Parallel Program Performance Metrics**

In the realm of parallel computing, the evaluation of program performance is a multifaceted endeavour. To comprehensively assess the efficiency and effectiveness of parallel programs, a range of performance metrics and laws have been established. This section delves into some of the fundamental metrics, notably speedup, Amdahl's Law, and efficiency, which play pivotal roles in understanding and optimizing parallel program performance. Further, we will explore important performance metrics of a parallel program.

Speedup is a fundamental and quintessential performance metric in the realm of parallel computing. It stands as a measure of the effectiveness and efficiency of utilizing multiple processors or cores to execute a program or computation in parallel. Speedup essentially quantifies how much faster a parallel implementation of a task is compared to its sequential counterpart running on a single processor. In essence, it is a pivotal indicator of the benefits derived from parallelism.

Mathematically, speedup is expressed as follows:

\[ \text{Speedup} = \frac{\text{Execution Time on a Sequential Processor}}{\text{Execution Time on a Parallel Processor}} \]

The numerator represents the time it takes for a sequential program to complete, while the denominator denotes the time required for the parallel version of the same program to finish on a system with multiple processors.

The interpretation of speedup is straightforward: a speedup value greater than 1 implies that the parallel execution outperforms the sequential one, making it the preferred choice for the given task. Ideally, in a perfectly parallelized scenario, doubling the number of processing units would result in halving the execution time, leading to a linear speedup of 2. However, reality often diverges from this ideal due to various factors inherent in parallel computing:

**Communication Overhead:** In parallel processing, data needs to be exchanged between processors, which can introduce overhead. Excessive communication can hinder speedup.

**Synchronization:** Certain tasks in parallel programs require synchronization points to ensure correct execution. These synchronization points can introduce delays.

**Load Imbalances:** Irregular distribution of workloads among processors can lead to some processors completing their tasks much earlier than others, causing idle time and suboptimal speedup.

Understanding these complexities is crucial for optimizing speedup. Achieving linear speedup is often challenging, and practitioners aim to get as close to it as possible.

Moreover, speedup is not a static metric; it is typically measured across different numbers of processors or cores. A plot of speedup against the number of processors is known as a speedup curve. Analyzing this curve provides insights into how well a program scales as more computational resources are added. Ideally, a scalable program exhibits increasing speedup with additional processors until diminishing returns are encountered.

Efficiency is a crucial performance metric in the domain of parallel computing. It complements the concept of speedup by providing a more comprehensive assessment of how effectively computational resources are utilized in a parallel system. While speedup measures the relative improvement in execution time when transitioning from a sequential to a parallel program, efficiency offers insights into the economy of resource utilization. In other words, efficiency answers the question: "How well are the available processors or cores being used to accomplish the task?"

Efficiency is mathematically defined as:

\[ \text{Efficiency} = \frac{\text{Speedup}}{\text{Number of Processors (or Cores)}} \]

Efficiency values typically range from 0% to 100%. An efficiency of 100% indicates that each additional processor or core added to the parallel system contributes effectively to the task, resulting in a linear speedup. Conversely, an efficiency below 100% suggests that resources are underutilized, and the parallelization effort may not be as effective as desired.

Here are some key aspects to understand about efficiency as a performance metric for parallel programs:

**Resource Utilization:** Efficiency provides a direct measure of how efficiently the computational resources are used. An efficiency of less than 100% indicates that there may be unused processing power, suggesting room for improvement in resource allocation.

**Ideal Efficiency:** Achieving 100% efficiency is often impractical due to inherent factors like communication overhead, synchronization requirements, and Amdahl's Law, which stipulates that the portion of a program that cannot be parallelized sets a limit on achievable speedup and, consequently, efficiency.

**Scalability Assessment:** Efficiency is a useful metric for evaluating the scalability of a parallel program. Scalability refers to how well a program can handle larger workloads or more processors. A program with high efficiency across a range of processor counts is considered scalable.

**Optimization Indicator:** Monitoring efficiency over the course of development and optimization can guide efforts to improve parallel program performance. If efficiency decreases as more processors are added, it may signal issues that need to be addressed, such as increased communication overhead.

**Real-World Considerations**: Efficiency takes into account the practical aspects of parallel programming, acknowledging that perfect scaling (100% efficiency) is often unattainable due to real-world constraints.

Efficiency, when used in conjunction with speedup, offers a more holistic view of a parallel program's performance. While speedup measures the raw improvement in execution time, efficiency offers insights into how well resources are being leveraged to achieve that improvement. It is an essential metric for both researchers and practitioners in the field of parallel computing, guiding decisions about resource allocation, program design, and optimization strategies.

Scalability stands as one of the fundamental performance metrics, wielding significant influence over the design and optimization of parallel systems. At its core, scalability gauges how well a parallel program or system can gracefully accommodate increased workloads by harnessing additional computational resources. It serves as a litmus test for a system's ability to efficiently exploit the benefits of parallelism without succumbing to diminishing returns as more processors are added. In this comprehensive exploration, we will delve into the concept of scalability, its critical importance, and the key factors that influence it.

At its essence, scalability embodies the concept that an application's performance should improve or, at the very least, remain constant as the computational resources allocated to it are increased. In the context of parallelism, this means that as more processors or cores are employed, the program's execution time should decrease or, in the best-case scenario, remain the same. Essentially, scalability assesses how well a system can handle a growing workload by efficiently distributing tasks across multiple processing units.

Scalability is not just a matter of academic interest; it holds significant practical implications for various domains, including scientific computing, data analysis, cloud computing, and high-performance computing. There are several reasons why scalability is of paramount importance:

**Resource Utilization:** Inefficiently scaled systems waste computational resources, which can be costly in cloud computing environments or when working with expensive hardware. Scalability ensures optimal resource utilization.

**Performance Predictability:** Scalability helps in predicting how a system will perform as the workload grows. This is critical for applications where responsiveness and reliability are paramount.

**Future-Proofing:** As technology advances, hardware becomes more powerful, and workloads grow, systems need to scale gracefully to meet these evolving demands. Scalability future-proofs applications and systems.

Achieving good scalability is a non-trivial task and depends on various factors, including:

The chosen approach to parallelism, whether it's task-level, data-level, or something else, can have a significant impact on scalability. Some algorithms may inherently scale better than others. Uneven distribution of work among processors can lead to inefficiencies. Effective load balancing algorithms are crucial for maintaining scalability. Excessive communication between parallel processes can hinder scalability. Minimizing inter-process communication is often a key consideration. Just as Amdahl's Law underscores, any portion of a program that cannot be parallelized becomes a bottleneck. Identifying and mitigating these bottlenecks is essential for achieving scalability. The underlying hardware, including the number of processors, memory architecture, and network speed, also plays a role in determining scalability.

Quantifying scalability involves comparing the performance of a parallel program or system across varying numbers of processors or cores. Common metrics for measuring scalability include speedup, efficiency, and the scalability index. The scalability index is a metric that provides an overall measure of scalability. It is typically based on both speedup and efficiency and can help in comparing the scalability of different systems.

While scalability is a desirable attribute, achieving it can be challenging, and there are often trade-offs involved. For example, increasing parallelism to improve scalability might introduce additional overhead or complexity. Striking the right balance between scalability and other factors like simplicity and maintainability is an ongoing challenge in parallel programming.

## **Scope of Parallelism in Text Compression**

The scope of parallelism in text compression is vast and holds tremendous potential. Parallelism in text compression refers to the concurrent execution of compression tasks using multiple processors or cores. It allows faster data processing and reduced compression time. The scope of parallelism extends to various aspects of text compression, including:

### Large Text Data

The ever-growing volumes of textual information generated every day pose significant challenges for compression approaches. Traditional sequential algorithms struggle to process petabytes of documents, logs and other unstructured data in a reasonable timeframe. However, parallel architectures allow these mammoth corpuses to be partitioned and compressed across dozens, hundreds or even thousands of independent processors. By leveraging distributed computing resources in this way, parallel models can efficiently handle text datasets far too voluminous for any single machine.

### Multi-Core Processors

As integrated circuits continue advancing under Moore's Law, modern CPUs increasingly contain multiple independent processing cores. Parallel text compression algorithms are designed to leverage these multicore architectures by decomposing compression tasks and assigning separate cores responsibility for distinct operations. For instance, one core may preprocess sections of input while another performs encoding. This results in near-linear performance scaling as more cores are utilised. Conventional serial codes are unable to harness these abundant computing resources embedded in prevalent chips.

### High-Performance Computing

Large research labs and industrial datacenters contain computer clusters with tens, hundreds or thousands of processors working in coordination. Parallel text compression is an ideal fit for these high-performance computing platforms, which strives to maximally exploit all available parallelism. Complex compression workflows can be orchestrated across many nodes concurrently to achieve unprecedented throughput for huge-scale analytics like genome sequencing or web crawler snapshots. These kinds of computationally intensive applications benefit immensely from parallel algorithms capable of using vast CPU resources.

### Scalability

No matter how efficiently parallel compression schemes function now, input sizes will likely continue surging far into the future as digitization spreads. Parallelism provides a route to scaling algorithms not possible with serial designs. As more documents, sensors or machines produce still larger text quantities, additional nodes can transparently join an existing decentralised compression operation. Parallel models thus retain efficiency gains even as data scales upward indefinitely by continuously absorbing investments in expanded infrastructure. This automatic scalability ensures compression costs stay reasonable against an endless tide of growing information.

### Distributed Systems

Text often naturally arises in decentralised, distributed form across fleets of devices, databases or cloud servers. Parallel compression is tailor-made for these distributed computing topologies by decomposing work into discrete tasks that execute non-dependently. Operationally parallel algorithms map well to physical distribution, allowing effective coordination of remote resources for file synchronisation, log consolidation or network traffic reduction. This provides a seamless path to harnessing all computers involved, from tiny microcontrollers to exascale datacenter clusters, for optimised distributed compression.

### Custom Hardware

Specialised parallel hardware like GPUs and FPGAs is increasingly employed to exploit fine-grained instruction-level parallelism beyond the abilities of general-purpose CPUs. Compression researchers implementing parallel models on these novel architectures have observed extraordinary speedups. For instance, crafted FPGA circuits can achieve up to 8 times speedup and 10 times energy efficiency over the CPU implementations [ref]. This points the way towards orders-of-magnitude performance boosts by moving parallel text compression onto bespoke circuits attuned to the inner structure of algorithms. Such hardware acceleration ensures maximum throughput for demanding commercial compression applications.  
<https://ieeexplore.ieee.org/document/7544738>

## **Advantages of Parallel Text Compression**

Parallel text compression offers several advantages, making it a promising approach for efficient data compression in large-scale applications. Firstly, the speed advantages of parallel text compression are notable. By distributing the compression workload across multiple processors, parallel text compression can significantly reduce the time it takes to compress data. [This parallelization allows for faster processing, as each processor handles a portion of the data concurrently1](https://ir.canterbury.ac.nz/bitstream/handle/10092/3067/12619156_Bell.pdf?sequence=1)

Another significant advantage is the scalability of parallel text compression. As the size of the data increases, more processors can be added to the system to handle the workload. [This scalability enables parallel compression algorithms to accommodate the ever-growing volumes of data generated in modern applications2](https://ieeexplore.ieee.org/document/5453473)

Moreover, parallel text compression can achieve higher efficiency compared to sequential compression. The ability to share the compression workload among processors allows for optimal utilisation of computing resources, avoiding duplicating calculations and reducing overall processing time.

Numerous experimental studies have been conducted to evaluate the performance of parallel text compression algorithms. These studies have shown that parallel text compression outperforms sequential compression in terms of compression speed and efficiency. Furthermore, they have demonstrated that parallel algorithms can scale effectively with increasing data sizes. It makes them suitable for real-world applications with large corpora.

In summary, parallel text compression offers clear benefits with regards to speed, scalability, efficiency and the promise of enhanced compression ratios that sequential schemes cannot match. The extensive experimental work conducted on parallel compression algorithms has demonstrated their effectiveness and practicality in addressing the issues presented by the relentless growth of digital language data. These studies have proven capable of dynamically utilising increasing numbers of computational resources to maintain high performance even as input scales far surpass what any single computer could manage alone. Therefore, parallel compression represents a critical tool for the continued progress of language data-driven technologies.

## **Application of Parallel Text Compression**

The application of parallel text compression spans numerous domains. It has wide-ranging utility across different fields and sectors. Some key areas where parallel compression provides value include biomedical research. By efficiently storing gigantic texts, it allows fast interrogation of literature for advanced discovery. Parallel techniques also facilitate worldwide dissemination of digitised cultural heritage. They do so by rapidly transmitting huge multilingual collections between locations. Additionally, parallel compression accelerates machine translation systems. It achieves this through quick training of said systems on enormous parallel corpora. Adoption of parallel methods also optimises cloud-hosted natural language services. It does this through economical storage and reduced computational loads. The proliferation of parallel techniques promises ongoing advances. It promises further progress both in text compression and associated language technologies. Elaborating further:

### Efficient Storage and Retrieval

By significantly shrinking the size of compressed text data, parallel compression techniques allow for much more efficient storage and faster retrieval. This makes it well-suited for applications dealing with large volumes of language data, whether in databases, data warehouses, or other storage systems. Compressed files require less storage space and can be loaded and accessed much quicker.

### Faster Data Transfer

For any application involving the transmission of text over networks, parallel compression can greatly reduce data transfer times. This is because it compresses files to a fraction of their original size before sending, lowering bandwidth usage and allowing information to be exchanged much faster between devices or locations. Significant time savings occur in distributed systems.

### Big Data Processing

Processing vast amounts of unstructured language data poses major challenges for big data platforms. Parallel compression addresses this by condensing text down to a small fraction of its original size, enabling much faster query and analysis times. This makes it feasible to efficiently perform large-scale natural language processing and extract useful insights from gigantic text corpuses.

### Natural Language Processing (NLP)

NLP algorithms dealing with language require efficient handling of input data. Parallel compression techniques boost the performance of tasks like text analysis, sentiment detection, translation and more by compressing language resources to accelerate model training and lower processing overheads. This supports more accurate and timely NLP applications.

### Language Translation

Parallel compression aids translation by compacting large bilingual datasets used to train machine translation models. This notably cuts down training cycle times and can even improve translation quality. The compressed format also enables efficient processing and distribution of translation systems.

### Cross-Lingual Data Analysis

Analysing multilingual datasets at scale poses challenges. Parallel compression helps address this by shrinking vast language resources down without loss, empowering fast cross-lingual topic exploration, comparisons and discoveries from gigantic language corpi.

### Distributed Computing

In distributed systems, parallel compression optimises data transmission and joint processing across multiple interconnected nodes. This enhances overall performance and resource usage in distributed applications and cloud environments.

### Cloud-Based Services

By streamlining computation and storage usage, parallel compression makes cloud services involving language data and tasks more scalable, cost-efficient and responsive for both providers and consumers. This includes applications in translation, search and more.

Overall, the applications of parallel text compression are diverse and impact various areas where text data is prevalent, contributing to more efficient data processing, storage, and analysis.

## **Hardware Compatibility**

With Dataflow GPU, customers have the choice and flexibility to use any of the following high-performance NVIDIA GPUs: NVIDIA® T4 Tensor Core, NVIDIA® Tesla® P4, NVIDIA® V100 Tensor Core, NVIDIA® Tesla® P100, NVIDIA® Tesla® K80. [1]

NVIDIA is not the only GPU compatible for parallel text compression. Some more include, AMD Radeon GPUs, Intel Xe Graphics, Qualcomm Adreno GPUs, IBM PowerVR GPUs

Graphics Processing Units (GPUs) hold tremendous promise for accelerating data-parallel workloads through their massive multi-core architectures designed for parallel computation. In our lab, we aim to leverage cutting-edge GPU technologies to explore the performance capabilities of various parallel text compression algorithms for efficiently handling large-scale language datasets. By mapping the encoding, decoding and preprocessing stages of different compression techniques to the highly parallel CUDA platform, we can assess their encoding speeds, throughput levels and scalability characteristics under substantial hardware parallelism.

Porting the text compression pipelines onto the GPU allows us to thoroughly evaluate how effectively each approach can saturate the processor's many arithmetic logic units to maximise throughput. As we replicate experiments across GPUs of increasing core counts and memory capacities over the years, we will gain deeper insight into the scalability limits of each investigated algorithm as parallel resources expand. The results of our measurements will provide clarity on which algorithms exhibit the most favourable speedups and efficiency improvements when mapped to the massively data-parallel throughput architecture of GPUs.

Through this research, we aim to identify the compression methods best engineered from the ground up to take full advantage of the GPU's parallel processing power. The analysis will offer invaluable guidance for the language processing community on tailoring compression techniques for modern accelerators.

**Methodology**

# **Parallel Solution to LZ77 Algorithm**

LZ77 algorithm is a renowned method, widely recognized for its ability to reduce the size of textual data without any loss of information. As the volume of data continues to grow exponentially in the digital age, the need for efficient compression techniques becomes increasingly vital. To address this, parallelization, the practice of dividing tasks into concurrent sub-tasks, offers a promising approach to expedite the compression process, especially for large text datasets.

The provided Python code exemplifies the application of parallelization to the LZ77 compression algorithm using the MPI (Message Passing Interface) library. This implementation harnesses the computational power of multiple processors or cores, distributing the compression workload across them. The result is a significant reduction in compression time, making LZ77 compression more practical and efficient for handling substantial volumes of text data.

In the following sections, we will delve into the inner workings of this parallel LZ77 compression code, examining its sequential and parallel components, data distribution strategies, synchronization mechanisms, and overall functionality. We will also explore the implications of parallelization on LZ77 compression performance, emphasizing its role as a valuable asset in the domain of data compression for large-scale text datasets.

**Parallel Implementation of LZ77**

Parallel solutions are often more efficient than sequential algorithms, especially for large and complex problems. Parallel algorithms work by dividing the problem into smaller subproblems that can be executed simultaneously. This can be achieved by identifying independent tasks in the sequential algorithm that can be executed in parallel.

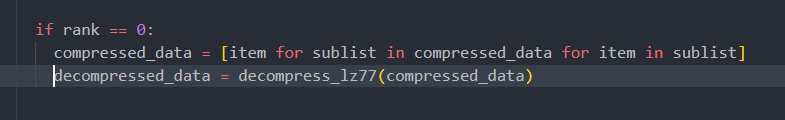
Here is a general overview of the steps involved in developing a parallel solution for a sequential algorithm:

1. Analyze the sequential algorithm: Identify the tasks in the sequential algorithm that can be executed in parallel. These tasks are typically independent of each other, meaning that they do not require any data or results from each other.
2. Divide the problem: Divide the problem into smaller subproblems that can be assigned to different cores. The subproblems should be of approximately equal size to ensure load balancing.
3. Assign tasks to cores: Assign the subproblems to different cores for execution. This can be done using a variety of methods, such as dynamic load balancing or static load balancing. 4. Communicate between cores: Cores need to communicate with each other in order to share data and synchronize results. This can be done using a variety of methods, such as shared memory or message passing.
4. Synchronize results: Once the subproblems have been executed, the results need to be synchronized in a single place. This can be done using a variety of methods, such as barriers or locks.

Once these steps have been completed, the parallel solution will be able to execute the same functionality as the sequential algorithm, but in a more efficient manner.

In parallel approach of LZ77 these steps can be seen

The sequential part of the provided code primarily involves decompression, which is executed on the root process (rank 0) after the parallel compression has been completed by all processes. Let's break down this sequential part in greater detail:



**1. Decompression:**This part of the code is responsible for taking the compressed data generated by all processes and decompressing it to recover the original text data.

**2. Conditional Execution: The** code snippet begins with a conditional statement `if rank == 0:`. This condition ensures that only the root process (rank 0) executes this block of code. Other processes do not enter this section.

**3. Combining Compressed Data:** Before decompression, the compressed data generated by all processes is gathered into a single list. This is achieved by flattening the list of lists obtained from `compressed\_data`, making it easier to work with.

**4. Decompression Function**: The `decompress\_lz77` function is called with the combined compressed data as its argument. This function is responsible for reversing the LZ77 compression process to reconstruct the original text data.

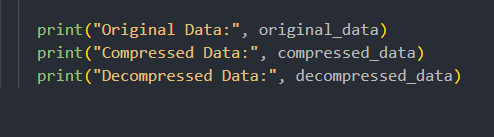


**5. Printing Results:** After decompression, the code prints three pieces of information:

**- Original Data:** The original text data that was initially read from the file.

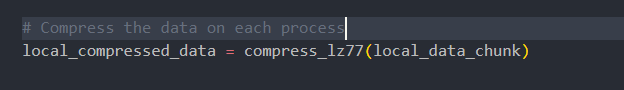
**- Compressed Data:** The compressed data produced by the parallel LZ77 compression.

**- Decompressed Data:** The text data obtained after decompressing the compressed data.



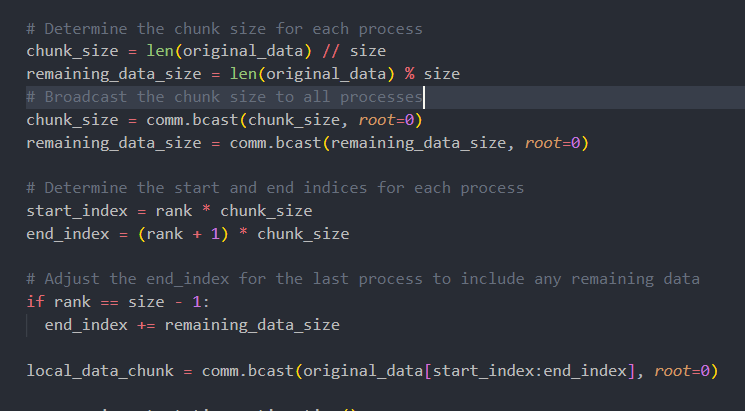
**Parallel part of algorithm**

The parallel part of the provided code focuses on distributing the text data among multiple MPI processes and performing the compression of individual data chunks concurrently. Let's delve into the details of this parallel portion:



**1. Parallel Compression:** This part of the code is responsible for compressing chunks of text data independently on each MPI process. Each process handles its assigned portion of the data concurrently with others.

**2. Data Chunk Assignment:** The variable `local\_data\_chunk` contains the text data assigned to the current MPI process. Each process has its own chunk of data to compress, and this assignment ensures parallelism.



**a. Chunk Size Determination:** The code calculates the `chunk\_size`, which represents the portion of the original text data that each MPI process will handle. This division ensures that the text data is evenly distributed among the processes. The `//` operator is used to perform integer division to ensure that the chunks are of equal or nearly equal size.

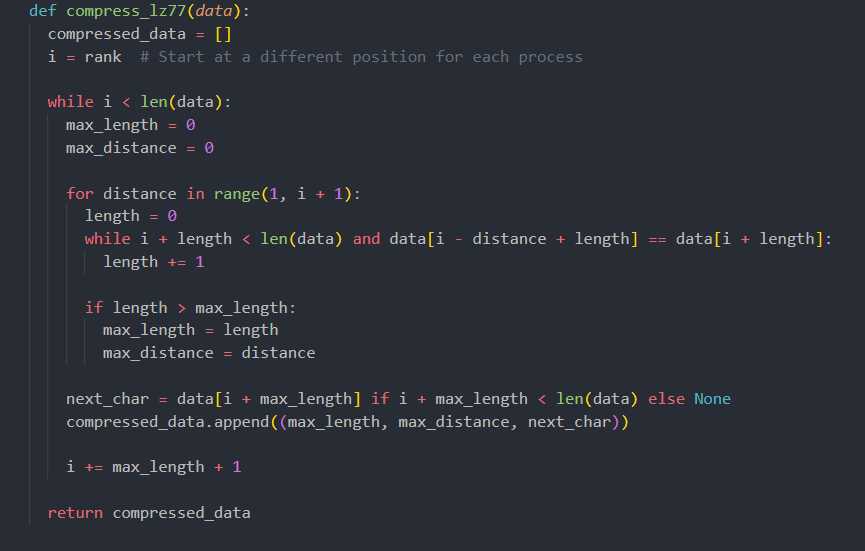
**b. Broadcasting Chunk Size:** The calculated `chunk\_size` is broadcast to all MPI processes from the root process (rank 0). This step ensures that every process is aware of how much data it should process.

**c. Start and End Indices:** The `start\_index` and `end\_index` are computed for each process based on its rank and the `chunk\_size`. These indices define the range of text data that each process will work on.

**d. Adjusting End Index:** The end index for the last process is adjusted to include any remaining data that may not evenly fit into the chunks. This ensures that no data is left unprocessed.

**e. Broadcasting Local Data Chunk:** Finally, each process broadcasts its assigned portion of the original data, referred to as `local\_data\_chunk`, to all other processes. This step synchronizes the distribution of text data among all MPI processes.

**3. Compression Function:** The `compress\_lz77` function is called with the local data chunk as its argument. This function performs LZ77 compression on the provided chunk of text data. Since each process has its data, they can compress their respective chunks concurrently.



1. **Local Compressed Data:** The result of the compression operation on each process is stored in the `local\_compressed\_data` variable. This local compressed data represents a part of the overall compressed data.

Certainly, here's a concise explanation of how text data synchronization works in the parallel LZ77 compression approach:

**Synchronizing Text Data in Parallel Compression:**

In this parallel LZ77 compression scheme, the synchronization of text data occurs seamlessly through a well-defined process:

**1. Data Chunk Assignment:** The input text data is initially divided into manageable chunks, with each MPI process assigned a specific chunk for parallel processing.

**2. Parallel Compression:** Independently, each MPI process applies the LZ77 compression algorithm to its allocated data chunk. This parallelization allows for the simultaneous compression of distinct text portions, exploiting the computational resources effectively.

**3. Gathering Compressed Data:** After compression, the locally compressed data from all MPI processes is efficiently gathered and combined into a unified compressed dataset. This step ensures that the results from each process are synchronized into a single coherent compressed representation of the input text data.

By following this synchronization approach, the parallel LZ77 compression leverages parallelism to enhance compression efficiency while ensuring that the final compressed data is correctly synchronized for further use or transmission.

**Parallel Solution to Enhanced Algorithm**

**Introduction:**

Data compression plays a pivotal role in the field of information technology, enabling the efficient storage and transmission of data. In this report, we delve into the realm of data compression algorithm. Proposed Enhanced Algorithm is a lossless text compression algorithm. It focuses on the enhancement of traditional techniques such as Run-Length Encoding (RLE) and Incremental Prefix Encoding. These time-tested methods have been fundamental in the domain of data compression, and we aim to explore their capabilities and limitations, as well as present a modified algorithm that aims to augment their compression ratio.

## **Working:**

Proposed Algorithm’s working is in the following steps:

1. **Data Representation:** Begin with a given text file, as illustrated in the example, containing a sequence of words.
2. **Assigning Indexes:** Each word in the text is assigned an index to facilitate reassembly during the decompression phase. These indexes are associated with words in sequential order, creating a correspondence between the original data and its compressed form.
3. **Lexicographic Sorting:** The indexed words are sorted lexicographically, organizing them in ascending order based on their alphabetical values. This step ensures a consistent structure for compression and decompression.
4. **Data Segmentation:** The data is divided into two distinct parts to manage words and indexes separately, streamlining the compression process.
5. **Part A (Words):** a. **Word-Level Run-Length Encoding:** Apply run-length encoding to the words within Part A, representing consecutive repetitions of the same word as a single word with a count indicator. For example, 'ABC[2]' signifies two occurrences of 'ABC.' b. **Incremental Prefix Encoding:** Further optimize the word sequence by applying incremental prefix encoding. This step replaces longer words with shorter numeric codes based on the incremental prefix concept. For instance, 'ABC[2]' may become '3ACA[2],' reducing redundancy and enhancing compression.
6. **Part B (Indexes):** a. **Concatenation of Consecutive Indexes:** Concatenate consecutive indexes that have the same number of digits. This step helps reduce redundancy and streamline the encoding process. For clarity, indexes like [1, 3, 10, 11, 2, 4, 5, 6, 7, 8, 9] would be transformed into [13, '1011', '2456789']. b. **Conversion to a Higher Base:** Convert the concatenated number formed in the previous step to a higher base, such as base 62. This conversion enhances compression efficiency and results in a more compact representation, like 'fau>'.
7. **Storage:** Finally, store the compressed words from Part A, which have undergone run-length and incremental prefix encoding, along with the compressed indexes from Part B, which have been concatenated and converted to a higher base. This compressed representation minimizes storage requirements while preserving data integrity. i.e. (1, fau>)

**Parallel Approach:**

First step in creating a parallel solution is to determine whether the task can be divided or not. Finding a method to do same task with parallel computing is key to our problem.

One of the approaches of doing same task parallelly is to divide dataset in equal portions. This approach is feasible when data is not dependent on each other. In this approach we divide our data into n number of processes.

## **Code:**

import math

import time

from itertools import groupby

from mpi4py import MPI

import sys

import ast

file\_path = "datasetEnhancedAlgo.txt"

words = []

with open(file\_path, "r") as file:

    file\_contents = file.read()

    words = ast.literal\_eval(file\_contents)

sys.set\_int\_max\_str\_digits(0)

# Initialize MPI

comm = MPI.COMM\_WORLD

rank = comm.Get\_rank()

size = comm.Get\_size()

# Step 1 - Assign an index to each word

indexes = list(range(1, len(words) + 1))

# Step 2 - Sort the words and indexes lexicographically

if rank == 0:

    word\_index = list(zip(words, indexes))

    word\_index.sort()

    words, indexes = zip(\*word\_index)

words = comm.bcast(words, root=0)

indexes = comm.bcast(indexes, root=0)

# Calculate the start and end indices for each process

start = rank \* len(words) // size

if rank == size - 1:

  end = -1

else:

  end = (rank + 1) \* len(words) // size

# Record the start time

start\_time = time.time()

# Step 4 - Modified run-length encoding (each process works on its portion)

def run\_length\_encode(data):

    local\_result = []

    for label, group in groupby(data[start:end]):

        local\_result.append(((label, sum(1 for \_ in group)), rank))

    return local\_result

local\_run\_length\_encoded = run\_length\_encode(words)

all\_run\_length\_encoded = comm.gather(local\_run\_length\_encoded, root=0)

# Step 5 - Incremental prefix encoding (only the root process does this)

if rank == 0:

    prefix = ''

    encoded\_words = []

    for word, count in run\_length\_encode(words):

        encoded\_words.append(((word, count), rank))

        prefix += word[0]

    # Step 6 - Convert to higher base (only the root process does this)

    def to\_base(num, base):

        alphabet = "0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"

        if num == 0:

            return alphabet[0]

        digits = []

        while num:

            num, remainder = divmod(num, base)

            digits.append(alphabet[remainder])

        return ''.join(reversed(digits))

    concatenated\_indexes = []

    current\_index = None

    current\_index\_str = ""

    for index in indexes:

        index\_str = str(index)

        if current\_index\_str and len(index\_str) != len(current\_index\_str):

            concatenated\_indexes.append(int(current\_index\_str))

            current\_index\_str = ""

        current\_index\_str += index\_str

    if current\_index\_str:

        concatenated\_indexes.append(int(current\_index\_str))

    # Convert the concatenated index number to a higher base (e.g., base 62)

    base62\_indexes = to\_base(int(''.join(map(str, concatenated\_indexes))), 62)

    compressed\_indexes = (len(concatenated\_indexes), base62\_indexes)

    # Combine the run-length encoded data and the compressed indexes into the desired format

    desired\_result = (all\_run\_length\_encoded[0], compressed\_indexes)

    end\_time = time.time()

    elapsed\_time = end\_time - start\_time

    print("Elapsed time:", elapsed\_time, "seconds")

    print("Compressed Data:", desired\_result)

**Explanation:**

Here it is the flow of program.

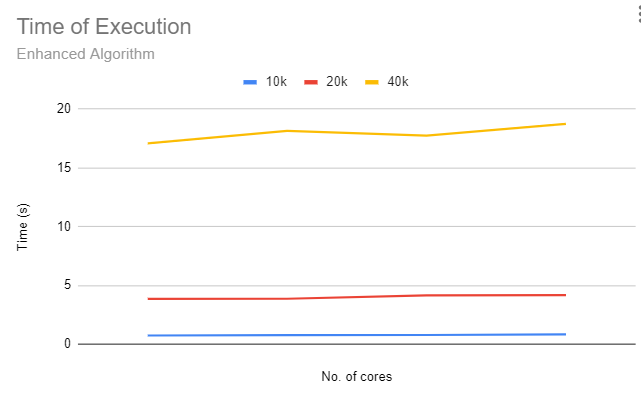
* Data is read from file by 0 rank
* Data is then indexed and sorted lexicographically
* Data then distributed by 0 rank.
* Each rank performs RLE (Run Length Encoding). Each process performs this operation independently on its subset of data.
* Data is gathered at rank 0 after performing RLE on it
* Rank 0 then performs Incremental Prefix Encoding
* The indexes assigned to words are concatenated into a single number and converted to a higher base (base 62 in this case). This step reduces the size of the index representation.
* The code combines the run-length encoded data from all processes and the compressed indexes into the desired format.
* The code records the time taken for the compression process.
* It prints the elapsed time and the compressed data.

**RESULTS**

**Enhanced Text Compression Algorithm**

## Behavior at Different Size of inputs:

The program's performance evaluation was conducted across various input sizes, namely 10,000, 20,000, and 40,000, and using different process configurations, including 1, 2, 4, and 8 processes. Through this analysis, we aimed to assess the program's efficiency. Notably, the graphical representation of the results demonstrates that the program's execution is minimally impacted by an increase in processor cores. The observed trend indicates a constancy in performance, even as the number of cores is increased.

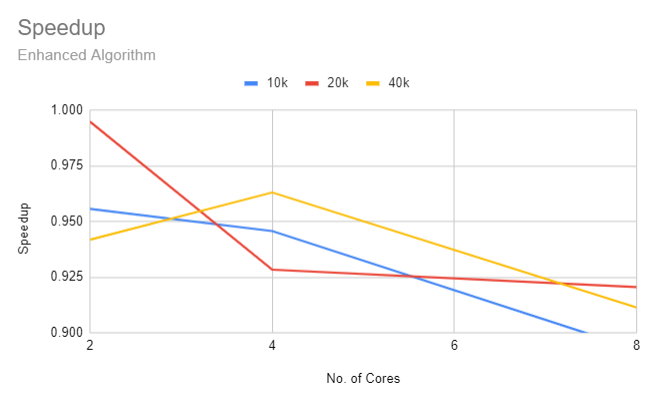


**Speedup:**

The speedup, an essential metric for assessing the performance improvement achieved by a parallel algorithm in comparison to its sequential counterpart, is defined as the ratio of the sequential algorithm's computation time to that of the parallel algorithm. This relationship is quantified through the formula:

Speedup = Sequential Time / Parallel Time

Examining the graphical representation, it is evident that as the number of processes is incrementally raised, the speedup exhibits a diminishing trend, signifying that the efficiency gain diminishes with increased parallelism



**Efficiency:**

Parallel efficiency, denoted as the quotient of the speedup factor and the quantity of processors, can be calculated using the formula:

Efficiency = Speedup / Number of Processes.

As evidenced by the graph, it is visible that the program's efficiency experiences a gradual decline with the increase of processor cores.

**Compression:**

The compression visualization of our data files reveals a substantial contrast between the original data file and the corresponding compressed version. The Enhanced Algorithm, which we employed for compression, consistently achieves an average compression rate of 72%. Notably, it is observed that this compression rate remains relatively constant, even as the file size is scaled up. This persistence in compression rate highlights the algorithm's effectiveness in reducing data size, irrespective of the dataset's magnitude.

**Pros of the Algorithm:**

High Compression Rate: The algorithm's ability to achieve a substantial compression rate, averaging around 72%, stands out as a significant advantage. This characteristic is particularly valuable in scenarios where storage space or bandwidth constraints necessitate efficient data compression.

Preservation of Data Integrity: A notable advantage of the Enhanced Algorithm is its lossless nature, ensuring that the decompressed data is an exact replica of the original. This attribute is indispensable in contexts where data accuracy and fidelity are non-negotiable.

**Cons of the Algorithm:**

Performance Scaling: Despite demonstrating consistency in performance across small to medium-sized datasets, the algorithm encountered challenges with larger datasets. The observed scalability issues may restrict its applicability in scenarios demanding compression of extensive data volumes, necessitating further optimization efforts.

Diminishing Returns: The speedup and efficiency analyses revealed diminishing returns associated with increasing parallelism. This finding suggests that additional refinements might be required to fully harness the potential of parallel processing for this algorithm.

**Achievements:**

Parallel Implementation: The study introduces and expounds upon the parallelization of Enhanced Algorithm, an innovative approach that amalgamates Run-Length Encoding, Incremental Prefix Encoding, and base conversion techniques to achieve efficient text data compression. This algorithm holds promise as a valuable asset within the domain of data compression.

Performance Analysis: The comprehensive performance analysis undertaken in this study, encompassing evaluations of speedup, efficiency, and compression rates, offers valuable insights into the algorithm's behavior under varying conditions. These insights serve as a foundation for future research endeavors aimed at optimizing and enhancing the algorithm's performance, particularly concerning larger datasets.

Parallel LZ77

Discussion

Conclusion