DEPARTMENT OF COMPUTER SCIENCE

Masters of Computer Science

CS 6025 Parallel and Distributed Processing I

"***LZ1 Compression***"

Presenters:
Neel Mehulkumar Prajapati
Rahul Mittal

Supervisor:
Eric Aubanel
(*Professor)*
(Faculty of Computer Science, UNB Fredericton)

ABSTRACT

The LZ1 algorithm is a widely used lossless compression algorithm. In this report, we present a serial implementation of the LZ1 algorithm using the Julia programming language. The primary objective of this project is to implement the LZ1 algorithm for serial and parallel execution.

Our methodology involves the creation of a sliding window to find repeated patterns in the input data and replace them with shorter references to the previous occurrences of the same pattern. It is known that the LZ1 algorithm achieves a high compression ratio for text data, making it a popular choice for data compression applications. However, the execution time of the serial implementation can be improved with parallel programming techniques. Our analysis suggests a parallel implementation of the LZ1 algorithm to further improve its performance.

LITERATURE REVIEW

The LZ1 compression algorithm is a classic algorithm that has been extensively studied in the literature. The algorithm was first introduced by Abraham Lempel and Jacob Ziv in 1977 and has since been improved with various extensions. The LZ1 algorithm is based on the sliding window technique, where the algorithm searches for repeated patterns in the input data and replaces them with shorter references to the previous occurrences of the same pattern. The algorithm achieves a high compression ratio for text data, making it a popular choice for data compression applications.

Numerous studies have explored the effectiveness of the LZ1 algorithm and its variants in different contexts. For example, Akbas et al. (2014) compared the performance of various lossless compression algorithms, including the LZ1 algorithm, on genomic data. The study found that the LZ1 algorithm achieved a high compression ratio for genomic data, making it an effective tool for compressing large genomic data sets.

Another study by Liu et al. (2016) proposed a novel approach to improving the compression ratio of the LZ1 algorithm. The approach involves dividing the input data into multiple segments and applying the LZ1 algorithm to each segment separately. The study found that this approach significantly improved the compression ratio of the LZ1 algorithm for large-scale data sets.

Parallel implementation of the LZ1 algorithm has also been extensively studied in the literature. For example, Huang et al. (2015) proposed a parallel implementation of the LZ1 algorithm using multi-core CPUs. The study found that the parallel implementation achieved a significant speedup compared to the serial implementation, making it a more efficient tool for compressing large-scale data sets.

In summary, the LZ1 algorithm is an effective tool for text file compression, achieving a high compression ratio for various types of data. Additionally, parallel implementation of the algorithm can significantly improve its performance, making it a more efficient tool for compressing large data sets.

METHODOLOGY

In this study, we implemented the LZ1 compression algorithm in a serial manner using the Julia programming language. We used a sliding window to find repeated patterns in the input data and replace them with shorter references to the previous occurrences of the same pattern.

The sliding window approach is a fundamental technique used in many lossless data compression algorithms, including LZ1. The idea behind the sliding window approach is to maintain a window of recently encountered data in memory, and then search for matches between this window and the incoming data.

In the case of LZ1, the sliding window is typically implemented as a buffer called the "lookahead buffer" and as the new input data is read into the compressor, it is added to the the lookahead buffer, and the compressor searches for matches between the input data and the contents of the sliding window. When LZ1 finds the best match, it encodes it by using a symbol that represents the position and length of the matched data within the sliding window.

The use of a sliding window in LZ1 has several advantages. First, it allows for efficient matching of long runs of repeating data, which can be represented using a compact variable-length code. Second, it provides a natural way to limit the size of the search space, since matches can only be found within the sliding window. Finally, it allows for incremental compression and decompression of data, since the compressor and decompressor can both maintain their own copies of the sliding window.

Therefore, by carefully managing the size of the sliding window, it is possible to achieve high compression ratios while maintaining reasonable performance.

DESIGN OF SERIAL ALGORITHM

(Compression)

Approach:

We implemented the LZ1 compression algorithm using the Julia programming language. The approach taken to implement the algorithm involved the following steps:

1. Reading input data:

   The input data is read from a file using the read() function in Julia. The input data is stored as a string.

2. Defining parameters:

   The algorithm requires the definition of various parameters, including the option for 12-bit or 16-bit encoding of offset and length of match , the length of the search buffer, and the length of the lookahead buffer. We defined these parameters in our implementation.

3. Encoding the first character:

   The first character of the input data is encoded separately, as it does not have any preceding characters to match with. We encoded the first character using either 12-bit or 16-bit encoding, depending on the option selected.

4. Sliding window approach:

   The algorithm uses a sliding window approach to search for repeated substrings. We implemented this approach by iterating through the input string, and for each character, searching for the longest match in the search buffer.

5. Encoding matches:

   When the longest match is found, it is encoded as an offset-length pair. The offset is the distance between the current character and the start of the match, while the length is the length of the match. We encoded the matches using either 12-bit or 16-bit encoding

Writing compressed data to file:

The encoded data is written to a file using the write() function in Julia. The compressed data is stored as a file.

6. Debugging output:

   We are also creating another file in addition to the compressed output file for debugging. This file shows the offset, length, the character following the match, and the matched characters for each entry in a tangible manner.

DESIGN OF SERIAL ALGORITHM

(Decompression)

The LZ1 decompression algorithm implementation consists of the following steps:

1. Open the compressed data file and the output file:

   We first open the compressed data file and the output file. The compressed data file is the file that contains the compressed data stream, while the output file is the file to which the decompressed data will be written.

2. Read the first byte of the compressed data stream:

   The first byte of the compressed data stream contains the option value using which the decompressor picks the option for the file at hand the length of the offset and match length values.

3. Read the compressed data stream:

   We then read the compressed data stream byte by byte. If the option value is 1, then we read the next three bytes, which contain the offset and match length values. If the option value is 2, then we read the next four bytes, which contain the offset and match length values.

4. Decompress the compressed data:

   Once we have read the offset and match length values, we use them to decompress the compressed data stream. We first check if the offset value is not equal to 0. If it is not 0, we then use it to find the corresponding in the uncompressed data stream. We of the output stream the match length.

5. Write the decompressed data to the output file:

   After decompressing the entire compressed data stream, we write the decompressed data to the output file. Finally, we close the compressed data file and the output file.

DESIGN OF SERIAL ALGORITHM

(Optimization)

Additionally, to optimize the encoding process in 12-bit encoding of integers, we combined the 4 most significant bits of the offset with the 4 most significant bits of the match length to create a 8-bit UInt8. Together with the 8 lower bits, this allowed us to encode both values in three bytes rather than four, reducing the number of bytes required to encode the offset-length pair.

When writing the compressed data to a file, we used the write() function in Julia to write the binary data to the output stream. The write() function takes the output stream and the data to be written as arguments. In our implementation, we wrote the encoded data as UInt8, UInt16 and Char types

To decompress the compressed data, the encoded data is read from the file using the read() function in Julia. The read() function takes the input stream and the type of information to be read as arguments. Once the data is read, it is decoded using the reverse process of the encoding algorithm.

In conclusion, the LZ1 compression algorithm is an effective method for reducing the size of data files. Our implementation using Julia involved reading input data, defining parameters, encoding the data using the sliding window approach, and writing the compressed data to a file using the write() function. Additionally, we optimized the encoding process by combining the 4 most significant bits of the offset with the 4 most significant bits of the max match length.

CHALLENGES FACED

One of the challenges faced during the implementation of the LZ1 compression algorithm using Julia was the potential issue with indexing when working with Unicode strings. Since Unicode characters can have varying byte lengths, it can be difficult to index into a string using an arithmetic operation on indices. This can lead to errors or unexpected results when iterating through the input data or searching for repeated substrings.

To overcome this challenge In Julia, we utilized the eachIndex(), prevind(), and nextind() functions. The eachIndex() function returns an iterator that generates all valid indices for a given collection. This function can be used to safely iterate over the elements of a String, including Unicode strings, without accessing the elements using an explicit index. The prevind() and nextind() functions return the previous and next indices, respectively, in a String These functions are particularly useful when iterating over a String when we need to access elements before or after the current index.

The adoption of these functions made the implementation very difficult, as indexing is relatively easy to work with and debug.

POTENTIAL AVENUES FOR PARALLEL IMPLEMENTATION

Parallelizing an algorithm requires analyzing the dependency between the steps and determining the parts of the algorithm that can be executed concurrently. It is essential to understand the input data structure, processing steps, and output structure to parallelize the algorithm. Below are some potential ways to parallelize the LZ1 algorithm:

1. Parallelizing the search phase:

   In the search phase, the algorithm searches for the longest match string in the search buffer. This step involves comparing substrings in the input string, and the comparison can be done independently for each substring. Therefore, this step can be parallelized by dividing the search buffer into several parts and comparing them concurrently.

2. Parallelizing by dividing the input into multiple independent parts:

   The compression phase involves encoding the matched string and writing the output to the compressed file. This step can be parallelized by assigning each chunk to thread and encoding them in parallel. The details of dimensions of chunks has to be stored in the output file for the decompression to work.

   Parallelizing the decompression phase:

   During the decompression phase, the decompressor selects divided chunks of compressed data and processes them in parallel to speed up the process.

3. Using GPU acceleration:

   The LZ1 algorithm can also be accelerated using GPU computation. The parallel processing capability of GPUs can be used to perform the string comparison and encoding steps in parallel.