Parallel LZ1 Compression Algorithm for Improved Data Compression and Decompression

?

1st Neel Mehulkumar Prajapati
Faculty of Computer Science
University of New Brunswick
Fredericton, Canada
neel.prajapati@unb.ca

2nd Rahul Mittal
Faculty of Computer Science
University of New Brunswick
Fredericton, Canada
rahul.mittal@unb.ca

you should both be 1st!

I. Introduction

Data compression techniques are an essential aspect of modern computational infrastructure due to the vast amounts of data that exist today. There are two primary reasons for compressing data: to save storage space and to save time when transmitting data. Redundancy is a common feature of most data files, and compression algorithms aim to exploit this feature to achieve space savings. Examples of redundancy include frequently occurring character sequences in text files.

This literature discusses various implementations of the LZ1 algorithm for both serial and parallel execution and combinations thereof. The LZ1 is a widely used lossless compression algorithm that replaces repeated patterns in the input data with shorter references to previous occurrences of the same pattern. The LZ1 algorithm achieves a high compression ratio for text data, making it a popular choice for data compression applications. We developed serial and parallel implementations of the LZ1 algorithm and compared their performance. Our analysis, as expected, indicates that the parallel implementation of the LZ1 algorithm significantly enhances its performance.

II. Groundwork for the Following Implmentations

The *Compressor* of LZ1 algorithm takes a string of characters as input and produces a sequence of elements in the groups of three elements. These groups follow a well-defined protocol that facilitates communication between *Compressor* and *Decompressor*, and a suitable sequence of them contains enough information for Decompressor to rebuild the original input.

The formation of the output by compressor involves encoding the matched characters in three element entries **<*Offset, Length of the Match, Following Character*>** that are either stored or transported. As the processing is done to minimise the size of the data, these entries representing data in its compressed form must be encoded efficiently in order to avoid the nullification of the processing effort. The elements "***Offset***" and "***Length of the Match***" represent ⅔ of elements of the set and have upper bounds on their sizes. For this reason, only the absolute necessary number of smallest units of data storage or transportation for a system should be utilised for optimal compression results.

Based on empirical evidence 12-bit and 16-bit representations for 4095 and 65536 unique values were selected for implementation. To optimise 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 8-bits. The remaining 8 lower bits of the two values combined with a recently calculated upper 4-bit combination allows us to encode both values in three bytes rather than four, reducing the number of bytes required to encode the offset-length pair.

For debugging purposes, each implementation has portions of code for generating a separate file alongside the compressed output file, which displays the offset, length, the character following the match, and the matched characters for each entry in an orderly and comprehensible manner.

A. *Sequential Implementations*

The sequential implementation of the LZ1 compression algorithm using the Julia programming language involves a series of crucial steps that enable efficient data compression. The sequential implementation provides a baseline for comparison to other implementations. and serves as a demonstration of the [illegible] principles of the LZ1 algorithm.

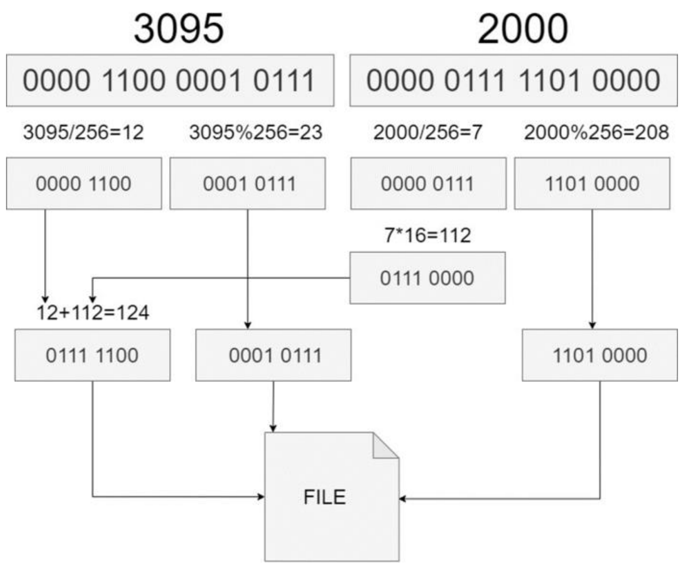

Figure 1: The implementation encodes matched phrases as pairs of offset and length. The encoding process is optimised by using either 12-bit or 16-bit encoding, depending on the user-selected option, to represent the offset and length information.

B. SEQUENTIAL COMPRESSION

The implementation begins its work by reading the input text data from the input file in the form of a string. The input string is converted to an array of characters. Several critical parameters required by the algorithm, including the encoding option for offset and length of match, the length of the search buffer, and the lookahead buffer, are also defined before the processing begins.

To begin the encoding process, the first character of the input data is encoded separately using either 12-bit or 16-bit encoding, depending on the option selected. This is done to have at least one character in the search buffer when the subsequent steps are executed.

The implementation then proceeds to iterate through the input, identifying repetitive patterns and replacing them with references to prior occurrences of the pattern. The LZ1 algorithm employs a sliding window approach, which virtually moves using pointers as the execution proceeds. For every new move of the window after an entry is completed, the search begins again with the unprocessed characters searched inside the search buffer for the longest match. The longest found match is encoded as an < ***Offset, Length of the Match, Following Character*** >, where the offset represents the distance between the first unprocessed character and the start of the longest match, while the length indicates the length of the longest pattern matched. After the encoded data is written to the output file .This process is repeated until the input data has unprocessed characters left in it.

C. SEQUENTIAL DECOMPRESSION

The *Decompressor* begins by reading the first byte from the compressed data file. This byte is used to notify the *Decompressor* of the option chosen during the compression. If the byte is equal to 1, then the *Decompressor* expects 12-bit encoding for the offset and length, and if it is equal to 2, then the *Decompressor* expects 16-bit encoding.

When option is 1, the *Decompressor* reads the next three elements from the file of UInt8 type. The first element contains the upper four bits of the offset and length of match, the second element contains the lower eight bits of the offset and the third element contains the lower eight bits of the length of the match, and the fourth element contains the lower eight bits of the length of the match. The 12-bit values of offset and length of the match are then calculated using these three elements. If the compression option is 2, then the Decompressor reads the next two elements from the file of UInt16. The first element contains the offset and the second is the length of the match.

Once the offset and length of the match have been calculated, the *Decompressor* checks if the offset is non-zero. If the offset is zero, the *Decompressor* simply avoids any processing. However, If the offset is non-zero, the *Decompressor* uses it and the length to generate the original data. If the difference between the offset and length of the match is greater than 0, the *Decompressor* calculates the start and end indices of the substring and develops the original data by concatenating the specified substring from the already processed output string with the output. If the difference is less than or equal to 0, the algorithm develops the original data by forming the substring up to the last character of the processed data and then developing the remaining characters one by one using a loop. These conditions have been placed to avoid developing the original data character by character, when a string concatenation suffices.

Finally, the Decompressor reads the character, i.e., the character that has followed the match in original data, from the compressed data file and adds it to the output. This process is repeated until all the compressed data has been processed.
D below for more information on proofreading, spelling and grammar.

III. Parallel Avenues for Implementation

The LZ1 algorithm works by identifying repeated sequences of characters in the text file and replacing them with references to the original sequence. The algorithm is well-suited for parallelization. It does not require knowledge of the entire input stream to perform compression. This means that each block of data can be compressed or decompressed independently of the others, and the results can be combined at the end to produce the final compressed output. Similarly, the search process in the algorithm can also be parallelized by dividing the search buffer into several parts and comparing them with the input characters under consideration concurrently. Since the process does not involve a write operation, the search within each block of the search buffer is independent of one another and can be processed in parallel. The following section discusses in detail ways in which the algorithm can be parallelized to achieve significant performance gains than serial implementation.

A. *Parallelizing by dividing the input into multiple independent parts*

This approach for compression involves dividing the input data into ***N*** smaller chunks, where ***N*** can be the number of threads ***n*** allocated to the program, function of number of threads ***f(n)*** allocated to the program or a predetermined number. The serial compression algorithm is then run on each individual chunk, compressing them in parallel. The compression process involves encoding the matched characters in three element entries < ***Offset, Length of the Match, Following Character*** >. These entries for individual chunks are stored in Arrays, which are uniquely mapped to the chunks. After initialising the compression processes for all chunks, the primary thread waits for the individual chunks to be processed successfully. This can be done using an explicit synchronisation step or an implicit one that is part of a construct such as parallel *for* loop. When the compression for all chunks is complete and once the synchronisation has been achieved, the compressed chunks are written to the file while maintaining the same sequence as the input file. The first compressed chunk is preceded by supporting information related to the entire compressed file. This information contains the number of entries that each compressed chunk contains along with the placement information of the compressed chunk in the output file. This is done to assist the Decompressor function establish the spread of the individual chunks in its input. Number of chunks ***N*** also needs to be communicated to the *Decompressor*, this piece of information directly follows the option and allows the *Decompressor* to prepare the environment to retrieve the original input from compressed chunks. The multi-threading approach to parallelize the compression process can significantly speed up the compression process, as multiple threads can work on different chunks of data at the same time.

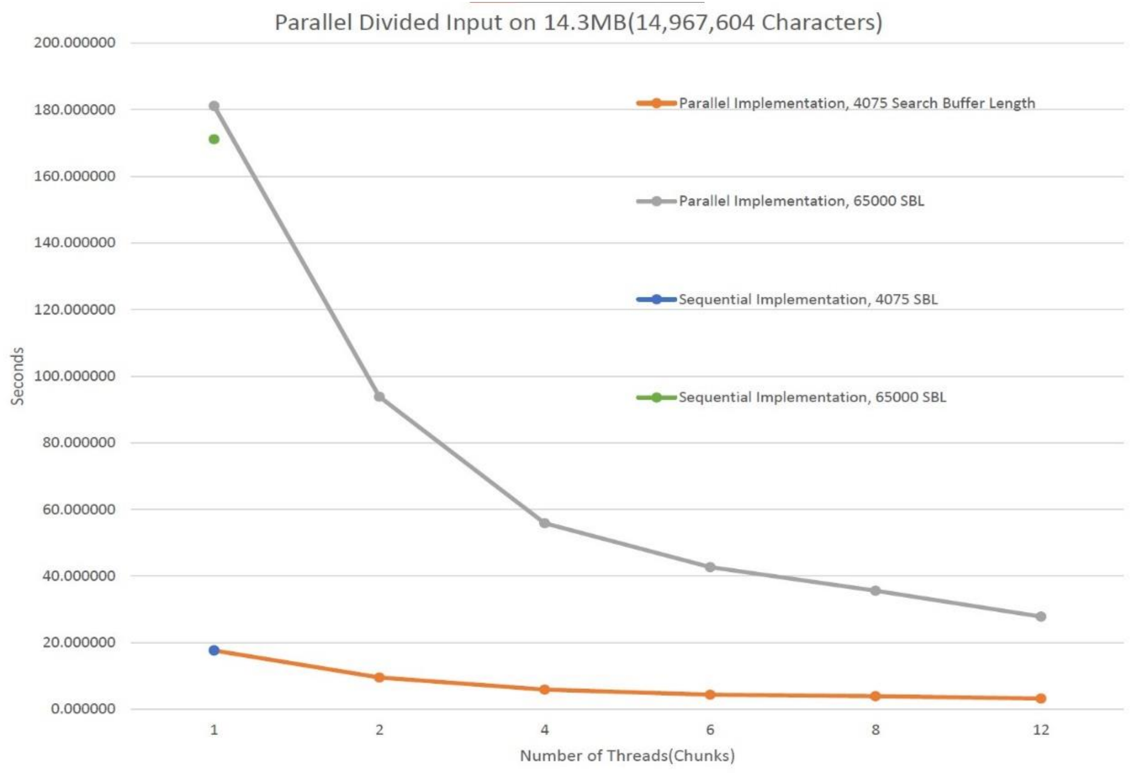

Figure 2: Parallel Divided input on 14.3 MB file size

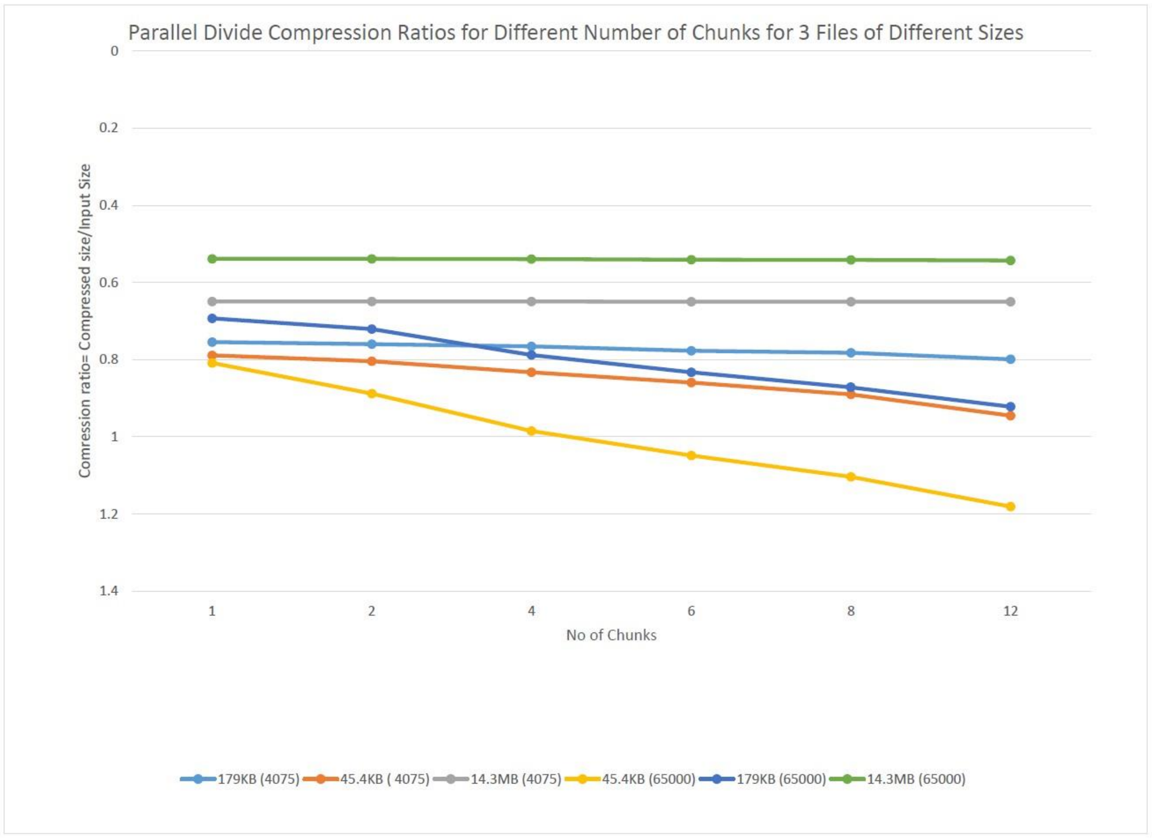

Figure 3: Parallel Divide Compression Ratios for different number of chunks for 3 files of different sizes

B. Parallelizing the search of the longest match

The search phase in the LZ1 compression algorithm involves finding the longest match between the characters of the input being processed starting from the first unprocessed character and a sliding search buffer. This means that the Compressor has to compare characters in the input with characters in the search buffer to find the longest match. If there is a match, the algorithm checks if there is a longer match by comparing the character on the right. This process continues until there is no longer a match or until the match length has reached the maximum length allowed. Due to the repetitive nature, this search process can be computationally expensive, especially for large inputs. To reduce the execution time of the search process, the Implementation parallelised the search by dividing the search buffer into several parts and performing search on them in parallel. This has been done spawning multiple threads to compare different parts of the search buffer simultaneously. By parallelizing the search phase, the algorithm can significantly reduce the amount of time it takes to compress a large input.

C. GPU Implementation

GPUs are highly parallel processing units that are designed to handle a large number of calculations simultaneously. Data compression is a computationally intensive task that can benefit greatly from the use of GPUs. When compressing data, a large number of calculations are performed on the data to be compressed. These calculations can be highly parallelizable, making them well-suited for execution on a GPU. This can result in significant speedup compared to using a CPU alone and this parallel processing capability allows the GPU to compress data much faster than a CPU. This implementation is a GPU parallelized search to find the longest match in the search buffer. It uses the CUDA.jl package to utilise the parallel processing power of the GPU. By taking advantage of multiple GPU cores and comparing each individual element of the search buffer using a thread uniquely associated to it, the comparisons can be done in parallel on thousands of cores. Each part of the search buffer can be searched in parallel, resulting in faster searches of large search buffers.

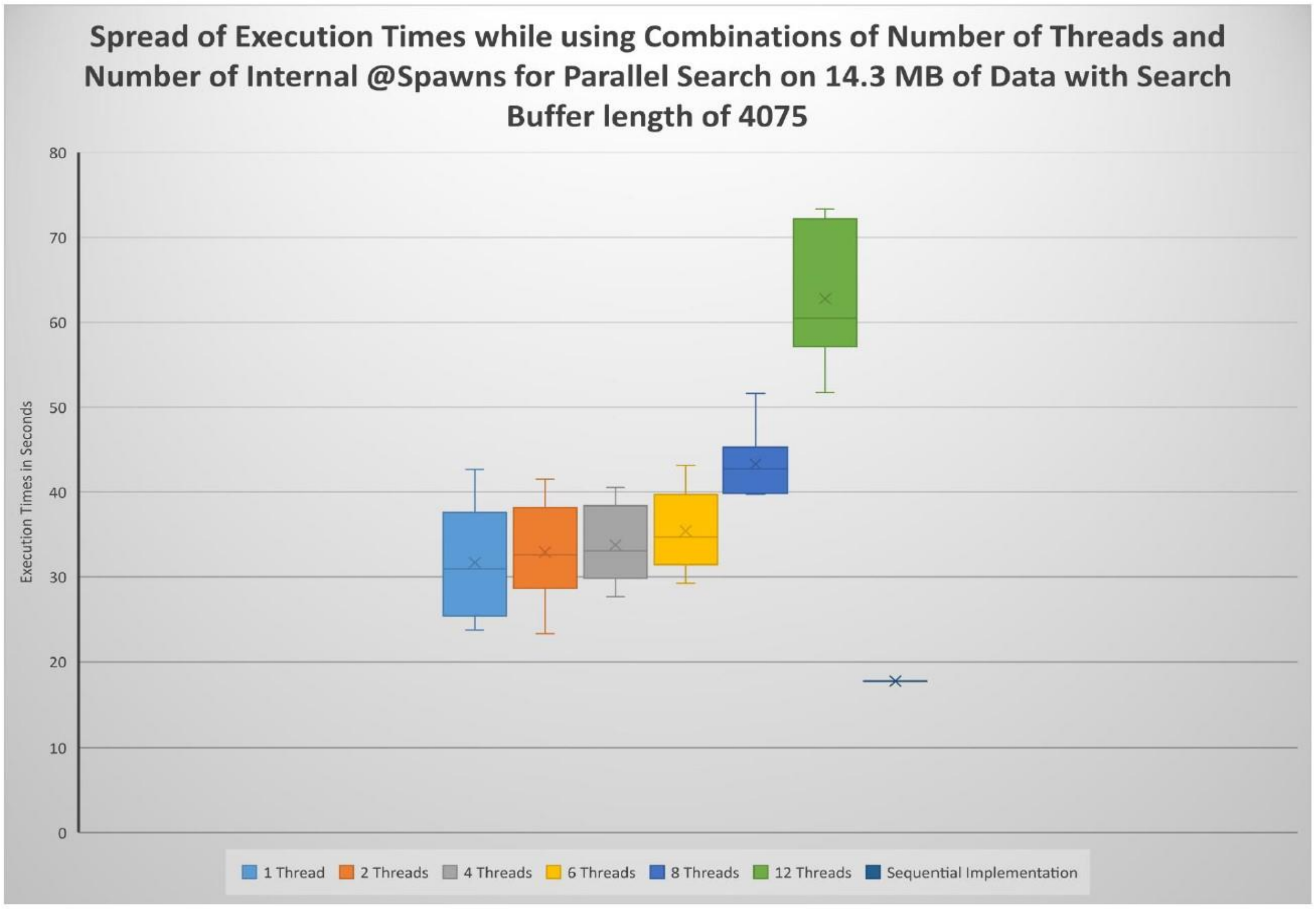

Figure 3: Speed of execution times while using combinations of Number of Threads and Number of Internals @Spawns for parallel Search on 14.3 MB of Data with Search Buffer length of 4075

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

Another issue we faced with ~~S~~strings was a performance penalty, primarily long execution times. This also resulted in difficulty. in testing even while working with small text files. To address this performance issue, we chose to shift to character arrays as primary data storage. This move was motivated by arrays of characters being faster and more memory-efficient than working with strings. Additionally, char arrays provide direct access to individual characters, making it easier to perform operations that require accessing or modifying specific characters. However, working with char arrays also presents its own set of challenges. For instance, char arrays do not provide any built-in support for common string operations.

Thirdly, we also faced several scope issues while working with variables in Julia. To address this, we introduced global variables. However, while comparing “Parallel Divide” and “Parallel Search” algorithms, we noted some of the results were irrational. For instance, not only did parallel divide significantly outperformed parallel search but the latter had an execution time of 212 seconds with 12 threads in comparison. to 180 seconds with 1 thread of parallel divide. We investigated further and noted that introduction of global variables contributed to performance degradation of the program. We then removed the global variables which resulted in significant performance gains. Finally, we encountered degraded performance when using the @threads macro in Julia programming language for parallelizing loops. Specifically, we observed that the performance of our code decreased significantly when using. @threads. To overcome this performance issue, we decided to switch to using the @spawn macro and observed a significant improvement in the performance of our code.

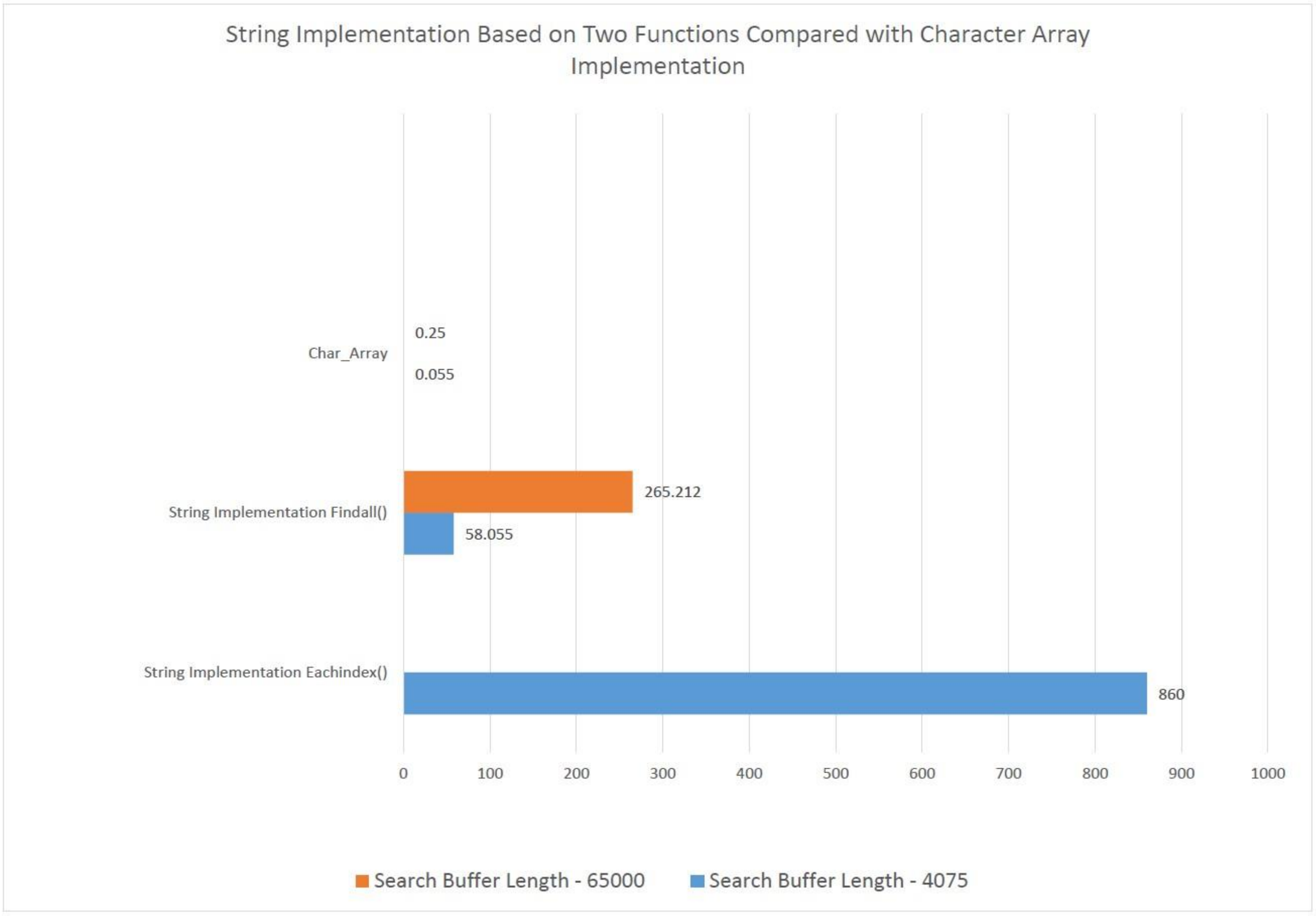

Figure 4: String Implementation based on two functions compared with character array implementation

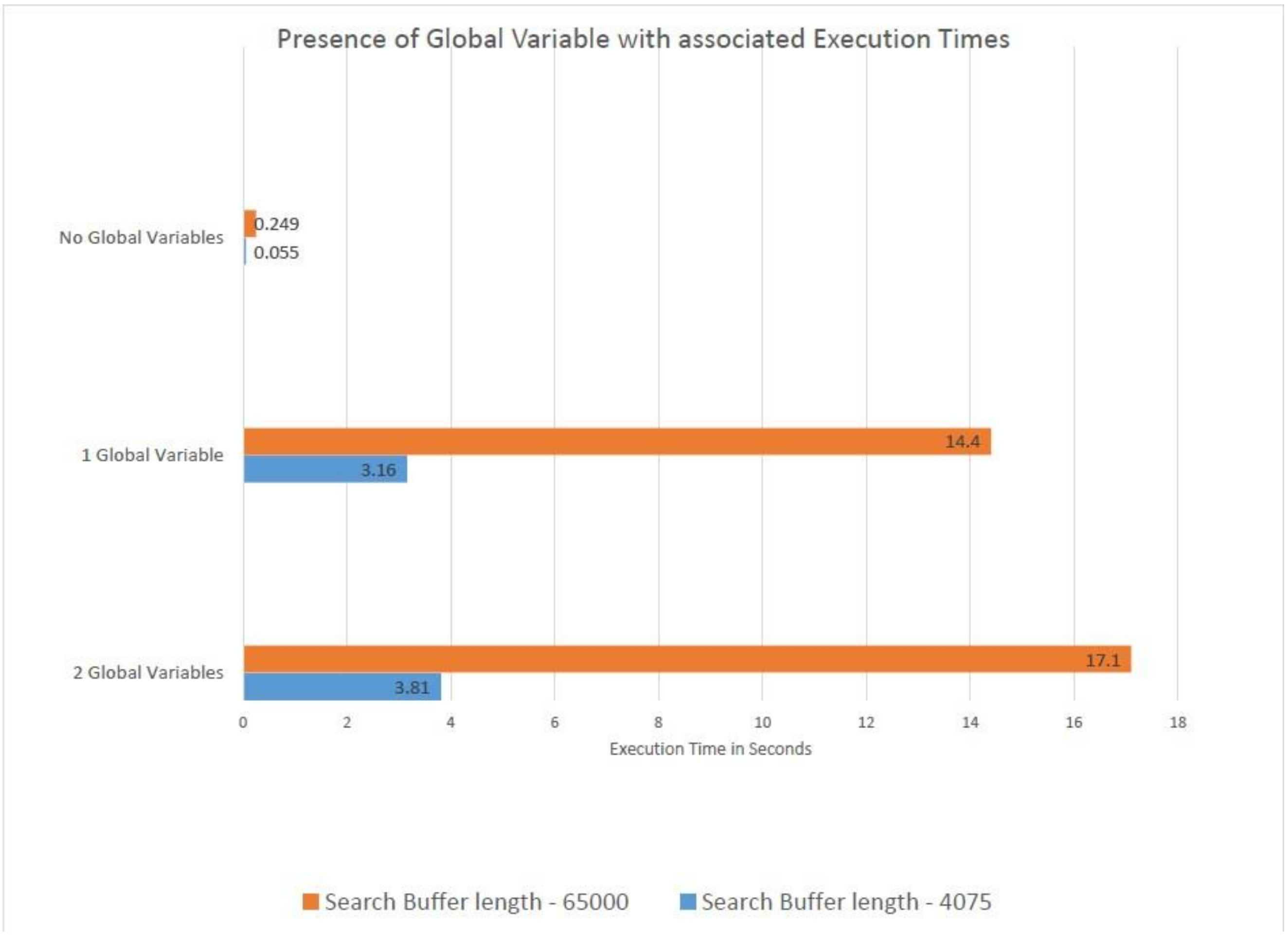

Figure 5: Presence of Global Variable with associated Execution Times

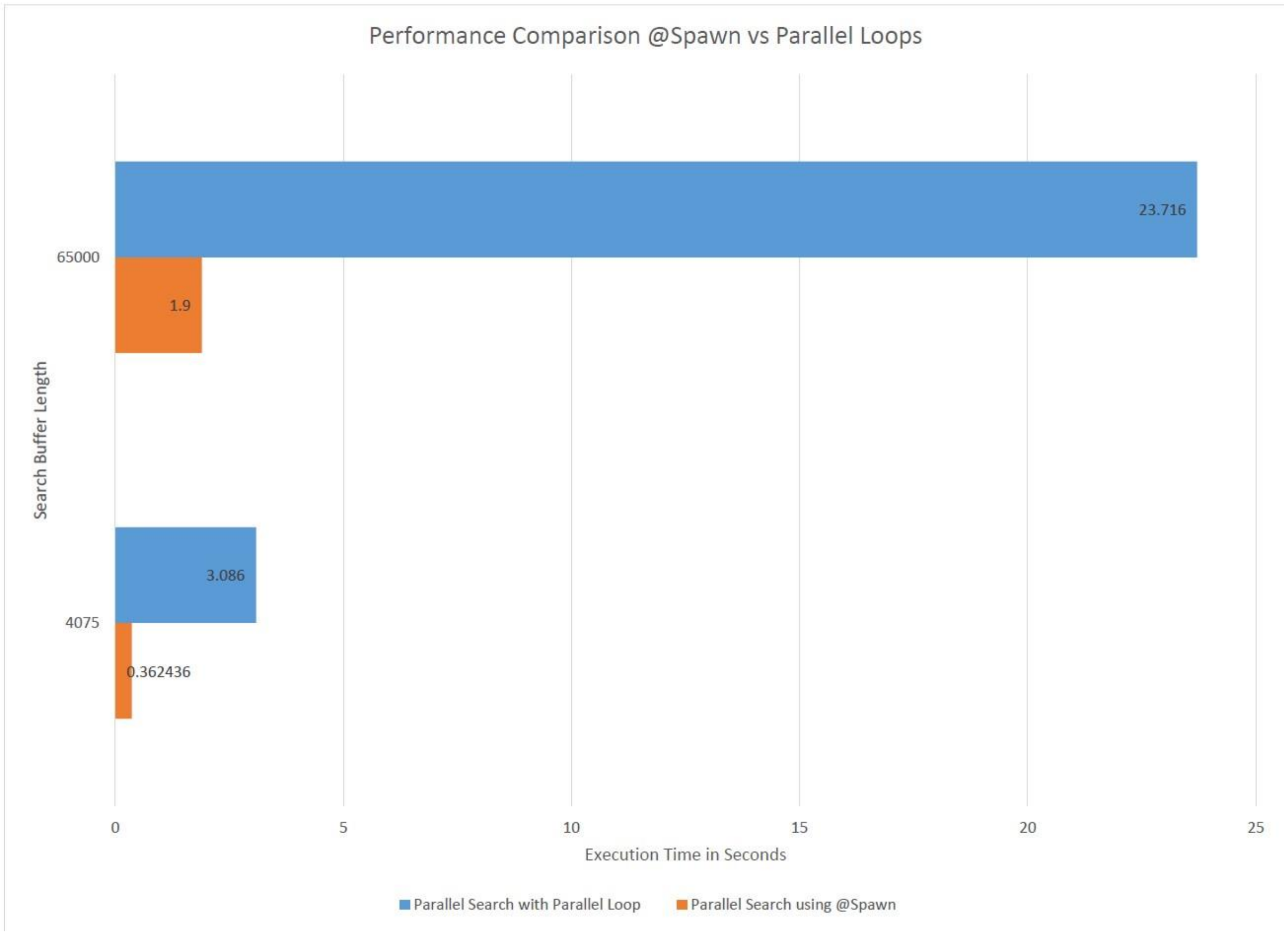

Figure 6: Performance comparison @Spawn vs Parallel Loops

CONCLUSION

In summary the parallel divided input implementation performed significantly better than the other implementations in terms of compression ratio and execution time. Although we achieved our initial targets set at the beginning of the project, there is still room for improvement for increasing the performance of the we have found that overhead of thread spawning mechanism and the parallel loops is so significant that it is difficult to gain any efficiency benefits with those implementations even if they theoretically sound feasible**.**