Accelerating Compressed File Pattern Matching
with Parabix Techniques

by

Xiangyu Wu

B.Eng., Tongji University, China, 2015

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

in the
School of Computing Science
Faculty of Applied Sciences

© Xiangyu Wu 2018
SIMON FRASER UNIVERSITY
Fall 2018

All rights reserved.
However, in accordance with the Copyright Act of Canada, this work may be
reproduced without authorization under the conditions for "Fair Dealing".
Therefore, limited reproduction of this work for the purposes of private study,
research, education, satire, parody, criticism, review and news reporting is likely
to be in accordance with the law, particularly if cited appropriately.
Approval

Name: Xiangyu Wu
Degree: Master of Science
Title: Accelerating Compressed File Pattern Matching with Parabix Techniques
Examinining Committee: Chair: Kay Wiese
Associate Professor

Robert Cameron
Senior Supervisor
Professor
William Sumner
Supervisor
Assistant Professor
Fred Popowich
Examiner
Professor

Date Defended: October 5, 2018
Abstract

Data compression is a commonly used technique in the big data environment. On the other hand, efficient information retrieval of those compressed big data is also a standard requirement. In this thesis, we proposed a new method of compressed file pattern matching inspired by the bitstream pattern matching approach from ICGrep, a high-performance regular expression matching tool based on Parabix techniques. Instead of using the traditional way that fully decompresses the compressed file before pattern matching, our approach handles many complex procedures in the small compressed space. We selected LZ4 as a sample compression format and implemented a compressed file pattern matching tool LZ4 Grep, which showed small performance improvement. Moreover, we proposed a new LZ4 compression algorithm for UTF-8 text files, which substantially improved the speed of compressed file pattern matching for Unicode regular expression, especially for those regular expressions with predefined Unicode categories.

Keywords: Compressed File Pattern Matching; ICGrep; Data Compression; LZ4
Dedication

I want to express my eternal gratitude and respect for my Senior Supervisor Professor Robert Cameron. It is my great honor to become a student of Prof. Cameron. Without his guidance, I cannot come up with all of the ideas of this thesis by myself. Also, his passion for research will also influence me forever.

I want to thank Professor William Sumner for his invaluable assistant and suggestions during the thesis review. Also, I want to thank all of the members and alumni in Prof. Cameron's Open Software Lab, especially for Nigel, for their great help and support in Parabix framework and pipeline.

Finally, I acknowledge my wife and my parents that have provided me with continuous support during my years of study.
# Table of Contents

Approval .............................................................................................................................. ii  
Abstract ............................................................................................................................. iii  
Dedication .......................................................................................................................... iv  
Table of Contents .............................................................................................................. v  
List of Tables ..................................................................................................................... vii  
List of Figures ................................................................................................................... viii  
List of Programs ............................................................................................................... ix  

## Chapter 1. Introduction ................................................................................................. 1  

## Chapter 2. Background ................................................................................................. 3  
2.1. Data Compression ........................................................................................................... 3  
  2.1.1. Overview .................................................................................................................. 3  
  2.1.2. LZ77 ........................................................................................................................ 4  
  2.1.3. LZRW ....................................................................................................................... 5  
  2.1.4. LZ4 .......................................................................................................................... 6  
2.2. Parabix Techniques ........................................................................................................ 11  
  2.2.1. Parabix Overview ...................................................................................................... 11  
  2.2.2. Character Classes .................................................................................................. 12  
  2.2.3. Kernel Programming ............................................................................................... 13  
2.3. Regular Expression Matching in ICGrep .................................................................... 14  
  2.3.1. Regular Expression ................................................................................................. 14  
  2.3.2. Overview of ICGrep ............................................................................................... 15  
  2.3.3. Matching Process in ICGrep .................................................................................. 17  
2.4. Related Work ............................................................................................................... 19  

## Chapter 3. Design Objective ......................................................................................... 21  
3.1. Compressed File Pattern Matching with Character-class Bitstreams ....................... 21  
3.2. Support for Unicode Regular Expression ................................................................... 23  
3.3. Multiplexed Character Class ...................................................................................... 23  

## Chapter 4. Implementation .......................................................................................... 26  
4.1. LZ4 Grep ..................................................................................................................... 26  
  4.1.1. Functionality ............................................................................................................ 26  
  4.1.2. Regular Expression Parsing and Unicode Support .................................................. 27  
  4.1.3. Overall Pipeline Design and Implementation ......................................................... 27  
  4.1.4. Twisted Form ......................................................................................................... 30  
4.2. LZ4 with UTF-8 Improvement .................................................................................... 32  
  4.2.1. Motivation ............................................................................................................... 32  
  4.2.2. Compression Algorithm ......................................................................................... 34  
  4.2.3. UTF-8 LZ4 Grep ..................................................................................................... 38  

## Chapter 5. Performance Evaluation ............................................................................. 40
5.1. Overview .......................................................................................................................... 40
5.2. Performance for Different Regex ..................................................................................... 41
   5.2.1. ASCII Regular Expression ......................................................................................... 41
   5.2.2. Unicode Regular Expression ....................................................................................... 45
5.3. Performance on Different Hardware Configurations ............................................................. 49

Chapter 6. Conclusion and Future Work ................................................................................. 51
6.1. Conclusion ........................................................................................................................ 51
6.2. Future Work ..................................................................................................................... 52
   6.2.1. Optimization for Regular Expression with a Long Sequence ......................... 52
   6.2.2. Multiple Threads Support for LZ4 Grep ................................................................. 54
   6.2.3. Choice of Different Approaches ............................................................................... 54
   6.2.4. Further Improvement of the Compression Ratio for UTF-8 File ......................... 55

References .................................................................................................................................. 56

Appendix A. List of Regex for Performance Evaluation .......................................................... 59
List of Tables

Table 4.1 Main Kernels CPU Cycles of LZ4 Grep for Regex "[0-9]" ......................... 33
Table 4.2 Main Kernels CPU Cycles of LZ4 Grep for Regex "id" .......................... 33
Table 4.3 Compression Statistics of Standard LZ4 and UTF-8 LZ4 .................... 38
Table 5.1 The Hardware Configurations of the Test Machines ......................... 41
Table 5.2 Performance of ASCII Regex Matching in LZ4 (ms/MB in Uncompressed Space) ................................................................. 42
Table 5.3 Main Kernels CPU Cycles/Item and Percentage of LZ4 Grep for ASCII Regex .......................................................................................... 44
Table 5.4 Main Kernels CPU Cycles/Item and Percentage of Fully Decompress Approach for ASCII Regex ............................................................. 44
Table 5.5 Performance of Unicode Regex Pattern Matching in Standard LZ4 Files (ms/MB in Uncompressed Space) ........................................... 46
Table 5.6 Performance of Unicode Regex Pattern Matching in UTF-8 LZ4 Files (ms/MB in Uncompressed Space) ................................................. 46
Table 5.7 Main Kernels CPU Cycles/Item and Overall Cycles Percentage in UTF-8 LZ4 Grep for Unicode Regex (Multiplexed BCC = 7) ............... 47
Table 5.8 Main Kernels CPU Cycles/Item and Overall Cycles Percentage in Normal LZ4 Grep for Unicode Regex (Multiplexed CC = 2) ............ 47
List of Figures

Figure 2.1. Sample LZ77 Compression Process .......................................................... 4
Figure 2.2. Sample LZRW Compression Process .......................................................... 5
Figure 2.3. Overview of LZ4 Format .............................................................................. 7
Figure 2.4 Sample LZ4 Compression Process ............................................................... 9
Figure 2.5 The Format of LZ4 Sequence ...................................................................... 9
Figure 2.6. Sample LZ4 Sequence ............................................................................... 10
Figure 2.7 Sample Bitstream Transpose ..................................................................... 12
Figure 2.8 Parabix Character-class Bitstream ............................................................... 12
Figure 2.9 Three-Level Architecture of ICGrep[2]. Used with Permission of Springer Nature ................................................................. 16
Figure 2.10 Match Regex "a[a-z]*g" in ICGrep ............................................................ 18
Figure 2.11 Match Chinese Character "ni3hao" in ICGrep .............................................. 19
Figure 3.1 Sample Compressed File Pattern Matching in LZ77 .................................. 22
Figure 4.1 Main Data Flow of LZ4 Grep ...................................................................... 28
Figure 4.2 Samples of Twisted Forms for 2 and 3 Bitstreams ........................................ 31
Figure 4.3 Sample LZ4 Matching Conversion ............................................................... 35
Figure 5.1 Performance of ASCII Regex Matching (ms/MB in Uncompressed Space) .............................................................................................................. 43
Figure 5.2 Performance of Unicode Regex Matching in UTF-8 LZ4 (ms/MB in Uncompressed Space) ................................................................. 47
Figure 5.3 Performance of ASCII Regex Matching on Different Hardware (Multiplexed BCC = 4) ......................................................................................... 49
Figure 5.4 Performance of Unicode Regex Matching for UTF-8 LZ4 Files on Different Hardware (Multiplexed BCC = 7) ............................................................................. 50
Figure 6.1 Possible Pipeline for Splitting Regex with Long Sequence ......................... 53
List of Programs

Program 2.1  Simplified Code of Standard LZ4 Compression Algorithm..................... 8
Program 4.1  Simplified Code of UTF-8 LZ4 Compression........................................ 37
Program 4.2  Definition of UTF-8 NonFinal.............................................................. 39
Chapter 1.

Introduction

With the growing amount of data that has been generated every day, data compression has become an important technique. People use data compression for many different purposes, including storing data effectively in a limited storage space, transferring data over limited bandwidth and so on. On the other hand, efficient information retrieval of those compressed big data is also a standard requirement. In some situations similar to compressed database and information retrieval system, the user may only need to process some particular parts of a large compressed file, and how to find and retrieve those parts effectively has become a research problem for many years.

Compressed file pattern matching is a technique that helps to retrieve useful information from compressed file effectively. It was first defined in the year of 1992 by [1] as the task of finding pattern occurrences in compressed sequence without first fully decompressing it. Over the last few decades, researchers have proposed many different approaches for compressed file pattern matching, but none of them has become a mature solution or product for this problem. Most of those approaches have two main limitations. The first limitation is that most of them can only work on a small number of compression algorithms or even just one format that is specially designed for their pattern matching algorithm. However, with the development of the data compression techniques, those compression algorithms or formats have become uncompetitive in terms of compression ratio or compression/decompression speed. And the second limitation is that, although those approaches are talking about "pattern matching", almost all of them can only handle string matching instead of regular expression matching, and the information from their matching results is quite limited. With most of those approaches, we can only know about the number of the occurrences of the target pattern or just a Boolean value that indicates whether the target pattern exists. Few of them can retrieve the location or the nearby contents of every matching.

ICGrep[2] is a regular expression matching tool that has shown a significant performance improvement compared to other conventional tools as GNU grep. Based on the parallel bitstream model of Parabix framework[3], ICGrep proposed a high-
performance way for regular expression matching using parallel bitstream operations, which showed a new solution to the uncompressed file pattern matching problem from a different perspective.

In this thesis, we proposed a bitstream-based method of compressed file pattern matching that can be applied to most of the existing data compression algorithms in the LZ family inspired by the pattern matching approach from ICGrep. Instead of using the traditional way that fully decompresses the compressed file before pattern matching, our method handles many complex procedures in the small compressed space to accelerate the overall performance. We chose LZ4[10], one of the fastest compression formats in the LZ family[4], as a sample compression format and implemented a compressed file pattern matching tool LZ4 Grep, which provided the similar functionalities as other grep tools for compressed files and showed a small performance improvement compared to the traditional approach. Moreover, we proposed a new LZ4 compression algorithm for UTF-8 text files by modifying the standard LZ4 compression algorithm. With some slight improvement on compression ratio, our new compression algorithm substantially improved the speed of compressed file pattern matching for Unicode regular expression, especially for those regular expressions with predefined Unicode categories. All our work in this thesis showed a new solution to the compressed file regular expression pattern matching problem and a new possibility of data storage and information retrieval for big data applications.

The chapters are organized as follows: Chapter 2 provides the basic background information about the data compression techniques, Parabix framework, regular expression matching process of ICGrep and some related work in the compressed file pattern matching problem. Chapter 3 shows our design objective of using multiplexed character-class bitstream technique from Parabix to accelerate the regular expression pattern matching process in compressed files. In Chapter 4, we introduce the detail implementation of our compressed file pattern matching algorithm in LZ4 format. Moreover, we describe how we modified the standard LZ4 compression algorithm for UTF-8 files to improve the performance of Unicode regular expression matching. In Chapter 5, we evaluate the performance of compressed file pattern matching in both standard LZ4 format and the newly proposed UTF-8 LZ4 format. Finally, we conclude this thesis and suggest some potential ways for further improvement in Chapter 6.
Chapter 2.

Background

2.1. Data Compression

2.1.1. Overview

Data compression is a process of transforming some data onto a new form of data, which is smaller than the original data in size. Meanwhile, the compressed data should be able to be converted back to the original data in the lossy or lossless manners. Since every single bit of data for most text files is necessary, we use lossless data compression for them in most cases.

We can classify lossless data compression algorithms in many ways. In this thesis, we will classify them as adaptive and non-adaptive algorithms.

In non-adaptive data compression algorithms, the same input string will always be compressed into the same compression form regardless of their locations in the original uncompressed file. Most non-adaptive data compression algorithms are based on statistical methods. They use variable-size codes for data representation, assigning short codes for input units with high frequency and long codes for those with low frequency. Some well-known algorithms in this category include Huffman Coding[5], Run Length Encoding[6] and Byte Pair Encoding [7].

While in adaptive data compression algorithms, the compressed representation of every input symbol is primarily depended on the context of that symbol. As a result, in this category of algorithms, the same input symbol in different locations will usually be compressed to different representations. Most compression algorithms in the well-known LZ (Ziv-Lempel) family are adaptive algorithms.

In this thesis, we focus on compressed file pattern matching in LZ family algorithms. So, in this section, we introduce some related compression algorithms in detail, which include LZ77, LZRW and LZ4.
2.1.2. LZ77

LZ family is one group of dictionary-based adaptive compression algorithms that rely on the idea of sliding windows. The basic concept of LZ family algorithms comes from LZ77[8] algorithm, which is the first algorithm in that family.

![Sample LZ77 Compression Process](image)

**Figure 2.1. Sample LZ77 Compression Process**

In LZ77, the encoder manages a sliding window, which moves with the input cursor as it encodes the input stream. The window is divided into two parts: the part before the input cursor is the search buffer, while the part after the cursor is the look-ahead buffer. The search buffer acts as a dictionary for current processing data, and the size of it is usually a few thousand bytes; while the look-ahead buffer contains all the current processing data, which is generally around ten bytes. During the encoding process, the encoder scans the search buffer backward to find the longest matching for the look-ahead buffer. Then, it emits the LZ77 token, which is in a format of "(match-offset, match-length, next-character)", where "match-offset" represents the distance between the first character of the emitted matching and current input cursor, "match-length" represents the length of the emitted matching, and the "next-character" is the first character after the emitted matching. If there is no matching in the search buffer, it will just emit token "(0, 0, next-character)".

One example of this process is shown in Figure 2.1. In the figure, during the backward searching process, the encoder first finds the string "def" in search buffer with an offset of 6, which matches the first three characters in the look-ahead buffer. After that, it continues to search backward without finding any other better matching, so it emits token "(6, 3, 'k')", in which "k" is the first character after "def" in search buffer. Then it moves the
input cursor forward by four positions and repeats this process until it encodes all the content of the input file.

As for the decompression algorithm, the decoder needs to maintain an output buffer that is larger than the search buffer. Then, it iterates through all the LZ77 tokens sequentially in their original order, copying the matching in output buffer based on the "match-offset" and "match-length" and appending "next-character" to the output buffer.

The LZ77 algorithm provides the basic idea of the dictionary-based algorithm. It has inspired many other researchers to design and implement new compression algorithms based on this idea, which form the well-known LZ family compression algorithms.

2.1.3. LZRW

In LZ77, since the encoder needs to loop through every single byte inside the search buffer for every input cursor during the match searching process, the encoding process is costly. To solve this problem, researchers proposed many different ways to accelerate this process, and among them, LZRW[9] is one of the simplest and fastest approaches.

![Sample LZRW Compression Process](image)

**Figure 2.2. Sample LZRW Compression Process**
The main improvement of LZRW is using a hash table to find the matching in one step. In LZRW, the length of the search buffer is 4KB, and the length of the look-ahead buffer is 16 bytes. Although the data representation of the output token in LZRW is a little different from that of LZ77, it still contains the same information (match-offset, match-length, and next-character). An integer array of 4096 elements is used as a hash table. During the encoding process, the first three characters of the look-ahead buffer are hashed into a 12-bit number I, which will be used as the index of the array. The encoder fetches the Ith element P of that array and then replaces the Ith element of the array with the current position of the input cursor. After that, the encoder starts searching for matching from position P if P is inside the search buffer. If position P is outside the search buffer, or the length of the matching from position P is less than 3, the encoder will emit token "(0, 0, next-character)"; while if the length of matching is between 3 ~ 16 (the maximum match length is 16 since the look-ahead buffer is 16 bytes), the encoder will emit token "(match-offset, match-length, next-token)".

Figure 2.2 shows an example of this process. In the figure, the encoder fetches the first three characters "def" in the look-ahead buffer, hashes it into a 12-bit number I and then looks up and updates the Ith element in the hash table. The old value of the Ith element in the hash table points to a position with an offset of 6 from the input cursor, which is inside the search buffer. Start from that position, the encoder finds a matching "defk", so it emits token "(6, 4, 'l')" and moves forward the input cursor by five positions.

Although the hash table cannot always find the best matching, [9] reported that it helps to make LZRW ten times faster than a simple implementation of LZ77 class algorithm, and this idea is adopted by many other fast compression algorithms in LZ family.

2.1.4. LZ4

LZ4[10] is one of the fastest lossless data compression format nowadays, which adopted the ideas from both LZ77 and LZRW.
In standard LZ4 compression algorithm, the encoder divides the original file into several file blocks of the same size (block size), and then compresses each file block independently into an LZ4 block. The block size is a parameter in the standard implementation of the LZ4 compressor with a default value of 4MB, while there is no requirement or information about block size in an LZ4 file based on LZ4 format.

Every LZ4 block is either compressed block or uncompressed block. Both types of block begin with a 4-byte block header. The most significant bit of block header indicates whether the block is compressed or uncompressed ("0" means compressed and "1" means uncompressed), and the remaining 31 bits describe the compressed size of the current LZ4 block in little-endian format. If the LZ4 block is uncompressed, the block body will be the same as the original input file block; while if it is compressed, it will contain several sequences, which contain all of the information from the original file block. The structures of different LZ4 blocks are shown in Figure 2.3.

During the file block compression process, the encoder keeps track of two positions ("cursor" and "anchor"), both of which are initialized to the beginning of the file block. The search buffer is before the cursor with a size of 64KB, while the first position of the look-ahead buffer is the same as the cursor. In LZ4, the size of the look-ahead buffer is infinity as long as it does not exceed the end of the file block, which means that LZ4 can support infinite match length.

The match searching process of each file block in LZ4 is similar to that of LZRW. By default, LZ4 also uses an integer array of 4096 elements as a hash table. Instead of using the first three characters, the encoder hashes the first four characters of the search
buffer into a 12-bit number I, fetches and updates the Ith element in the array and gets the position P for a possible matching. If the position P is outside the search buffer, or the match length from position P is less than 4, the encoder will move forward the cursor by one position without emitting any output. Otherwise, the encoder will produce an LZ4 sequence, in which the literal part comes from the input data between anchor and cursor, and the match part constitutes of the match-offset and match-length of the target matching.

Once the output sequence has been produced, the encoder moves forward the cursor by match-length positions and updates the anchor to be the same as the cursor. After that, the encoder repeats this process until the size of the remaining file block is less than 12 bytes. The remaining part will be encoded as the literal part of the last sequence, which does not have the match part. The simplified code of the whole LZ4 compression process is shown in Program 2.1.

```c
void encodeLz4(char* fileBuffer, size_t fileSize){
    size_t cursor = 0;
    while (cursor < fileSize) {
        size_t blockSize = min(DefaultBlockSize, fileSize - cursor);
        char* buffer = fileBuffer + cursor;
        encodeBlock(buffer, blockSize);
        cursor += blockSize;
    }
}
void encodeBlock(char* buffer, size_t size) {
    size_t cursor = 0, anchor = 0;
    while (cursor < size - LastLiteralLength) {
        (matchOffset, matchLength) = findMatch(buffer, cursor);
        if (matchLength < 4) {
            cursor++;
        } else {
            appendSequence(buffer, anchor, cursor, matchOffset, matchLength);
            anchor = cursor = cursor + matchLength;
        }
    }
    // last sequence only contains literal part
    appendLastSequence(buffer, anchor, size - anchor);
}
```

**Program 2.1 Simplified Code of Standard LZ4 Compression Algorithm**

Figure 2.4 shows an example of this process. At the beginning, both of the anchor and cursor point to the character "A". Since there is no matching for all the uppercase characters between "A" and "F", the cursor moves forward until it points to the character "d" while the anchor still points to the character "A". And finally, at the position of the
character "d", the encoder finds a 5-character matching with an offset of 12, so it emits an LZ4 sequence, in which the literal part contains the string "ABCDEF", while the match part encodes the match offset (12) and match length (5). After emitting the output sequence, the cursor and anchor both point to the position of the character "m" and repeat the same process.

Figure 2.4 Sample LZ4 Compression Process

<table>
<thead>
<tr>
<th>LZ4 Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Token</td>
</tr>
</tbody>
</table>

Figure 2.5 The Format of LZ4 Sequence

Since in LZ4 there is no upper bound for either literal length or match length of every single sequence, it uses a concept of extension to represent the literal length and match length. The format of an LZ4 sequence is shown in Figure 2.5. The first byte of the sequence is called "token". The four most significant bits of the token represent base literal length, while the remaining 4 bits represent base match length. After token, there may be an optional "literal length" part. The literal length part will exist only when the base literal length is 15, which is the maximum value that can be represented by 4 bits. To calculate the literal length of the current sequence, we need to scan the literal length part, and this process is called "literal extension". Start from the beginning of the literal length part, if the
value of the current byte is 0xFF, the next byte will still be inside the literal length part; otherwise, the current byte will be the last byte of literal length part. Based on this rule, we can scan until the end of literal length part, and the literal length of the current sequence is the sum of the values of every single byte in literal length part, plus the based literal length from token. After literal length part, there will be the "literal data" part, and the size of the literal data is the same as the literal length of the current sequence. If the base literal length from token is 0, there will be no literal data in current sequence. After literal data, there will be a 2-bytes "match offset" in little-endian format. After match offset, there may be an optional "match length" part. The match length part is quite similar to literal length part, which will exist only when the value of base match length is 15. Also, the decoding process of the match length part is the same as that of the literal length part, which is called "match extension". Since the minimum value of match length in LZ4 is 4, the final amount of match length will be the sum of the values of every single byte in match length part pluses the value of base match length adds 4. As a result, if the base match length in a sequence is 0, there will be not match length part in that sequence, and the value of the match length is 4.

<table>
<thead>
<tr>
<th>LZ4 Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFF 0xFF 0x3 ‘A’ ... ‘B’ 0x56 0x23 0xFF 0x10</td>
</tr>
<tr>
<td>1 Byte Token 2 Bytes Literal Length 273 Bytes Literal Data 2 Bytes Match Offset 2 Bytes Match Length</td>
</tr>
</tbody>
</table>

Figure 2.6. Sample LZ4 Sequence

A sample LZ4 sequence is shown in Figure 2.6. In this sequence, the value of the token is 0xFF, which means that there will be both literal length part and match length part in current sequence. After the token, the first byte of the literal length part is 0xFF, while the section byte of it is 0x3, which means that the size of the literal length part is 2 bytes, and the value of the literal length is 273 (0xF + 0xFF + 0x3). As a result, the literal data part starts from the 4th byte of the current sequence with a size of 273 bytes. After the literal data part, there is a 2-byte match offset with the value of 0x2356 in little-endian format. Similar to the literal length part, the match length part after match offset is also 2 bytes, and the actual amount of the match length in this example is 290 (0xF + 0xFF + 0x10 + 0x4).
As for the decoding process, the decoder maintains an output buffer that is larger than 64KB and decodes all the blocks sequentially. For every block, it decodes all the sequences in their original order. For every sequence, the decoder interprets it as four numbers: literal-start (the begin position of literal data), literal-length (the size of the literal data), match-offset (the distance between the position of the target matching and the output cursor) and match-length (the length of the target matching). Based on the literal-start and literal-length, the decoder appends the literal data onto the output buffer (we call this process "literal copy"); and then based on the match-offset and match-length, the decoder copies the target matching and also appends it to the output buffer (we call this process "match copy").

2.2. Parabix Techniques

2.2.1. Parabix Overview

Parabix[3][11] is a high-performance text processing framework based on LLVM infrastructure[12][13]. It boosts the performance of text processing by utilizing CPU data parallelism. Instead of only working on conventional byte streams, Parabix proposed a new programming model that works on parallel basis bitstreams. The basic idea of this programming model is to convert the sequential byte stream into several parallel basis bitstreams, where each bit in a bitstream represents some particular information of the corresponding byte in the original byte stream. Figure 2.7 shows an example of this conversion, which transposes a UTF-8 Byte Stream into eight parallel basis bitstreams. The transposition between byte stream and basis bitstreams can be processed by the S2P (string to parallel basis bits) and P2S (parallel basis bits to string) logic effectively. After that, the programmers need to convert the sequential byte stream processing logic into bitwise logic in these bitstreams so that it can process bit-block-width data at one time. The bit block width is determined by the width of SIMD register in different CPU architectures. It is 128 bits in SSE2, 256 bits in AVX2 and 512 bits in AVX512. As a result, the bitstream programming model in Parabix can help to fully take advantage of CPU SIMD(Single Instruction Multiple Data) registers and boost the performance.
2.2.2. Character Classes

Character Class (CC) is a set of one or more characters. For example, all the characters between 'a' and 'c' will be a character class, and the size of every single character can be more than one byte. The character-class bitstream is a commonly used bitstream that can be produced by the Character Class Computing Kernel in Parabix. Every single bit in the character-class bitstream is corresponding to one byte in the original text stream. If a codepoint (an Unicode character) in the original text stream belongs to a
character class, the value of the bit corresponding to the last byte of that codepoint in that character-class bitstream will be "1", otherwise it will be "0". Figure 2.8 shows three examples of character-class bitstreams, where "ni3" and "hao" at the end of the UTF-8 text represent two 3-byte Chinese characters respectively. As a result, in the character-class bitstream of codepoint "ni3", only the last position of "ni3" in the UTF-8 text stream is marked as "1".

2.2.3. Kernel Programming

Every Parabix application consists of Stream Sets and Kernels, where Stream Sets can be considered as some particular kinds of memory buffers that contain input and output data, and kernels are the programming units that provide the stream processing logic. Usually, each kernel handles some basic logic by consuming data from input stream sets and producing data to output stream sets, and one or more kernels form the full text-processing pipeline of a Parabix application.

The basic idea of kernel programming is that, the kernel programmers just need to configure some attributes (such as processing rate of each input/output stream) and update some properties (such as the number of the items that has been processed/produced in each input/output stream) for every stream set and kernel, and they only need to implement the logic of processing one stride of data (the stride size is also an attribute of the kernel), and the remaining part of the application will be handled by Parabix. More specifically, Parabix will be responsible for managing input and output memory buffers and calling kernel methods in suitable times based on those attributes and properties to make sure there will always be enough input data to be processed and enough space to produce output in every kernel method call.

There are mainly two kinds of kernels, Pablo kernel and LLVM-IR based kernel.

If the whole kernel logic can be processed by bitwise operations among input and output bitstreams, it will be suitable to be implemented in Pablo Kernel. Pablo is a domain specific language in Parabix with a basic idea of infinite-length bitstreams, and it can be generated by PabloBuilder, which provides a set of builder APIs similar to IRBuilder in LLVM[14]. When developing Pablo Kernel, programmers can always assume that each operation will be applied to an arbitrary number of bitstreams sequentially, and the Parabix
framework will be responsible for handling all the detail implementations about the multiple-bitstream processing logic effectively, including generating SIMD instructions, processing carried data between different bit blocks and so on.

If the kernel logic cannot entirely fit into the infinite-length bitstreams model of Pablo Kernel, programmers can choose the LLVM-IR Based Kernel, in which programmers can use IDISA Builder[15] to generate LLVM IR directly. There are mainly three types of LLVM-IR Based Kernels, which can be used in different situations.

- **SegmentOrientedKernel**: It is the most flexible LLVM-IR Based Kernel in which programmers need to handle most of the stuff by themselves. In this kind of kernels, programmers need to implement the "doSegment" method, which processes and produces at least one stride of data based on its application logic. In the doSegment method, programmers need to update the amount of data that have been processed/produced for every input/output stream set, and they also need to terminate the kernel manually when they have handled all of the input data.

- **MultiblockKernel**: In this kind of kernels, programmers need to implement the "multiblockMethod", which processes a certain number of strides of data. The Parabix pipeline framework will pass the expected number of strides to be processed as a parameter each time it calls the multiblockMethod. Ideally, the kernel method should always handle the corresponding number of strides of data, and the multiblockKernel pipeline will update related attributes of each input/output stream set automatically based on the processing rate.

- **BlockOrientedKernel**: We can treat it as a particular kind of MultiblockKernel whose number of stride is always one. In this kind of kernel, programmers need to implement the "doBlockMethod", which only process one stride of data. If there are any special handling in the last block, programmers also need to implement the "doFinalBlockMethod"; otherwise, Parabix pipeline will use the doBlockMethod for the last block.

### 2.3. Regular Expression Matching in ICGrep

#### 2.3.1. Regular Expression

A regular expression (regex) is a special text string that can define a text pattern. It is commonly used in many search tools including Unix grep, sed and awk. Also, most programming languages also have built-in support of many capabilities of regular expression.

The formal definition of the regular expression[16] is:
Given a finite alphabet \( \Sigma \), the following constants are regular expressions:

1. "\( \emptyset \)" , denoting empty set.
2. "\( \varepsilon \)" , denoting the set containing only the empty string.
3. "\( a \)" , denoting a set containing only character "a".

And given the regular expressions "\( R \)" and "\( S \)":

1. "\( RS \)" is a regular expression representing the concatenation of "\( R \)" and "\( S \)".
2. "\( R|S \)" is a regular expression representing the alternation of "\( R \)" and "\( S \)".
3. "\( R^* \)" is a regular expression representing zero or more occurrences of "\( R \)".

Also, modern regular expression has many extensions to the basic form. One example of those extensions is Unicode regular expression, which is defined in UTS#18 standard[17]. All of those extensions can be represented by the basic definition.

2.3.2. Overview of ICGrep

ICGrep[2] is a high performance Unicode regular expression matching tool based on Parabix Framework. It follows a three-level architecture, which is shown in Figure 2.9. Each level has its own intermediate representation (IR), transformation and compilation modules.
The first level, RegEx AST Level, is mainly responsible for transforming the input regular expression string into simplified abstract syntax tree (AST). In this level, RegEx Parser will parse the input regular expression strings into standard AST. For example, regular expression "ab" will be parsed as "Seq[Unicode_61, Unicode_62]", and "a|b" will be parsed as "Alt[Unicode_61, Unicode_62]". After that, the RegEx Transformations module will simplify the AST by applying several regular expression optimization passes to improve the performance of the regular expression matching in the next two levels. The regular expression optimization passes here are similar to the code optimization passes in LLVM. Every optimization pass accepts a regular expression AST from the previous pass (or from RegEx Parser for the first optimization pass) as input, and produces another optimized regular expression AST as output, which will be the input of the next optimization pass (or the input of RegEx Compiler for the last optimization pass). For example, the

Figure 2.9 Three-Level Architecture of ICGrep[2]. Used with Permission of Springer Nature.
"remove nullable prefix (suffix)" pass can remove unnecessary prefix and suffix of the input regular expression. Assume that the input regular expression for this pass is "a*bcd*", since both of the prefix (a*) and suffix (d*) of that regular expression are optional, the "remove nullable prefix (suffix)" pass will remove both of them and convert the input regular expression to "bc", which will be pass to the next optimization pass. The simplified regular expression AST from RegEx Transformations will be passed to RegEx Compiler to be used in the next level.

The second level, Parabix Level, will transform regular expression AST from the previous level into Parabix IR. Regex Compiler will first transpose input text stream from byte stream to 8 parallel basis bitstreams. Then, it will parse the regular expression AST recursively, converting each node in regular expression AST into Pablo operations on the parallel basis bitstreams in the representation of Parabix IRs. All of the result Parabix IRs of these steps will be passed to the Parabix Transformation module, in which they will be optimized by some general Parabix IR optimization passes such as dead code elimination, common subexpression elimination, and constant folding. After that, they will be passed to Parabix Compiler.

The final level is LLVM level, in which Parabix Compiler will convert the Parabix IRs directly into LLVM IRs. In this level, the LLVM infrastructure also provides many code optimization passes in LLVM compiler to simplify LLVM IRs. In the end, the simplified LLVM IRs will be converted to a runtime function by the Just-in-Time(JIT) engine inside Parabix framework, and ICGrep will use this function to identify the occurrences of the target regular expression in the input file.

2.3.3. Matching Process in ICGrep

The regular expression matching process in ICGrep highly relies on the character classes and infinite-length bitstream model in Pablo.

Besides the basic bitwise operations, ICGrep also proposed some advance operations in Pablo, which include:

- Advance: The Advance operation takes a bitstream as input, and move every bit marker in the bitstream forward by a specified number of positions.
• **MatchStar**: MatchStar can be used to find all matches of character class repetitions (denoted by * in regular expression). This operation is defined as: 
\[
\text{MatchStar}(M, C) = (((M \land C) + C) \oplus C) \lor M.
\]

• **ScanThru[18]**: The ScanThru operation can be used to move the marker of each bitstream to the final position of every multiple-byte character class. This operation is defined as: 
\[
\text{ScanThru}(M, C) = (M + C) \land \neg C.
\]

In order to handle the pattern matching for Unicode, ICgrep also define a helper bitstream NonFinal for UTF-8 encoding:

• **NonFinal** = \(\text{CharClass}((\backslash xC2-\backslash F4)) \lor \text{Advance}(\text{CharClass}((\backslash xE0-\backslash xF4))) \lor \text{Advance}(\text{CharClass}((\backslash xF0-\backslash xF4)), 2)\)

Based on the basic definition of regular expression, the matching rules in ICgrep are shown as follows (where \(m\) represents the bitstream before matching)[2]:

1. \(\text{Match}(m, C) = \text{Advance}(\text{CharClass}(C) \land m)\)
2. \(\text{Match}(m, E_1E_2) = \text{Match}(\text{Match}(m, E_1), E_2)\)
3. \(\text{Match}(m, E_1|E_2) = \text{Match}(m, E_1) \lor \text{Match}(m, E_2)\)
4. \(\text{Match}(m, C^*) = \text{MatchStar}(m, \text{CharClass}(C))\)
5. \(\text{Match}(m, E_1^*) = m \lor \text{Match}(\text{Match}(m, E_1), E_1^*)\)

<table>
<thead>
<tr>
<th>UTF-8 text</th>
<th>amazing interesting astonishing abandon</th>
</tr>
</thead>
<tbody>
<tr>
<td>CC[a]</td>
<td>1.1..................1.............1.1...</td>
</tr>
<tr>
<td>CC[a-z]</td>
<td>1111111111111111111111111111111111111111</td>
</tr>
<tr>
<td>CC[g]</td>
<td>1................1................1...........</td>
</tr>
<tr>
<td>M1=Advance(CC[a])</td>
<td>.1.1..................1.............1.1...</td>
</tr>
<tr>
<td>M2=MatchStar(M1, CC[a-z])</td>
<td>1111111111111111111111111111111111111111</td>
</tr>
<tr>
<td>M3=Advance(M2 ∧ CC[g])</td>
<td>1................1................1...........</td>
</tr>
</tbody>
</table>

Figure 2.10  Match Regex "a[a-z]*g" in ICgrep
Two examples of the matching process are shown in Figure 2.10 and Figure 2.11. In Figure 2.10, the target regular expression "a[a-z]*g" only contains ASCII character classes, all of which are single-byte in UTF-8 encoding. While the Figure 2.11 shows the process of matching Chinese character "ni3hao", each group of 3 characters within the same underscore represents one Chinese character. Since the target regular expression contains multiple-byte character classes, we need to use the helper bitstream NonFinal to move the bit marker from the beginning position to the final position of every character classes.

2.4. Related Work

There is much existing research about compressed file pattern matching over the last few decades, which can be classified into two groups: one group focuses on non-adaptive compression algorithms, while the other group focuses on adaptive compression algorithms.

The research for compressed file pattern matching on non-adaptive compression algorithm is quite mature. Since in non-adaptive compression algorithm, the same input unit will always be compressed into the same compression form regardless of their location in the original uncompressed file. As a result, to search the non-adaptive compressed file, we can convert the target pattern into the compressed form and then use a conventional pattern matching algorithm with some small modification to match the compressed pattern in the compressed file. The only difficulty of this approach is that there may be some extra work to handle the data alignment (or bit alignment) issue in some
compression algorithms based on variable-size codes (such as Huffman encoding). Two early work in this area can be found in [19] and [20]. Basically, both of them designed and implemented their own statistics-based compression formats that relied on the frequencies of the input units, and also made their compressed representation byte-aligned so that they could use any existing pattern matching algorithms directly.

Although the solution to the compressed file pattern matching problem in this kind of algorithms is quite mature, there is no mature product of it. One reason is that generally using a non-adaptive compression algorithm alone is not enough to reach a fair compression ratio compared to most of the commonly used compression formats nowadays. Those state-of-the-art compression formats nowadays usually apply adaptive algorithms only or use combinations of adaptive and non-adaptive algorithms. As a result, if we want to build a sophisticated compressed file pattern matching tool, it is necessary to have a good solution for adaptive compression algorithms.

However, the compressed file pattern matching problem in adaptive compression algorithms is difficult and challenging. There are many limitations in the existing research. For example, almost all of the current studies can only handle string matching instead of regular expression matching, and much research can only work on some specially designed compression algorithms for some specific purposes.

As for the general adaptive compression algorithm, most existing research focused on the LZW algorithm[21], which used to be a popular compression algorithm applied by many applications including Unix Compress. For example, [22] and [23] proposed approaches to rebuild the LZW tries by scanning the whole compressed file, and they check the existence of the target pattern by analyzing LZW tries. However, both of them only designed and investigated their algorithms theoretically without any implementation. While [24] applied Shift-And searching algorithm[25] in LZW format during the compressed file pattern matching, which is about 1.5 times faster than the traditional approach that fully decompresses the compressed file before searching. However, those algorithms are specially designed for LZW, and we are not able to apply similar strategies to any other compression algorithm in the LZ family. Since LZW is not commonly used in text compression in recent years (replaced by gzip[26] and LZ4), there are fewer chances that we can apply those approaches.
Chapter 3.

Design Objective

3.1. Compressed File Pattern Matching with Character-class Bitstreams

Our primary objective in this thesis is to propose a general compressed file pattern matching approach for Unicode regular expression that can be applied to most of the compression algorithms in the LZ family. To introduce our approach, first we define two general decompression processes in most adaptive compression algorithms:

1. Literal Copy: It is a general process that the decoder copies some byte-aligned parts from the compressed file to the output buffer during decompression. For example, appending "next-character" to the output stream during sequence decoding process of LZ77/LZRW, and copying literal data to the output stream during sequence decoding process of LZ4 are both literal copy processes. Also, we call both "next-character" byte from LZ77/LZRW and literal data from LZ4 as "literal data".

2. Match Copy: It is a general process that the decoder copies some byte-aligned parts from previously produced output to the output buffer during decompression. For example, copying the matched part based on match offset and match length in LZ77, LZRW and LZ4 are all match copy processes.

As described in Section 2.1, during the decoding process of LZ77, the contents of the decompressed output are either produced by literal copy or by match copy. Meanwhile, most of the other compression formats in LZ family, including LZRW and LZ4, have the same feature. Ideally, if a compression format has this feature, we are able to decompress character-class bitstreams instead of decompressing the whole file. More specifically, we can calculate all the required character-class bitstreams for the target regular expression in the entire compressed data (or even in literal data only) first. Then, during decompression, we decode the original compressed file for necessary information (such as the positions and lengths of the literal data and target matching), and do literal copy and match copy based on the corresponding parts of compressed character-class bitstreams. In this way, we can produce uncompressed character-class bitstreams and doing pattern matching in ICGrep's approach using these uncompressed bitstreams.
Figure 3.1 shows an example of this approach in LZ77. Assume that in our target LZ77 compressed file we want to match regular expression "[ab]c", which has two character classes ":[ab]" and "c". First, we convert the original LZ77 compressed file into eight basis bitstreams, to which we apply character class computing kernel and produce two character-class bitstreams CC([ab]) and CC(c) in compressed space respectively. Actually, for the two character-class bitstreams, we only care about those parts corresponding to the literal data of the original compressed file. After that, we decode the compressed file with the standard LZ77 decompression algorithm. However, instead of doing literal copy and match copy in the original literal data, we do literal copy and match copy in the two compressed character-class bitstreams, which produces two character-class bitstreams M1 and M2 in uncompressed space. Although we do not have the original uncompressed file, we can still use "Advance(Advance(M1) ∧ M2)" to get the result of pattern matching for the original uncompressed file.

This approach has two possible advantages compared to the traditional method that fully decompresses the compressed file before pattern matching:

1. With this approach, we can handle many complex procedures, including the character-class bitstreams calculation, in the small compressed space instead of the sizeable uncompressed space.

2. Assume that we have N character-class bitstreams, ideally we just need to produce $\frac{N}{8}$ of data compared to that of decompressing the original byte stream. When N is less than 8, we can produce less data, which may accelerate the literal copy and match copy processes.
As a result, ideally our approach can accelerate the overall performance of compressed file pattern matching.

3.2. Support for Unicode Regular Expression

One main feature of ICGrep is its excellent support for Unicode regular expression. As described in Section 2.2.2, Parabix framework supports Unicode by its character-class bitstreams. Ideally, if a compression algorithm guarantees that its literal data always have complete Unicode codepoints and its match copy processes always copy full Unicode codepoints, we can still calculate the Unicode character-class bitstreams and Unicode helper stream NonFinal in compressed space and then apply the approach explained in Section 3.1. However, since most compression algorithms in LZ family are byte-oriented, one multiple-byte Unicode codepoint may be broken into different sequences, and we are not able to compute correct character-class bitstreams in compressed space. To support Unicode regular expression in those formats, we convert every multiple-byte Unicode Character Class into multiple Byte Character Classes (BCCs), and use byte-character-class bitstreams instead of character-class bitstreams.

Similar to the character-class bitstream, the byte-character-class bitstream is a bitstream that each '1' bit indicate the occurrence of the specific byte value in the corresponding position. It will take N byte-character-class bitstreams to represent a Unicode character with a size of N bytes. For example, the Unicode value of Chinese character "ni3" is 0x4F60, which will be represented as 0xE4BDA0 (3 bytes in total) in UTF-8. As a result, we need to use three byte-character-class bitstreams (we call them BCC_E4, BCC_BD and BCC_A0), and we can match the Chinese character "ni3" with "Advance(Advance(BCC_E4) ∧ BCC_BD) ∧ BCC_A0". Since every "1" bit in the byte-character-class bitstream can only represent 1 byte, we do not need to use the helper stream NonFinal when we are doing regular expression matching with byte-character-class bitstream.

3.3. Multiplexed Character Class

Although the idea of decompressing character-class bitstreams is simple and straightforward, there is one problem that cannot be ignored. When we are doing pattern matching for complex regular expressions, it is highly possible that the number of the
character classes or byte character classes are higher than 8. In this case, if we want to apply our approach directly, we need to do literal copy and match copy for more than 8 bitstreams, and the size of which will be larger than the original byte stream. It is possible that in this kind of situations the performance of the decompression process will be slowed down. To solve this problem, we apply the Multiplexed Character Class technique[27] to reduce the number of the character-class bitstreams.

In ICGrep, Multiplexed Character Class is a technique that can be used to find the minimum number of character-class bitstreams from which we can calculate all of the necessary character-class bitstreams for our target regular expression. This technique can significantly reduce the number of the character-class bitstreams, especially when it is large in original regular expression.

To find the multiplexed character classes, first, we need to find a set of mutually-exclusive and collectively-exhaustive character classes that can represent the nominal character classes and input character classes. For example, in regular expression "[a-e]+[c-ho-s]djp", we have 5 character classes:

- CC0  [a-e]
- CC1  [c-ho-s]
- CC2  d
- CC3  j
- CC4  p

Any other character that is not included in those character classes can be represented by [^a-hjo-s], while the minimum mutually-exclusive and collectively-exhaustive character classes are [ab], [f-hoq-s], [ce], [d], [j] and [p].

Then, we use consecutive numbers to encode those character classes. Since we have 7 character classes in total, we can use 3-bit number to encode all of them.

- 000  [^a-hjo-s]
- 001  [ab]
- 010  [f-hoq-s]
Finally, we can compute the multiplexed bitstream based on the encoding result. In our example, the original 5 character-class bitstreams will be compressed to 3 multiplexed character-class bitstreams.

- \( mpx_0 = [ab] \mid [ce] \mid [j] = [abcej] \)
- \( mpx_1 = [f-hoq-s] \mid [ce] \mid [p] = [ce-ho-sp] \)
- \( mpx_2 = [d] \mid [j] \mid [p] = [djp] \)

Based on the multiplexed character-class bitstreams, we can calculate the original character-class bitstreams easily.

- \( CC0 = mpx_0 \& \sim mpx_2 \mid (\sim mpx_0 \& \sim mpx_1 \& mpx_2) \)
- \( CC1 = (mpx_1 \mid mpx_2) \& (\sim mpx_0 \& mpx_2) \)
- \( CC2 = \sim mpx_0 \& \sim mpx_1 \& mpx_2 \)
- \( CC3 = mpx_0 \& \sim mpx_1 \& mpx_2 \)
- \( CC4 = \sim mpx_0 \& mpx_1 \& mpx_2 \)

With the multiplexed character class technique, we can use \( n \) multiplexed character-class bitstreams to represent a maximum number of \( 2^n \) character-class bitstreams. As a result, when we use CC bitstreams, as long as the target regular expression contains less than \( 2^8 \) classes, decompressing bitstreams will normally produce less data than that of fully decompressing the original compressed file; and when we use BCC bitstreams, the maximum number of the multiplexed BCC bitstreams will be 8 since there are 256 possible values for a single byte, and decompressing bitstreams will never produce more data than that of fully decompressing the original compressed file.
Chapter 4.

Implementation

4.1. LZ4 Grep

The compressed file pattern matching algorithm we proposed in this thesis can be applied to most compression algorithms in LZ family, including the LZW algorithm. Although most of the previous research in this area focused on the LZW algorithm, we do not choose it as a sample compression algorithm for two reasons:

1. LZW is a data compression algorithm instead of a compression format. As a result, it is difficult to find a format that can be treated as a standard implementation of LZW.

2. Nowadays LZW is mainly used for image compression (GIF format). As for general data compression or text compression, people prefer to use some other compression algorithms such as Gzip and LZ4.

Meanwhile, LZ4 is one of the fastest compression algorithms in the world that is widely used in many big data applications and search engines, it will be good if we can further improve the performance of searching LZ4 file. As a result, we decide to choose LZ4 as our sample compression format and implement the compressed file pattern matching tool LZ4 Grep based on Parabix framework with our idea of multiplexed character-class bitstreams decompression.

4.1.1. Functionality

The functionality of LZ4 Grep is similar to that of piping the decompression result of an LZ4 file to ICGrep or Unix Grep. It mainly has two modes:

1. Count-only Mode: LZ4 Grep will enter count-only mode when we are using command line parameter "-c". In count-only mode, it prints out a number as the result of grep, which indicates the amount of the lines that have occurrence(s) of the target regular expression in the original uncompressed version of the target LZ4 file. In this mode, Two occurrences of the target regular expression in the same line will still be counted as 1.

2. Scan-match Mode: If we do not add any particular command line parameter, LZ4 Grep will enter scan-match mode, in which it prints out
those lines that have at least one occurrence of the target regular expression.

4.1.2. Regular Expression Parsing and Unicode Support

In LZ4 Grep, Regular expression parsing process is before any other procedures.

As described in Section 2.3.2, the RegEx AST level of ICGrep can parse the input regular expression to its internal regular expression AST and apply regular expression optimization passes. So in our implementation, we use the RegEx AST level of ICGrep directly for regular expression parsing.

Since LZ4 is a byte-oriented data compression algorithm, we use byte character class instead of regular character class to support Unicode regular expression, and the conversion from Unicode codepoint to byte codepoint also happens during the regular expression parsing process. We add a Unicode conversion pass at the end of the regex optimization passes, which converts each the Unicode codepoint in the parsed regular expression AST to a sequence of byte codepoints. Then we collect all of the byte codepoints from regular expression AST to calculate the definition of the multiplexed BCCs.

For example, assume that our target regular expression is "ni3a", where "ni3" is a Chinese character with a Unicode value of 0x4F60 and a UTF-8 representation of 0xE4BDA0, and "a" is just an English character with an ASCII code of 0x61. This regular expression will be parsed as "Seq[Unicode_4F60, Unicode_61]" by the normal regular expression parser of ICGrep, while the Unicode conversion pass will further convert the "Unicode_4F60" to "Seq[Byte_E4, Byte_BD, Byte_A0]" and convert the "Unicode_61" to "Byte_61". As a result, the regular expression AST will become "Seq[Byte_E4, Byte_BD, Byte_A0, Byte_61]". Based on this regular expression AST, we have 5 BCCs (including the BCC for line break) in total, which will be converted to 3 multiplexed BCCs.

4.1.3. Overall Pipeline Design and Implementation

The main logic of LZ4 Grep is invoked after we get the definition of multiplexed BCCs of the target regular expression. Since Parabix framework organizes its
programming logic as a pipeline of different kernels, we also introduce our implementation based on the kernel pipeline.

Figure 4.1  Main Data Flow of LZ4 Grep

The primary data flow of LZ4 Grep is shown in Figure 4.1. The count-only pipeline and scan-match pipeline use the same logic to compute the match bitstream, in which each "1" bit represents an occurrence of target regular expression in uncompressed space. The count-only pipeline will convert the match bitstream to match line bitstream in which there will be only one "1" bit for each line that contains matching, and the popcount of the match line bitstream is the count-only result. As for the scan-match pipeline, since every LZ4 block can be decompressed independently, the scan-match pipeline fully decompresses those LZ4 blocks that contain "1" bit in the match bitstream and then collects each line that contains at least one matching as the result of grepping.
\[ LZ4Bytes = MMapSourceKernel(LZ4\_file\_name) \]
\[ LZ4BasisBits = S2PKernel(LZ4Bytes) \]
\[ CompressedMtxBCC = BCCComputingKernel(LZ4BasisBits) \]
\[ TwistedCompressedMtxBCC = TwistKernel(CompressedMtxBCC) \]
\[ TwistedMtxBCC = LZ4TwistDecompression(LZ4Bytes,TwistedCompressedMtxBCC) \]
\[ MultiplexedBCC = UntwistKernel(TwistedUncompressedMtxBCC) \]
\[ MatchBits = GrepKernel(target\_regex,MultiplexedBCC) \]
\[ LineBreak = GrepKernel(linebreak\_regex,MultiplexedBCC) \] (4.1)

The common part of the two pipelines is shown in (4.1), where we use Camel-Case Naming for the stream sets and underscore naming for the scalar variables. We first use MMapSourceKernel to load the byte data based on the name of the LZ4 compressed file and convert it to 8 basis bitstreams with S2PKernel. Then, based on the target regular expression, we compute the definition of the multiplexed byte character classes (BCCs) and produce the multiplexed BCC bitstreams. One thing to be noted is that, besides the BCCs from target regular expression, we also add the BCC for the line break character (0x0A) during our multiplexed BCCs calculation, so there will be at least one multiplexed BCC. To accelerate the decompression process, we twist those BCC bitstreams into "twisted form" (twisted form will be introduced later), decompressing them with LZ4TwistDecompression kernel and then untwist them back to multiplexed BCC bitstreams in uncompressed space. With the uncompressed multiplexed BCC bitstreams, we use GrepKernel twice to get the match bitstream and the line break bitstream.

\[ MatchLineBits = MatchLineKernel(MatchBits,LineBreak) \]
\[ match\_count = PopcountKernel(MatchLineBits) \] (4.2)

The remaining part of count-only pipeline is shown in (4.2). With the match bitstream and the line break bitstream, the MatchLineKernel calculates and produces a new match line bitstream in which each line that matches the target regular expression will only contain a single "1" bit, and the popcount of the match line bitstream is the result of the count-only pipeline.

\[ ByteStream = LZ4ByteDecompression(LZ4Bytes,MatchBits) \]
The remaining part of the scan-match pipeline is shown in (4.3). In this pipeline, the LZ4ByteDecompression kernel fully decompresses those LZ4 blocks whose corresponding parts in the match bitstream have at least a single "1" bit. For those LZ4 blocks whose corresponding parts in the match bitstream are all "0" bits, empty memory buffer will be produced. And finally, ScanMatchKernel collects and prints those uncompressed lines with at least one matching based on the match bitstream and the line break bitstream.

4.1.4. Twisted Form

During our implementation of LZ4 grep, we noticed that if we decompress multiplexed BCC bitstreams directly in their original form, the literal copy and match copy processes are much slower than that of the byte stream. Since usually the number of the multiplexed BCC bitstreams is more than one, we need to do memory copy for every bitstream individually; when the number of multiplexed BCC bitstreams is N, the number of the load and store instructions during the literal copy and match copy processes may be increased by at most N times when the lengths of literal data and match data are small in every single sequence.

For example, assume that on a 64-bit machine, we use 64-bit registers for data copy, and the number of the multiplexed bitstreams for the target regular expression is 4. Consider a sequence with the match length of 8: if we do match copy in the byte stream, we can transfer all of the 8-byte data with two instructions (one load instruction and one store instruction); however, if we are copying multiplexed bitstreams, we need eight instructions, and each instruction can only load/store 8 bits (1 byte) of data. In this case, although we can produce less data with our bitstream decompression approach, we need to execute more instructions, which may still slow down the overall performance.

To solve this issue, we proposed twisted form for bitstreams, which is more suitable for data copy.

The relation between the twisted form and the regular form of bitstreams is similar to that between parallel bitstreams and byte stream. As described in Section 2.2, the S2P and P2S kernels in Parabix can handle the conversion between 1 byte stream (1 × i8
stream) and eight bitstreams (8 × i1 streams). Similarly, if we have N bitstreams (N × i1 streams), we can convert them to 1 × iN stream, and we call the iN stream as the "twisted form" of the N bitstreams, and we call N as "twist width".

Since N-bit integer does not exist in any CPU architecture for N between 2 and 7, we use the 8-bit integer to represent the twisted form. In order to avoid the memory alignment issue, we only support twist width of 1, 2, 4 and 8. For 3 bitstreams and 5 ~ 7 bitstreams, we use twist width 4 and 8 respectively. Specially, for twist width 1, the twisted form is the same as the original bitstream; and for twist width 8, the twisted form is the same as the byte stream converted by P2SKernel.

Figure 4.2  Samples of Twisted Forms for 2 and 3 Bitstreams

Two examples of twisted forms for 2 and 3 bitstreams are shown in Figure 4.2. For 3 bitstreams, we use twist width of 4 and assume that the Stream3 is all "0" bits when doing the twist conversion.
4.2. LZ4 with UTF-8 Improvement

4.2.1. Motivation

Although our approach can work correctly in LZ4 format, it is still not the best format for our approach. The main problem is that LZ4 does not have any support for Unicode. As described in Section 2.1.4, during the match searching process of the standard LZ4 compression algorithm, the compressor treats the input data as bytes instead of Unicode, and it does not guarantee that every matching will be based on complete Unicode codepoints. As a result, one multiple-byte Unicode codepoint may be broken into different LZ4 sequences, and we are not able to compute correct character-class bitstreams from the LZ4 compressed produced by the standard LZ4 compression algorithm. That is the reason why we have to use byte character classes instead of regular character classes in our implementation of LZ4 Grep.

Meanwhile, modern Unicode regular expression is not limited to the regular expressions with Unicode characters. It also includes many advanced features for matching commonly used Unicode categories, which are defined by the UTS#18 standard[17]. For example, in Unicode regular expression, "p{greek}" and "p{sc=Hira}" can match any Greek and Japanese Hiragana character respectively, while "p{Uppercase}" and "p{LowerCase}" can match uppercase letters and lowercase letters from different languages respectively. Another example is that the widely used character category "\d" (which matches a digit codepoint) is not as simple as regular expression "[0-9]" in Unicode regular expression. Besides the digital number 0 ~ 9 in ASCII, it also matches other Unicode digits from different languages.

Because of the complexity of modern Unicode regular expression, there are two limitations in LZ4 Grep:

The first limitation is that, since we use the byte character class instead of the regular character class, the number of the multiplexed bitstreams that need to be decompressed increases significantly, especially when we use some predefined Unicode categories. For example, a regular expression as simple as "\d" will produce 5 multiplexed byte-character-class-bitstreams. However, if we can use regular character class, we only
need to decompress 2 multiplexed bitstreams (including character class for line break), which will undoubtedly reduce the cost of literal copy and match copy.

The second limitation is that the character-class bitstreams computing process for Unicode character class is complex, especially when we are dealing with the predefined Unicode categories. One main advantage of our approach is that we can handle many complex processes in the small compressed space. However, since we have to use byte character class in LZ4 Grep, we still need to handle a large number of operations about Unicode character-classes bitstream computing in the sizeable uncompressed space. If we can use regular character class, we can handle all of those expensive Unicode character-class bitstream computing processes in the compressed space, which will possibly further improve the overall performance.

Table 4.1 Main Kernels CPU Cycles of LZ4 Grep for Regex "[0-9]"

<table>
<thead>
<tr>
<th>Kernel Name</th>
<th>Item Processed</th>
<th>CPU Cycles</th>
<th>Cycles Per Item</th>
<th>Percentage of CPU Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>S2P</td>
<td>1.06e+09</td>
<td>8.67e+08</td>
<td>0.82</td>
<td>6.94%</td>
</tr>
<tr>
<td>BCC</td>
<td>1.06e+09</td>
<td>6.57e+07</td>
<td>0.06</td>
<td>0.53%</td>
</tr>
<tr>
<td>Twist</td>
<td>1.06e+09</td>
<td>1.36e+08</td>
<td>0.13</td>
<td>1.09%</td>
</tr>
<tr>
<td>TwistDecompression</td>
<td>1.06e+09</td>
<td>9.82e+09</td>
<td>9.26</td>
<td>78.67%</td>
</tr>
<tr>
<td>Untwist</td>
<td>5.15e+09</td>
<td>6.80e+08</td>
<td>0.13</td>
<td>5.44%</td>
</tr>
<tr>
<td>Grep(LineBreak)</td>
<td>5.15e+09</td>
<td>1.77e+08</td>
<td>0.03</td>
<td>1.42%</td>
</tr>
<tr>
<td>Grep(TargetRegex)</td>
<td>5.15e+09</td>
<td>1.99e+08</td>
<td>0.04</td>
<td>1.60%</td>
</tr>
<tr>
<td>MatchLine</td>
<td>5.15e+09</td>
<td>3.06e+08</td>
<td>0.06</td>
<td>2.45%</td>
</tr>
<tr>
<td>Popcount</td>
<td>5.15e+09</td>
<td>1.91e+08</td>
<td>0.04</td>
<td>1.53%</td>
</tr>
</tbody>
</table>

Table 4.2 Main Kernels CPU Cycles of LZ4 Grep for Regex "\d"

<table>
<thead>
<tr>
<th>Kernel Name</th>
<th>Item Processed</th>
<th>CPU Cycles</th>
<th>Cycles Per Item</th>
<th>Percentage of CPU Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>S2P</td>
<td>1.06e+09</td>
<td>9.59e+08</td>
<td>0.90</td>
<td>3.57%</td>
</tr>
<tr>
<td>BCC</td>
<td>1.06e+09</td>
<td>4.05e+08</td>
<td>0.38</td>
<td>1.51%</td>
</tr>
<tr>
<td>Twist(P2S)</td>
<td>1.06e+09</td>
<td>2.89e+08</td>
<td>0.27</td>
<td>1.08%</td>
</tr>
<tr>
<td>TwistDecompression</td>
<td>1.06e+09</td>
<td>1.53e+10</td>
<td>14.41</td>
<td>56.96%</td>
</tr>
<tr>
<td>Untwist(S2P)</td>
<td>5.15e+09</td>
<td>2.74e+09</td>
<td>0.53</td>
<td>10.21%</td>
</tr>
<tr>
<td>Grep(LineBreak)</td>
<td>5.15e+09</td>
<td>3.75e+08</td>
<td>0.07</td>
<td>1.40%</td>
</tr>
<tr>
<td>Grep(TargetRegex)</td>
<td>5.15e+09</td>
<td>6.12e+09</td>
<td>1.19</td>
<td>22.80%</td>
</tr>
<tr>
<td>MatchLine</td>
<td>5.15e+09</td>
<td>3.85e+08</td>
<td>0.07</td>
<td>1.43%</td>
</tr>
<tr>
<td>Popcount</td>
<td>5.15e+09</td>
<td>1.58e+08</td>
<td>0.03</td>
<td>0.59%</td>
</tr>
</tbody>
</table>

Table 4.1 and 4.2 show the CPU cycles of main kernel when matching ASCII digit (with regular expression "[0-9]", which produces 2 multiplexed byte-character-class bitstreams) and Unicode digit (with regular expression "\d", which produces 5 multiplexed byte-character-class bitstreams) in a 1.06GB LZ4 File (5.15GB after decompression) on
an Ubuntu machine with LZ4 Grep, which can show the two limitations clearly. As the number of the multiplexed byte-character-class bitstreams increase from 2 to 5, the average CPU cycles per item for twisted bitstreams decompression process increase from 9.26 to 14.41. Meanwhile, for regular expression "[0-9]", GrepKernel for target regular expression only takes 2.45% of total CPU cycles, while it takes 22.80% for regular expression "\d", which means that conducting Unicode character-class bitstreams computing process for predefined Unicode categories in uncompressed space will cost large performance overhead in GrepKernel.

Since supporting predefined Unicode category is an important feature of Unicode regular expression, the performance bottleneck of it in LZ4 Grep is unavoidable. In order to solve this problem, in this section, we propose a modified version of LZ4 compression algorithm that can produce LZ4 files based on complete UTF-8 codepoints, and we also introduce our UTF-8 LZ4 Grep pipeline, which uses regular character-class bitstreams instead of byte-character-class bitstreams.

### 4.2.2. Compression Algorithm

For simplicity, we can treat every match searching process of the standard LZ4 compression algorithm as a process that tries to find the matching information that contains four numbers:

- **Literal Start**: The begin position of the literal data in the original uncompressed file.
- **Literal Length**: The length of the literal data.
- **Match Offset**: The distance between the target matching and the cursor.
- **Match Length**: The length of the target matching.

And we convert each group of these four numbers into an LZ4 sequence.

The basic idea of our modification is that the prefix or suffix of any matching in an LZ4 sequence can be converted to the literal part of the current LZ4 sequence or the next LZ4 sequence respectively.

Assume that based on the standard LZ4 compression algorithm, we find two continuous groups of match info
The beginning $B_0$ bytes and the last $B_1$ bytes of the first matching are not complete UTF-8 codepoints, and we call them "prefix mismatch" and "suffix mismatch" respectively.

If the remaining fully match part of the first matching is smaller than 4 bytes, we convert the two groups of match information into one group as

\[
< \text{LiteralStart}_0, \text{LiteralLength}_0 + \text{MatchLength}_0 + \text{LiteralLength}_1, \\
\text{MatchOffset}_1, \text{MatchLength}_1 >
\]  

Otherwise, we convert them into two groups as

\[
< \text{LiteralStart}_0, \text{LiteralLength}_0 + B_0, \text{MatchOffset}_0, \text{MatchLength}_0 - B_0 - B_1 >, \\
< \text{LiteralStart}_1 - B_1, \text{LiteralLength} + B_1, \text{MatchOffset}_1, \text{MatchLength}_1 >
\]  

One example of this conversion is shown in Figure 4.3. In this figure, we show 7 2-byte or 3-byte UTF-8 characters that will be parsed as two continuous LZ4 sequences in their byte representation. One pair of matchings that might be found by the standard LZ4 compression algorithm is shown in blue lines, in which the prefix "0xA0" and the suffix "0xD0" are not complete UTF-8 codepoints. Since the length of the remaining fully match part is 6 bytes, we move the prefix "0xA0" to the literal part of the first sequence and the suffix "0xD0" to the literal part of the second sequence so that the matching can be based on complete UTF-8 codepoints.

![Diagram](image)

Sequence Format: <literal-data, match-offset, match-length>

Standard LZ4: <"0xE4 0xBD 0xA0 0xE4 0xBD 0xA1 0xE4 0xBD 0xA2 0xD0 0xA0 0xE4 0xBD 0xA1 0xE4 0xBD 0xA2 0xD0",8,8>,<"0xA2",...>,...

UTF-8 LZ4: <"0xE4 0xBD 0xA0 0xE4 0xBD 0xA1 0xE4 0xBD 0xA2 0xD0 0xA0",8,6>,<"0xD0 0xA2",...>,...

**Figure 4.3 Sample LZ4 Matching Conversion**

Since there is no match part in the last sequence of every LZ4 block, we can use this approach to make sure the matching of every LZ4 sequence is based on complete UTF-8 codepoints. And if we conduct this process during the match searching, we may be
able to find a better matching for uncompressed data started from position \((\text{LiteralStart}_1 - B_1)\).

Meanwhile, we also make sure that each LZ4 block only encodes complete UTF-8 codepoints from the original file. If the last few bytes of an LZ4 block are UTF-8 incomplete, we move it to the beginning of the next LZ4 block. Since LZ4 format does not have any requirement for the uncompressed block size, we can still produce valid LZ4 compressed file even if the uncompressed block sizes of each LZ4 block are different.

If every LZ4 block only contains complete UTF-8 codepoints, and each matching of the LZ4 sequence is based on complete UTF-8 codepoints, we can know that every literal part of the LZ4 sequence will also be complete UTF-8 codepoints as long as the original uncompressed input file is a valid UTF-8 file. We can prove it as follow:

(1) For the first byte of the literal data in an LZ4 sequence: If the sequence is the first sequence of an LZ4 block, the first byte of it will be the first byte of the original uncompressed file block for that LZ4 block, which will always be the first byte of a UTF-8 codepoint. If it is not the first sequence, it will always be the first byte after the previous matching. Since the matching is based on UTF-8 codepoints, the last byte of it will always be the last byte of a UTF-8 codepoint, so the first byte of the next literal data will always be the first byte of the next UTF-8 codepoint. As a result, the first byte of every literal data is always the first byte of a UTF-8 codepoint.

(2) For the last byte of the literal data in an LZ4 sequence: if the sequence is the last sequence of an LZ4 block, the last byte of it will be the last byte of the original uncompressed file for that LZ4 block (since there is no match part in the last sequence), which will always be the last byte of a UTF-8 codepoint. If it is not the last sequence, assume the last byte of that literal data is not the last byte of a UTF-8 codepoint, then the first byte of that UTF-8 codepoint will be in literal part, while the last byte of that UTF-8 codepoint will be in match part, and the match part of the first LZ4 sequence is not based on complete UTF-8 codepoints, which breaks our assumption. So, the last byte of every literal data will always be the last byte of a UTF-8 codepoint.

Based on (1) and (2), all the literal data will be based on complete UTF-8 codepoints.

Based on this idea, we add two verification steps after the standard implementation of the LZ4 compression algorithm. One step adjusts the block size for each LZ4 block to ensure it is based on complete UTF-8 codepoints, while the other step makes sure each matching is also based on complete UTF-8 codepoints.
Program 4.1 shows the simplified code of our UTF-8 LZ4 compression algorithm. The difference (the two verification steps) between our algorithm and the standard LZ4 compressed algorithm (Program 2.1) is shown in blue color.

```
// Return true if c is the first byte of a UTF-8 codepoint
bool isUtf8Initial(char c);

void encodeLz4(char* fileBuffer, size_t fileSize){
    size_t cursor = 0;
    while (cursor < fileSize) {
        size_t blockSize = min(DefaultBlockSize, fileSize - cursor);
        // Make sure block data are complete UTF-8 codepoints
        while(!isUtf8Initial(fileBuffer[cursor+blockSize])) {
            --blockSize;
        }
        char* buffer = fileBuffer + cursor;
        encodeBlock(buffer, blockSize);
        cursor += blockSize;
    }
}

void encodeBlock(char* buffer, size_t size) {
    size_t cursor = 0, anchor = 0;
    while (cursor < size - LastLiteralLength) {
        if (!isUtf8Initial(buffer[cursor])) {
            // For prefix of matching
            ++cursor; continue;
        }
        (matchOffset, matchLength) = findMatch(buffer, cursor);
        while (!isUtf8Initial(buffer(cursor + matchLength))) {
            // For suffix of matching
            --matchLength;
        }
        if (matchLength < 4) {
            cursor++;
        } else {
            appendSequence(buffer, anchor, cursor, matchOffset, matchLength);
            anchor = cursor = cursor + matchLength;
        }
    }

    // last sequence only contains literal part
    appendLastSequence(buffer, anchor, size - anchor);
}
```

Program 4.1 Simplified Code of UTF-8 LZ4 Compression

Table 4.3 show the compression statistics of standard LZ4 and UTF-8 LZ4 in some sample files, where "nci" and "webster" are the only two UTF-8 text files from commonly used data compression benchmark set Silesia Corpus[28], while the remaining files are 3 files of different formats from English Wikipedia Archive[26]. Based on Table 4.3, our
change in the standard LZ4 Compression algorithm has some small improvement on compression ratio (less than 3% on average).

Table 4.3  Compression Statistics of Standard LZ4 and UTF-8 LZ4

<table>
<thead>
<tr>
<th>File Name</th>
<th>Size</th>
<th>Standard LZ4</th>
<th>UTF-8 LZ4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Size</td>
<td>Ratio</td>
<td>Size</td>
</tr>
<tr>
<td>nci</td>
<td>33,553,445</td>
<td>6.057</td>
<td>5,539,348</td>
</tr>
<tr>
<td>webster</td>
<td>41,458,703</td>
<td>2.058</td>
<td>20,149,252</td>
</tr>
<tr>
<td>enwiki-latest-abstract.xml</td>
<td>5,154,462,618</td>
<td>4.860</td>
<td>1,060,554,053</td>
</tr>
<tr>
<td>enwiki-latest-all-titles</td>
<td>1,032,108,310</td>
<td>2.307</td>
<td>447,392,956</td>
</tr>
</tbody>
</table>

4.2.3. UTF-8 LZ4 Grep

Similar to LZ4 Grep for normal LZ4 files, UTF-8 LZ4 Grep also has count-only pipeline and scan-match pipeline, both of which share the same pipeline to produce match bitstream and line break bitstream. And the common part of them is shown in (4.7).

\[
\begin{align*}
LZ4\text{Bytes} &= M\text{MapSourceKernel}(LZ4\_file\_name) \\
LZ4\text{BasisBits} &= S2P\text{Kernel}(LZ4\text{Bytes}) \\
CompressedMtxCC &= \text{CCComputingKernel}(LZ4\text{BasisBits}) \\
TwistedCompressedCC &= \text{TwistKernel}(CompressedMtxCC) \\
TwistedCC &= \text{TwistDecompression}(LZ4\text{Bytes}, TwistedCompressedCC) \\
MtxCC &= \text{UntwistKernel}(TwistedBits) \\
\text{Final} &= \text{GrepKernel(u8_final\_regex, MtxCC)} \\
\text{NonFinal} &= \text{BitStreamNot(Final)} \\
\text{MatchBits} &= \text{GrepKernel(target\_regex, MtxCC, NonFinal)} \\
\text{LineBreak} &= \text{GrepKernel(linebreak\_regex, MtxCC, NonFinal)}
\end{align*}
\]

There are mainly two differences between the UTF-8 LZ4 Grep pipeline shown in (4.7) and the LZ4 Grep pipeline shown in (4.1). The first difference is that, we use Character Class Computing Kernel instead of Byte Character Class Computing Kernel in the compressed space to generate character-class bitstreams, while the second difference is that we also need to compute UTF-8 NonFinal bitstream in compressed space.

As described in Section 2.3.3, UTF-8 NonFinal bitstream is the helper bitstream for UTF-8 regular expression matching, and we show the definition of it from the latest
version of ICGrep in Program 4.2. Since the UTF-8 NonFinal bitstream is not a valid UTF-8 codepoint, it does not meet the requirement of our character class multiplexing algorithm, and we cannot merge it into the multiplexed character class bitstreams directly. We use two different approaches in different situation to solve this problem:

1. We use the UTF-8 Final bitstream during the multiplexing process and convert it to UTF-8 NonFinal bitstream with a not operation in Pablo. Since the character-class bitstream mark the last byte of every Unicode codepoint, the character class for UTF-8 Final is just the character class that contains all of the Unicode codepoints.

2. We calculate the UTF-8 NonFinal bitstream in compressed space, and then twisted it together with the multiplexed class-character bitstreams for decompression.

For each input regular expression, we choose one of the two approaches which produces a twisted bitstreams with a smaller twist width. If the twist widths from the two approaches are the same (which usually happens when the twist width is 4), we choose the second approach since it will increase the overall performance of these situations by 13% on average compared to that of the first approach based on our measurement.

```
Alt[
  Byte(C2-F4),
  Seq[Byte(E0-F4), Byte(80-BF)],
  Seq[Byte(F0-F4), Byte(80-BF), Byte(80-BF)]
]
```

**Program 4.2 Definition of UTF-8 NonFinal**

The remaining part of the count-only pipeline and scan-match pipeline are precisely the same as the pipelines shown in (4.2) and (4.3) respectively, so we do not introduce them again here.
Chapter 5.

Performance Evaluation

5.1. Overview

In this chapter, we evaluate the performance of LZ4 Grep and UTF-8 LZ4 Grep on different types of regular expressions and different hardware configurations.

As described in Section 2.4, for now, there is not a sophisticated compressed file pattern matching tool for LZ4 that can support regular expression, we mainly compare the performance of LZ4 Grep with the approach that fully decompresses the LZ4 file before pattern matching. So we measured the performance of 4 different compressed file pattern matching approaches during our evaluation:

1. Unix Pipe: In this approach, we used the standard implementation of LZ4 decompressor to decompress the LZ4 file fully, and then used Unix pipe to pass the result to ICGrep directly.

2. Fully Decompress: In this approach, we implemented a kernel in Parabix framework that fully decompresses the LZ4 file into a byte stream set with the same algorithm as the standard implementation of LZ4 decompressor, and then we used the stream set to pass the uncompressed data to ICGrep internally.

3. LZ4 Grep: It is the normal LZ4 Grep that uses byte character classes for Unicode regular expression.

4. UTF-8 LZ4 Grep: It is the LZ4 Grep that uses regular character classes. It cannot handle Unicode regular expression correctly in the standard LZ4 file.

At the same time, we also measured the performance results of ICGrep in the original uncompressed files for reference.

Since we wanted to avoid the performance overhead caused by producing a significant amount of text output, we only measured the performance of count-only pipeline of every approach.

For all the overall performance related experiments in this chapter, we matched each regular expression six times in the corresponding version (standard LZ4 compressed
version, UTF-8 LZ4 compressed version or the original version) of every test file and recorded the running time for all of them. However, for each group of six numbers, we ignored the first one and calculated the average and the sample standard deviation of the remaining five numbers as the final result. Since the operating system will cache parts of the target file started from the second time, our final results will focus on the pattern matching itself instead of some other overheads like data loading. We chose UTF-8 text file enwiki-latest-abstract.xml, enwiki-latest-all-titles and enwiki-latest-change_tag.sql as our test files, and the detail statistics of them can be found in Table 4.3.

The detail list of the test regular expressions can be found in Table A.1 and A.2 in Appendix A. Most of the Unicode categories in the test Unicode regular expressions come from the examples of UTS#18 standard. The hardware configurations of our test machines are shown in Table 5.1. We mainly use the results from Ubuntu_AVX512 to analyse the performance of different regular expressions. We also compare the results from the two machines to evaluate the performance on different hardware configurations in Section 5.3.

<table>
<thead>
<tr>
<th>Table 5.1</th>
<th>The Hardware Configurations of the Test Machines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Machine Name</td>
<td>Ubuntu_AVX512</td>
</tr>
<tr>
<td>Operating System</td>
<td>Ubuntu 16.04.4</td>
</tr>
<tr>
<td>Architecture</td>
<td>X86_64</td>
</tr>
<tr>
<td>CPU Model</td>
<td>Intel Xeon</td>
</tr>
<tr>
<td>CPU Max MHz</td>
<td>2900</td>
</tr>
<tr>
<td>CPU Cores</td>
<td>4</td>
</tr>
<tr>
<td>L1 Cache</td>
<td>32KB</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>1024KB</td>
</tr>
<tr>
<td>L3 Cache</td>
<td>8448KB</td>
</tr>
<tr>
<td>Memory</td>
<td>8GB 2666MHZ</td>
</tr>
<tr>
<td>SIMD</td>
<td>AVX512</td>
</tr>
<tr>
<td>Average Disk I/O</td>
<td>1.4GB/s</td>
</tr>
</tbody>
</table>

5.2. Performance for Different Regex

5.2.1. ASCII Regular Expression

For ASCII regular expression, Table 5.2 and Figure 5.1 show the average time of pattern matching by different approaches for every megabyte in uncompressed space. Since in ASCII regular expression, the size of every codepoint is always one byte, the numbers of the multiplexed character classes and byte character classes are always the same. As a result, the behaviors of LZ4 Grep and UTF-8 LZ4 Grep are the same as each
other, and we merge the performance results of these two approaches together. Also, when the number of the multiplexed byte character classes is 1, we only need to decompress the line break bitstream. This situation happens when the target regular expression can be optimized to empty (such as the regular expression "a").

Based on Figure 5.1, the performance of Fully Decompress approach is always similar to that of Unix Pipe approach, so we treat its performance as the standard performance of the approach that fully decompresses the compressed file before pattern matching.

Since in the experiments, we have reduced the overhead of loading the sizeable uncompressed data with the cache provided by the operating system and the fast disk I/O speed of our test machine, matching ASCII regular expression in the original uncompressed file with ICGrep is faster than any other compressed file pattern matching approach.

The performance of LZ4 Grep (and UTF-8 LZ4 Grep) is better than that of Fully Decompress when the number of the multiplexed byte character class (or character class) is less than 5, which means that our approach of decompressing multiplexed character-class bitstream is suitable for those short regular expressions with less than 16 character classes.

<table>
<thead>
<tr>
<th>Multiplexed BCC</th>
<th>ICGrep</th>
<th>Unix Pipe</th>
<th>Fully Decompress</th>
<th>LZ4 Grep</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.23±0.03</td>
<td>1.41±0.06</td>
<td>1.56±0.19</td>
<td>0.99±0.20</td>
</tr>
<tr>
<td>2</td>
<td>0.25±0.08</td>
<td>1.45±0.06</td>
<td>1.57±0.19</td>
<td>1.13±0.21</td>
</tr>
<tr>
<td>3</td>
<td>0.32±0.10</td>
<td>1.57±0.15</td>
<td>1.59±0.19</td>
<td>1.34±0.21</td>
</tr>
<tr>
<td>4</td>
<td>0.40±0.07</td>
<td>1.73±0.12</td>
<td>1.60±0.19</td>
<td>1.42±0.21</td>
</tr>
<tr>
<td>5</td>
<td>0.39±0.07</td>
<td>1.73±0.12</td>
<td>1.58±0.20</td>
<td>1.72±0.27</td>
</tr>
<tr>
<td>6</td>
<td>0.37±0.02</td>
<td>1.70±0.11</td>
<td>1.57±0.19</td>
<td>1.75±0.28</td>
</tr>
</tbody>
</table>
In order to have a deeper understanding of the performance results, we also used the CPU cycle counter provided by Parabix to measure the statistics of CPU cycles of each kernel in Fully Decompress approach and LZ4 Grep approach. Since the CPU cycle counter itself has some performance overhead, we ran the experiment that measures the CPU cycles separately. Although the CPU cycle counter cannot measure the compiling time of each kernel, it is safe to ignore the compiling time since all of our test files are quite large.

We show the statistics about CPU cycles of main kernels in LZ4 Grep approach and Fully Decompress approach in Table 5.3 and 5.4, and an item here means a basic unit (byte for byte stream or bit for parallel bitstream) in the principal input stream of every kernel. In Table 5.3, the S2P, (B)CC, Twist and Decompress Kernels are in compressed space with a smaller number of total items, while Untwist and Grep kernels are in uncompressed space with a more significant number of total items. Similarly, in Table 5.4, the Decompress Kernel is in compressed space, while the S2P and Grep kernels are in uncompressed space. Also, the results of the Grep kernel in both tables include the grepping processes for both line break stream and target regular expression.
One thing to be noted is that Parabix uses memory-mapped file [30] technique for input file loading. Since this technique uses lazy loading, the performance overhead of the data loading will be added to the first kernel that read those data, which are the S2P kernel in LZ4 Grep approach and the Decompress Kernel in Fully Decompress approach. As a result, the CPU cycle results of the Decompress kernel in Table 5.4 also include the performance overhead for data loading, which is around 0.2 cycle per item in the test machine.

Based on these two tables, when the target regular expression is ASCII regular expression, the regular expression matching itself is quite fast, and the Grep kernel does not take a significant amount of CPU cycles (less than 7% in both approaches); Meanwhile, the decompression process in both of these two approaches take the major part of the CPU cycles (more than 65%).

For both of these two approaches, the CPU cycles in uncompressed space are similar. However, when the number of the multiplexed (byte) character classes is less or equal than 4, the CPU cycles/item of Decompress Kernel in Table 5.3 are much smaller than those in Table 5.4, and the sums of CPU cycles/item of those kernels in compressed space (BCC, Twist and Decompress) in Table 5.3 are still smaller than that of the Decompress kernel in Table 5.4, so we can conclude that the performance improvement
of LZ4 Grep for short ASCII regular expression is mainly because we need to decompress less data in that approach. Since we need to use twist width of 8 when the number of the multiplexed bitstreams is higher than 4, the decompression process of these two approaches will be precisely the same, and actually the CPU cycles/item of the Decompress Kernels in Table 5.3 and 5.4 are similar in this situation. While in LZ4 Grep, there is some other performance overhead caused by the BCC and Twist kernels, and that is the reason why the overall performance of LZ4 Grep is slightly slower than the Fully Decompress approach when the number of the multiplexed bitstreams is greater than 4.

5.2.2. Unicode Regular Expression

For Unicode regular expression, the overall performance results of pattern matching by different approaches are shown in Table 5.5 and 5.6. Since we conduct Unicode regular expression pattern matching in Standard LZ4 Files in Table 5.5, we are not able to apply UTF-8 LZ4 Grep. By comparing the results from Table 5.5 and Table 5.6, we know that for the Unix Pipe, Fully Decompress and LZ4 Grep approaches, the performances of the UTF-8 LZ4 files and the standard LZ4 files are quite similar, while the UTF-8 LZ4 format enables the UTF-8 LZ4 Grep approach, which can be faster than other compressed file pattern matching approaches in most cases, especially when the number of the byte character classes in the target regular expression is more than 4. Moreover, when the number of the byte character classes is 7, UTF-8 LZ4 Grep can be even faster than using ICGrep in the original uncompressed file. We show the relationship between the performance results and the number of the byte character classes in Figure 5.2, and a large number of the byte character classes also means the definitions of the Unicode categories in the test regular expressions are complex.
Table 5.5 Performance of Unicode Regex Pattern Matching in Standard LZ4 Files (ms/MB in Uncompressed Space)

<table>
<thead>
<tr>
<th>Multiplexed CC</th>
<th>Multiplexed BCC</th>
<th>ICGrep</th>
<th>Unix Pipe</th>
<th>Fully Decompress</th>
<th>LZ4 Grep</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0.23±0.02</td>
<td>1.41±0.06</td>
<td>1.56±0.19</td>
<td>0.99±0.21</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>0.24±0.01</td>
<td>1.46±0.06</td>
<td>1.58±0.19</td>
<td>1.32±0.22</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0.44±0.03</td>
<td>1.77±0.12</td>
<td>1.64±0.19</td>
<td>1.41±0.22</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>0.46±0.02</td>
<td>1.78±0.13</td>
<td>1.66±0.20</td>
<td>1.78±0.27</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>0.69±0.12</td>
<td>1.90±0.09</td>
<td>1.89±0.22</td>
<td>2.04±0.30</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>8.83±0.04</td>
<td>10.29±0.08</td>
<td>10.11±0.20</td>
<td>10.94±0.73</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>0.41±0.04</td>
<td>1.75±0.12</td>
<td>1.60±0.19</td>
<td>1.41±0.23</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>0.51±0.03</td>
<td>1.81±0.14</td>
<td>1.70±0.19</td>
<td>1.76±0.28</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>0.82±0.02</td>
<td>1.96±0.08</td>
<td>2.02±0.20</td>
<td>2.17±0.29</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>7.29±0.02</td>
<td>8.83±0.10</td>
<td>8.55±0.23</td>
<td>9.86±0.74</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>0.49±0.05</td>
<td>1.78±0.13</td>
<td>1.68±0.19</td>
<td>1.78±0.27</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>0.83±0.04</td>
<td>1.95±0.08</td>
<td>2.02±0.20</td>
<td>2.18±0.30</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>7.36±0.06</td>
<td>8.91±0.10</td>
<td>8.53±0.22</td>
<td>9.71±0.75</td>
</tr>
</tbody>
</table>

Table 5.6 Performance of Unicode Regex Pattern Matching in UTF-8 LZ4 Files (ms/MB in Uncompressed Space)

<table>
<thead>
<tr>
<th>Mtxed CC</th>
<th>Mtxed BCC</th>
<th>ICGrep</th>
<th>Unix Pipe</th>
<th>Fully Decompress</th>
<th>LZ4 Grep</th>
<th>UTF-8 LZ4 Grep</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0.23±0.02</td>
<td>1.41±0.06</td>
<td>1.62±0.28</td>
<td>1.05±0.28</td>
<td>1.05±0.28</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>0.24±0.01</td>
<td>1.45±0.06</td>
<td>1.62±0.28</td>
<td>1.34±0.26</td>
<td>1.34±0.31</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0.44±0.03</td>
<td>1.76±0.12</td>
<td>1.71±0.28</td>
<td>1.45±0.26</td>
<td>1.35±0.31</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>0.46±0.02</td>
<td>1.79±0.15</td>
<td>1.72±0.27</td>
<td>1.84±0.34</td>
<td>1.39±0.33</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>0.69±0.12</td>
<td>1.89±0.08</td>
<td>1.95±0.29</td>
<td>2.10±0.36</td>
<td>1.41±0.33</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>8.83±0.04</td>
<td>10.27±0.08</td>
<td>10.14±0.29</td>
<td>10.99±0.78</td>
<td>2.33±0.60</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>0.41±0.04</td>
<td>1.73±0.11</td>
<td>1.66±0.28</td>
<td>1.44±0.27</td>
<td>1.48±0.27</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>0.51±0.03</td>
<td>1.78±0.13</td>
<td>1.77±0.28</td>
<td>1.81±0.35</td>
<td>1.44±0.28</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>0.82±0.02</td>
<td>1.94±0.07</td>
<td>2.07±0.27</td>
<td>2.23±0.36</td>
<td>1.58±0.30</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>7.29±0.02</td>
<td>8.81±0.10</td>
<td>8.55±0.28</td>
<td>9.90±0.81</td>
<td>2.54±0.57</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>0.49±0.05</td>
<td>1.77±0.13</td>
<td>1.74±0.28</td>
<td>1.83±0.35</td>
<td>1.76±0.34</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>0.83±0.04</td>
<td>1.95±0.08</td>
<td>2.08±0.28</td>
<td>2.23±0.36</td>
<td>1.79±0.35</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>7.36±0.06</td>
<td>8.89±0.12</td>
<td>8.56±0.27</td>
<td>9.76±0.80</td>
<td>2.90±0.66</td>
</tr>
</tbody>
</table>
We can see from Table 5.8 that, as the Unicode regular expression becomes more and more complex, the Grep kernel takes more and more percentage of overall CPU cycles (increases from 3.17% to 85.71%). Since most of our test regular expressions with 2 multiplexed character-class only contain 1 Unicode character class (and line break is the other character class for multiplexing), the Grep kernel only needs to construct one Unicode character-class bitstream and one line break bitstream from the multiplexed byte-character-class bitstream, and the total CPU percentage of the Grep kernel in Table 5.8 means that this process is quite expensive. There are mainly two reasons for it:
1. With the increase of the number of the multiplexed byte character classes, the total number of the original byte character classes grows exponentially, and the number of the steps to construct every single byte-character-class bitstream also increases. For example, if the number of the multiplexed byte-character-class bitstreams is 4, we need to construct at most \(2^4\) byte-character-class bitstreams, and usually, we only need 4 operations to construct every byte-character-class bitstream. However, when the number of the multiplexed character-class bitstreams is 7, we need to construct at least \(2^7\) byte-character-class bitstreams, and usually, we need 7 operations to construct every byte-character-class bitstream, which increases the total number of operations significantly.

2. The definitions of many commonly used Unicode character classes are quite complex, which also increase the performance overhead. For example, if we convert the definition of "d" into a byte sequence, it will contain 54 different Unicode sets, which need to be represented by 54 byte-character-class bitstreams. Even if we have already constructed all of the 54 byte-character-class bitstreams, we still need at least 53 operations to merge those byte-character-classes, which also slow down the overall performance.

We solved this issue successfully in our UTF-8 LZ4 Grep approach by moving the complex Unicode character-class bitstreams computing process from larger uncompressed space to smaller compressed space. If we compare the experiment results for multiplexed BCC = 7 and Multiplexed CC = 2 between Table 5.7 and 5.8, it is obvious that by changing from normal LZ4 Grep to UTF-8 LZ4 Grep, the most significant percentage of overall CPU cycles moves from Grep kernel to CC kernel, which shows that the Unicode character-class bitstreams computing process has been moved from Grep kernel in the larger uncompressed space to CC kernel in the smaller compressed space, and the overall performance also increases dramatically from 10.99ms/MB to 2.33ms/MB based on the result from Table 5.6.

At the same time, since the number of the character classes is always smaller than or equal to the number of the byte character classes, the decompression process in UTF-8 LZ4 Grep should usually be faster than that in normal LZ4 grep, which can also be observed from Table 5.7 and 5.8.

These results showed that, with our UTF-8 LZ4 Grep approach, the LZ4 compressed files produced by our UTF-8 LZ4 compression algorithm are quite suitable for compressed file pattern matching, especially for the Unicode regular expressions with predefined Unicode categories such as "\w", "\p{greek}" and so on.
5.3. Performance on Different Hardware Configurations

In order to evaluate the scalability of our approaches in different hardware configurations, we also ran the performance tests in both of our two test machines and included some of the results in this section, and we still merge the LZ4 Grep and UTF-8 LZ4 Grep together since they are exactly the same for ASCII regular expression.

![Figure 5.3](image)

**Figure 5.3 Performance of ASCII Regex Matching on Different Hardware (Multiplexed BCC = 4)**

When the number of the multiplexed byte character classes is 4, the performance results of ASCII regular expressions matching on different hardware configurations are shown in Figure 5.3. As discussed in Section 5.2, the LZ4 Grep approach has the advantage during the decompression process when the number of the multiplexed byte character classes is no more than 4, which results in performance improvement in all of the two test machines.

At the same time, one thing we need to notice is that, in the Ubuntu_AVX2 machine, the Fully Decompress approach is slightly slower than the Unix Pipeline approach. The main reason is that there are two running processes (lz4 and ICGrep) in Unix Pipe approach. As a result, it can take advantage of multiple-core CPU by nature. However, it is not an excellent multiple-core model since the CPU cycles of these two processes are unbalanced (lz4 decompression takes much more CPU cycles than ICGrep when matching ASCII regular expression). As a result, the multiple-core advantage of Unix Pipe approach is eliminated when we change to a better hardware configuration in Ubuntu_AVX512 machine, which can also show the good scalability of the other three approaches in better hardware.
As for the Unicode regular expression, we show the performance results of different hardware when the number of the multiplexed byte character classes is 7 in Figure 5.4, and the advantage of computing Unicode character classes in compressed space can also be found in the two different machines. Moreover, since the Unicode character-class bitstreams computing process in Parabix can take advantage of wider SIMD registers, significant performance improvement of the UTF-8 LZ4 Grep approach can also be observed when changing from the AVX2 machine to the AVX512 machine. And it also makes our approach more competitive on the future machines with wider SIMD registers compared to the other traditional approaches which do not utilizing the SIMD features.
Chapter 6.

Conclusion and Future Work

6.1. Conclusion

In this thesis, we proposed a new approach for compressed file regular expression pattern matching that can be applied to most compression formats in LZ family based on the idea of character-class bitstream from ICGrep and Parabix techniques. Our approach improves the performance of compressed file pattern matching in two aspects:

1. Compared to the traditional approach that needs to fully decompress the compressed byte data, our approach only needs to decompressed multiplexed character-class bitstreams, which helps to reduce the total amount of data that need to be decompressed and achieve some small performance improvement.

2. Compared to the traditional approach that has to handle the whole pattern matching process in the entire uncompressed data, our approach handles many expensive procedures of pattern matching, such as the Unicode character-class bitstreams computing procedure, in the small compressed space instead of the sizeable uncompressed space, which can accelerate the overall performance significantly.

We chose LZ4 as a sample compression format and implemented a compressed file pattern matching tool LZ4 Grep based on our approach, which showed a small performance improvement for short ASCII regular expression. Moreover, we improved the standard LZ4 compression algorithm for UTF-8 text files so that it can produce a special kind of LZ4 compressed files which is more suitable for Unicode regular expression matching. With the UTF-8 LZ4 files produced by our new compression algorithm, our compressed file pattern matching approach improved the performance substantially, especially for the Unicode regular expressions with predefined Unicode categories.

Compared to other related research in compressed file pattern matching, our approach has better expandability among different compressed formats and provides output results with more useful information. Also, it is one of the first compressed file pattern matching algorithms that support Unicode regular expression. Moreover, since the Unicode category is a necessary feature in modern Unicode regular expression, our UTF-
LZ4 compression algorithm and compressed file pattern matching method also show a new possibility of information storage and retrieval for future big data applications.

6.2. Future Work

6.2.1. Optimization for Regular Expression with a Long Sequence

As discussed in Chapter 5, when the number of the multiplexed character classes is greater than 4, we need to use twist width 8 during the character-class bitstreams decompression process, which makes LZ4 Grep a little slower than the fully decompress approach.

Although the numbers of multiplexed character classes of many commonly used regular expressions are less than 5, one particular situation is that the user may want to search for a long Unicode sentence in some special language, whose number of multiplexed character classes is greater than 4.

One possible solution to this situation is that we might split the sentence into several short character classes sequences, all of which contain less than 5 multiplexed character classes; and then, we match all of those sequences consecutively. Similar to the scan-match pipeline in LZ4 grep, we may decompress the multiplexed character-class bitstreams from every LZ4 block for the first sequence, while started from the second sequence, we may only decompress those LZ4 blocks which have at least one match for the previous sequence.
We show a possible pipeline of this approach in Figure 6.1. Assume that the target regular expression is a Chinese sentence with 20 different characters (5 multiplexed character classes), which is parsed as

\[ \text{Regex} = \text{Seq}[A_0, A_1, \ldots, A_{19}] \] (6.1)

In order to handle this long sequence, we may split it into two character-class sequences as

\[ \text{Regex}_0 = \text{Seq}[A_0, A_1, \ldots, A_{14}] \]
\[ \text{Regex}_1 = \text{Seq}[\text{Regex}_0, A_{15}, A_{16}, \ldots, A_{19}] \] (6.2)

For \text{Regex}_0, we can use normal LZ4 Grep to calculate the match bitstream; when handling \text{Regex}_1, we can ignore those LZ4 blocks whose corresponding part in the match bitstream of \text{Regex}_0 only contains "0" bits (such as LZ4 Block0 in Figure 6.1).
Since a character-class sequence with 15 different character classes can usually filter out significant amount of LZ4 blocks, ideally there will be only a few remaining LZ4 blocks started from the matching process of the second character-class sequence. As a result, this approach may help to improve the performance for compressed file pattern matching for regular expression with a long sequence.

6.2.2. Multiple Threads Support for LZ4 Grep

In our performance evaluation, the Unix Pipe approach utilizes the multiple-core feature of modern CPU by using two processes. However, because of the unbalanced costs of the lz4 decompress process and the regular expression matching process, the performance improvement by this multiple-core model is limited. There might be a better multiple-core model for LZ4 format.

In LZ4 format, every block can be decompressed independently; at the same time, if we use our UTF-8 LZ4 format, we can also handle the character-class bitstreams computing in each block individually. Ideally, we can handle the character-class bitstreams computing procedure and the decompression procedure of different blocks in different threads and then handle the remaining part sequentially in the end. Since the decompression process and Unicode character-class bitstreams computing process are the two most timing-consuming process in both ASCII regular expression and Unicode regular expression, we can expect that this multiple-core model can make much performance improvement. However, since for now, Parabix framework does not support this multiple-thread model, we have only implemented single thread version of LZ4 grep. In the future, we may need to improve the multiple-thread model of Parabix framework so that it can support this approach.

6.2.3. Choice of Different Approaches

As shown in our performance evaluation, our LZ4 Grep and UTF-8 LZ4 Grep approaches have performance improvement in many situations, while for some particular regular expression the performance of the fully decompress approach may be slightly better. Since we implemented all of the three approaches in Parabix framework, ideally, we can build a compressed file pattern matching tool that chooses a suitable approach wisely by analyzing the target regular expression. For example, if the target regular
expression is an ASCII regular expression with more than 5 multiplexed character classes, we should choose the fully decompress approach without using the multiplexed character classes technique.

In order to handle this situation, we may need to design a better way to evaluate the complexity of input regular expression, especially for Unicode regular expression.

6.2.4. Further Improvement of the Compression Ratio for UTF-8 File

In the LZ4 files with our UTF-8 improvement, the literal-length and match-length represent the number of bytes of the literal data and the matching respectively. However, since our modification can only work on valid UTF-8 text files, we may use UTF-8 codepoint as the basic unit, and the literal-length and match-length will mean the number of UTF-8 codepoints of the literal data and the matching respectively.

Since a UTF-8 codepoint may contain multiple bytes, we may reduce the size of some literal length parts and match length parts in this way, which may further improve the compression ratio for UTF-8 text files. However, if we apply this change, the output compressed files will no longer be compatible with the standard LZ4 format.
References


Appendix A.

List of Regex for Performance Evaluation

Table A.1  ASCII Regular Expressions for Performance Evaluation

<table>
<thead>
<tr>
<th>Multiplexed (B)CC</th>
<th>Regular Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a*</td>
</tr>
<tr>
<td></td>
<td>b*</td>
</tr>
<tr>
<td>2</td>
<td>a</td>
</tr>
<tr>
<td></td>
<td>ti</td>
</tr>
<tr>
<td></td>
<td>[a-z], [0-9]{4}-[0-9]{1,2}-[0-9]{1,2}</td>
</tr>
<tr>
<td>3</td>
<td>title</td>
</tr>
<tr>
<td></td>
<td>first</td>
</tr>
<tr>
<td></td>
<td>[a-z][a-z0-9]<em>@[a-z0-9]</em>.[a-z0-9]*</td>
</tr>
<tr>
<td>4</td>
<td>interesting</td>
</tr>
<tr>
<td></td>
<td>astonishing</td>
</tr>
<tr>
<td></td>
<td>([http/ftp</td>
</tr>
<tr>
<td>5</td>
<td>Professional wrestling career</td>
</tr>
<tr>
<td></td>
<td>The Colossal Connection (1989–1990)</td>
</tr>
<tr>
<td></td>
<td>[A-Z][a-z]professional wrestling car[a-zA-Z]er</td>
</tr>
<tr>
<td>6</td>
<td>abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ</td>
</tr>
</tbody>
</table>

Table A.2  Unicode Regular Expressions for Performance Evaluation

<table>
<thead>
<tr>
<th>Multiplexed CC</th>
<th>Multiplexed BCC</th>
<th>Regular Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>\w*</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>\u4F60</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>\p{sc=Hira}</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>\p{sc=Hira}</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>\p{greek}</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>\w</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>\u4E00-\u9FA5\u4F60\u597D</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>\p{sc=Hira}</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>\u4F60\u597D</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>\w</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>\p{sc=Hira}</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>\u4F60\u597D</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>\w</td>
</tr>
</tbody>
</table>