Baltic J. Modern Computing, Vol. 12 (2024), No. 4, pp. 415–430
https://doi.org/10.22364/bjmc.2024.12.4.04

Enhancing Georgian Text Processing: Transliteration Techniques

Archil ELIZBARASHVILI, Magda TSINTSADZE, Manana KHACHIDZE

Ivane Javakhishvili Tbilisi State University, Georgia

archil.elizbarashvili@tsu.ge, magda.tsintsadze@tsu.ge, manana.khachidze@tsu.ge

Abstract. This article explores character encoding within NLP, emphasizing the relevance of UTF-8, especially for Georgian. It examines transliteration as a solution to enhance data processing efficiency, addressing challenges with Georgian characters in NLP tasks. Using Python scripts and shell commands, a comprehensive experiment compares the performance of transliteration and detransliteration. The `sed` and `vim` commands demonstrate superior efficiency, especially in handling larger files. The results highlight the consistent advantage of transliterated texts over originals with Georgian characters, with shell commands processing them approximately 18 times faster. Emphasizing the importance of method selection based on task nature and data volume, the article underscores the practical advantages of shell commands, the importance of disk buffering and cache for optimizing data reading and writing processes, especially when dealing with cached data. Overall, the study contributes valuable insights into character encoding complexities, offering practical considerations for optimizing NLP data processing, particularly in languages like Georgian.

Keywords: Character encoding, Georgian language, Transliteration, Text processing, NLP, UTF-8.

1 Introduction

Natural language processing (NLP), a prominent field within computer science and a subset of artificial intelligence, encompasses a diverse array of tasks and applications, including text classification, named entity recognition, sentiment analysis, machine translation, question answering, text summarization, language generation, document classification, information extraction, and text clustering. These tasks are closely tied to handling extensive datasets, where digital information is presented in various languages. UTF-8 stands out as a crucial encoding

system for representing digital content, especially in languages like Georgian. This encoding system facilitates comprehensive character and script processing, using additional bytes to store Georgian characters, thereby increasing data volume in computers. Given that data volume significantly impacts storage and processing resources in terms of time and accuracy for most NLP tasks, exploring alternatives to reduce data volume becomes paramount for enhancing NLP efficiency in problem-solving.

Transliteration emerges as a promising solution for optimizing data processing and addressing NLP-related challenges across diverse languages (IBM, 1954; Jennings, 2010). Notably, considering that all three alphabets in the Georgian language are represented by 3 bytes in UTF-8 encoding, the role of transliteration becomes particularly crucial for efficient handling of Georgian texts.

In the domain of NLP tasks, it is standard practice to work with extensive text files, where datasets often include compilations of books, articles, websites, and various written sources. KartuVerbs (Elizbarashvili et al., 2023), an ongoing project, which we are developing concurrently, serves as an illustrative instance of this practice. This project aims to construct a linked database of inflected forms for Georgian verbs, facilitating the learning process for non-native speakers of Georgian. In this initiative, structured textual data is employed. KartuVerbs currently contains more than 5 million inflected forms related to more than 16 000 verbs; there are more than 80 million links in the base. To enrich the database with additional Georgian verbs, we employ text mining techniques on diverse Georgian textual resources, including books, articles, websites, and other written materials. In essence, the project involves the management of exceptionally large Georgian text files, some of which incorporate elements in Latin characters as well. These files are characterized by sizes in the order of gigabytes.

Various shell commands are employed to process these extensive files for various purposes. The testing procedures are performed iteratively, and each instance of processing large files consumes a significant amount of time. Adding to the complexity, Georgian characters are three bytes in size, whereas Latin ASCII characters occupy only one byte. Consequently, the handling process is prolonged. In an effort to expedite experiments, we are exploring methods to convert Georgian texts into Latin ASCII characters. To achieve this, we have developed several approaches for transliterating Georgian text into Latin. Before diving into the details of our experiment let us shortly consider the character encoding terminology and then pay attention to the peculiarities of UTF-8 encoding for Georgian language.

1.1 Character Encoding Terminology

The act of writing serves as a fundamental tool for recording and transmitting speech, employing graphic signs known as glyphs (Strizver, 2011). This study explores the evolution of character encoding systems (Jennings, 2010) and their impact on information representation in digital systems, with a focus on the challenges posed by languages like Georgian (Tsintsadze et al., 2022; Soselia et al., 2018).

The terminology employed in this context is crucial for understanding the intricate processes involved. A character, as the smallest semantic unit, can encompass letters, numbers, punctuation marks, or mathematical symbols. A character set, comprising abstract characters, involves two components – the repertoire and its numerical correspondence. A coded character set further refines this abstraction, assigning each character a specific numerical value or code point. The character code point plays a pivotal role. This concept is essential for understanding the code space – a range of numerical values associated with character encoding. The character repertoire, an abstract set awaiting numerical mapping, contributes to the comprehensive character encoding framework.

Mapping, a one-to-one correspondence of characters with numerical values, is a crucial aspect of character encoding (Mackenzie, 1980). Encoding itself is the mechanism through which code points are converted into sequences of octets – eight-bit units in computers. The code unit, representing the length of the record in character encoding, can vary, with 7-bit, 8-bit, and 16-bit units being common. Some encoding schemes utilize variable lengths for specific characters.

The concept of a grapheme, the smallest unit corresponding to a phoneme, adds a linguistic layer to the discussion. Meanwhile, glyphs, concrete visual representations of letters, introduce the visual dimension. The variety of glyphs associated with a single character, exemplified by different fonts, highlights the complexity of visual representation. Lastly, ligatures, formed by combining multiple letters and signs, add an additional layer of complexity to character representation. These ligatures convey multiple sounds within a single visual entity, exemplifying the nuanced nature of character encoding systems.

In the intricate realm of digital communication and information processing, the fundamental building blocks lie in the nuanced terminology surrounding character encoding. At the core of this exploration are terms that define how characters, the smallest semantic units, are represented and transmitted in the digital landscape. The mechanism of character encoding, a crucial link in this chain, is dissected to reveal its role in converting code points into sequences of octets (eight-bit sequences, defining a standard unit of digital information in computing) – shaping how letters and symbols find their place in memory (Kozierok, 2017; Glossary of Unicode Terms, 2024).

Understanding these terminologies and concepts is essential for grasping the intricacies of character encoding and its role in information processing within digital systems. The study also touches upon the historical developments in character encoding, from the 7-bit ASCII standard to the emergence of Unicode and ISO/IEC 10646 (ISO/IEC, 2020) as universal standards.

2 UTF-8 Encoding for Georgian

In the contemporary digital landscape, Unicode (The Unicode Standard, 2024) has become ubiquitous, covering a rich array of 149,813 characters[1], including

[1] `https://www.unicode.org/versions/stats/charcountv15_1.html`

support for the Georgian language. With 294 scripts[2], both modern and historical, Unicode plays a vital role in facilitating communication across diverse languages and writing systems.

As digital texts find their home in computer systems, various encodings come into play, representing the code points of characters in binary notation. The encoding landscape features different bit lengths, but the prevalence of 8-bit encodings is notable for capturing the richness of the world's scripts. Prominent among these are ASCII-compatible (Mackenzie, 1980), variable code unit length encodings like UTF-8, UTF-16, and GB18030.

Delving into UTF-8 encoding, it transforms a 2-byte Unicode entry for a Georgian character into a 3-byte representation as follows.

```
1st Byte 2nd Byte 3rd Byte 4th Byte  Nof Free Bits Max Exp. Unicode Value
0xxxxxxx                                         7  007F hex (127)
110xxxxx 10xxxxxx                         (5+6)=11  07FF hex (2047)
1110xxxx 10xxxxxx 10xxxxxx              (4+6+6)=16  FFFF hex (65535)
11110xxx 10xxxxxx 10xxxxxx 10xxxxxx  (3+6+6+6)=21  10FFFF hex (1,114,111)
```

Taking the Georgian character "ა" (a) as an example (U+10D0), its hexadecimal value "10D0" translates to the binary sequence 00010000 11010000. When converted to UTF-8, this binary sequence aligns with the rules, producing 11100001 10000011 10010000. A practical illustration with the Georgian letter "ა" (a) validates the encoding process. By creating a file containing only the letter "ა" (a) and examining its binary representation, we observe a matching sequence of 11100001 10000011 10010000. This reaffirms that the UTF-8 encoding accurately captures the essence of character representations, providing a standardized and efficient means of handling diverse scripts in digital environments.

In the encoding system, especially UTF-8, plays a crucial role in accurately representing and storing Georgian characters in digital environments. The unique characteristics of the Georgian script, with its 2-byte Unicode entries, require careful consideration in encoding to ensure efficient storage and seamless digital communication.

The UTF-8 encoding, with its ability to handle a wide range of characters and scripts, proves indispensable for Georgian. Despite the additional byte needed for the representation of Georgian characters, the system provides a standardized and reliable method for storing and transmitting textual information. The example of the Georgian letter "ა" (a, U+10D0) demonstrates the effective conversion from its Unicode value to the corresponding UTF-8 binary sequence, showcasing the encoding system's role in preserving the integrity of Georgian characters.

As the digital landscape continues to evolve, and global communication becomes increasingly diverse, a robust and flexible encoding system like UTF-8 ensures the accurate and efficient representation of Georgian language and culture within the broader context of digital communication.

[2] https://www.worldswritingsystems.org/

Each letter within the three Georgian alphabets occupies 3 bytes in memory when encoded in UTF-8 (Unicode, 2023).

2.1 Georgian Character Size Challenge

All Georgian characters are transformed in 3-byte sequences when Using UTF-8 encoding. Multi-byte characters generally require more resources than one-byte characters. In terms of storage, each multi-byte character typically requires more than one byte to represent, compared to one-byte characters. This means they occupy more storage space. Regarding processing, the CPU must determine the encoding of each character to interpret it correctly. For multi-byte characters, this involves checking multiple bytes, which can be more computationally intensive. In terms of memory, as an application processes more multi-byte characters, it may need to allocate additional memory for temporary data structures or intermediate results. Caching can further consume memory, particularly when dealing with large amounts of multi-byte text.

When working with Georgian texts, it's common to encounter a mix of Georgian letters along with Latin or other foreign characters. This situation involves handling both single-byte and multi-byte characters simultaneously.

If Georgian characters weree ncoded as a single byte, working with text would be less computationally intensive. To accomplish this, specific transliteration rules (Mammadzade, 2018) would need to be adapted for both Georgian and Latin characters. The State Language Department of Georgia has established guidelines for transliterating the Georgian language's sound system into Latin characters[3]. However, this rule does not support bijective transliteration, meaning that once Georgian text is transliterated into Latin, it cannot be accurately converted back to Georgian on a one-to-one basis. Along with this, if the original Georgian text contains Latin characters, those characters will be entirely lost in the twice-transliterated text.

3 Methodology

Our approach to addressing the issue of Georgian character size involves isolating the non-Georgian parts of the text and transliterating only the Georgian portions of the text while leaving the rest unchanged. To accomplish this, we use a custom transliteration table along with a marker system. Modern Georgian has 33 letters, so our transliteration table maps these characters to both lowercase and uppercase basic Latin characters. Before applying this table, we first employ markers to identify patterns in the text that contain basic Latin characters, which are part of our transliteration table. We assume that the marker character does not appear in the text. This process ensures that non-Georgian characters in the original text remain unchanged, while transliteration and detransliteration are applied only to the unmarked segments. Detransliteration is

[3] https://enadep.gov.ge/uploads/Guidelines_for_Latin_Transliteration_of_the_Sound_System_of_the_Georgian_Language_.pdf

technically a form of transliteration. It is the reverse process of transliteration. While transliteration converts text from one script to another, detransliteration converts the transliterated text back to its original script. By mapping each Georgian character to a unique basic Latin equivalent and vice versa, we can achieve a bijective conversion process. Unlike Oriental languages such as Japanese, Korean, Chinese, and Taiwanese, which employ variable-length encoding schemes (Spencer, 2001), Georgian uses a fixed-length encoding scheme, making the proposed conversion approach both feasible and effective.

When applying our method to a large volume of mixed characters, it's essential to ensure that the tools used for transliteration and detransliteration are both time-efficient and consume as few computational resources as possible. To select the optimal solution, our objective is to develop various conversion methods whether character-based, line-based, or buffer-based, test these processes, and compare the performance of different techniques across a range of file sizes.

For the character by character transliteration process, a script or program can be created using a variety of programming languages; we opted for Python in our case. For line by line transliteration, we used the shell command `sed`, which is well-suited for processing text line by line. It's a stream editor for searching, replacing, and manipulating text and `sed` can be used to perform transliteration by using regular expressions and appropriate substitution patterns. For buffer-based transliteration, we selected the shell command `vim`, as Vim is a text editor capable of handling text in buffers. Vim's ex commands offer a powerful way to execute various text manipulations, including transliteration. Although the `tr` shell command is specifically designed for character transliteration, we can't use it because it works with single-byte characters and when dealing with multi-byte characters, like Georgian characters, it can lead to unexpected results.

To confirm that the original version is identical to the transliterated and subsequently detransliterated version, we use the GNU implementation of the `cmp` command, which compares two files byte by byte. Alternatively, the `diff` and `comm` commands can also be used. They compare files line by line and identify specific changes.

4 Experimental Setup

For the experiments, we utilized files of varying sizes: 10 MB, 100 MB, 1 GB, and 5 GB. These files predominantly consist of Georgian characters, occasionally interspersed with fragments of Latin characters. The text was extracted from a news portal, ensuring a representation of literate language. Subsequently, we performed transliteration on these files, converting the Georgian characters into their Latin equivalents as per our transliteration table, and enclosed these Latin segments within given markers. As a result, for each file, we obtained a transliterated version with sizes approximately reduced to one third.

In the experiment we selected ASCII control characters as markers. Specifically, we tested the NULL character, the file separator (FS) and the record separator (RS). These characters were chosen for their minimal size, being only

1 byte each, and their invisibility, ensuring they do not affect the visual representation of the text. However, it's important to note that handling NULL characters may vary slightly across different platforms and programming languages. In our scripts, alternative markers whether custom control characters or special sequences, visible or invisible can also be used, depending on which is most appropriate for a given application.

To transliterate Georgian text to basic Latin characters on a character by character basis, we developed the `ka2lat.py` Python script, which uses the custom *ka2lat* translation table to map Georgian characters to Latin characters. For detransliteration, we created `lat2kat.py`, which employs the custom *lat2kat* translation table. Both scripts require two parameters: *File* to specify the file path containing text and and *Marker* to define symbols used for marking the corresponding Latin segments. All the scripts produced so far, including Python scripts are publicly and freely available online[4].

To transliterate and detransliterate Georgian text characters on a line by line basis, we used the `sed` command with the following syntax:

```
### Define variables
export Marker='\x0' # Marker=$'\036' # Marker=$'\034'
export Lat='abgdevzTiklmnopJrstufqRySCcZwWxjh'
export Ka='აბგდევზთიკლმნოპჟრსტუფქღყშჩცძწჭხჯჰ'

### Transliteration
sed -e "s/[a-zTJRSCZW]*[a-zTJRSCZW]/${Marker}&${Marker}/g"
    -e "y/$Ka/$Lat/" File > File_tr

### Detransliteration
sed "s/\(${Marker}[^${Marker}]*[^${Marker}]${Marker}\)\1*/\n&\n /g
    ;G;s/^/ /" File_tr | sed -e '/^ /y/'${Lat}/${Ka}'/;/./{H;d;}'
    -e'x;s/\n \{0,1\}//g' -e "s/${Marker}//g" >File_restored
```

In this process, the file `File` contains a mix of Georgian and non-Georgian characters. The transliterated version is saved in `File_tr` and the restored version is saved in `File_restored`.

For buffer-based transliteration using `vim`, we used the following syntax:

```
### Define variables
export Marker=$'\036' # Marker=$'\034'

### Transliteration
vim -c ":%s/[a-zTJRSCZW*[a-zTJRSCZW]/$Marker&$Marker/g"
    -c '%!sed "y/აბგდევზთიკლმნოპჟრსტუფქღყშჩცძწჭხჯჰ/abgdevzTiklm
    nopJrstufqRySCcZwWxjh/"' -c':wq' File_tr

### Detransliteration
```

[4] https://github.com/aelizbarashvili/Ka2Lat2Ka

```
vim -S <(echo "source tr.vim") -c "%s/$Marker\zs[^$Marker]*
    $Marker\|\zs[a-zTJRSCZW]/\=Translate(submatch(0))/g"
    -c "%s/$Marker//g" -c 'wq' File_restored

### The Translate function is defined in the tr.vim file:
" translate.vim
function! Translate(char)
    let charmap = {'a': 'ა', 'b': 'ბ', 'g': 'გ', 'd': 'დ', 'e': 'ე',
    'v': 'ვ', 'z': 'ზ', 'T': 'თ', 'i': 'ი', 'k': 'კ', 'l': 'ლ', 'm':
     'მ', 'n': 'ნ', 'o': 'ო', 'p': 'პ', 'J': 'ჟ', 'r': 'რ', 's': 'ს',
    't': 'ტ', 'u': 'უ', 'f': 'ფ', 'q': 'ქ', 'R': 'ღ', 'y': 'ყ', 'S':
     'შ', 'C': 'ჩ', 'c': 'ც', 'Z': 'ძ', 'w': 'წ', 'W': 'ჭ', 'x': 'ხ',
    'j': 'ჯ', 'h': 'ჰ'}
    return get(charmap, a:char, a:char)
endfunction
```

These scripts have been successfully tested for transliteration and detransliteration on our sample files of varying sizes: 10 MB, 100 MB, 1 GB, and 5 GB. In each case, the restored files were compared to the originals using the `cmp` command, and all were found to be 100% identical. This confirms the accuracy of the transliteration and restoration process, serving as the best quality evaluation of these transformations. However, as file sizes increased, processing time became a limiting factor.

The experiment considered the importance of disk buffering and cache in optimizing data reading and writing. Measurements were performed on cached data of varying file sizes.

Overall, the experiment aimed to compare different approaches to transliteration and detranslation, considering practicality and performance.

5 Performance Measurements and Results

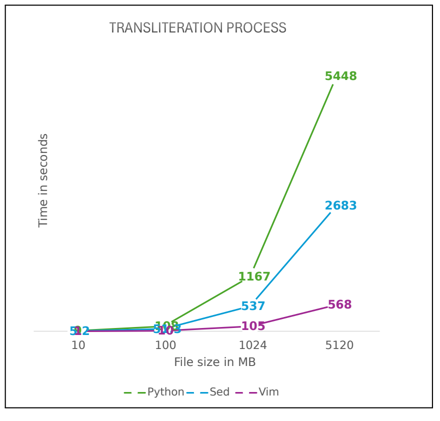
During the experiment we collected detailed resource usage information across files of various sizes – 10 MB, 100 MB, 1 GB, and 5 GB – using Python scripts, Sed, and Vim. We monitored these processes using the shell's `time` command with the `-v` option, which measures various performance metrics and provides a comprehensive report, including execution time, CPU utilization and memory usage.

Table 1 presents the processing times for transliteration and detransliteration across different file sizes using three different approaches: character by character processing with Python scripts (`ka2lat.py` and `lat2ka.py`), line by line processing with the `sed` command, and buffer-based processing with the `vim` command.

Charts in Figure 1 depict the data presented in the Table 1, unveiling that, in terms of processing time, the `vim` command achieves the highest performance in transliteration, while the `sed` command excels in detransliteration.

Table 1. Processing time for transliteration and detransliteration.

	Python				Sed				Vim			
File Size (MB)	10	100	1024	5120	10	100	1024	5120	10	100	1024	5120
Tr. time (sec)	9	108	1167	5448	5.2	51.3	537	2683	1	10	105	568
Detr. time (sec)	3.4	39.5	523	1954	0.41	4	39.5	200	43	571	1140	35845



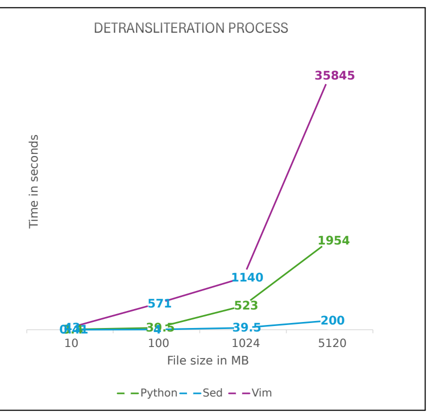


Fig. 1. The transliteration and detransliteration processing's details

Table 2 displays memory usage for transliteration and detransliteration across different file sizes using the mentioned three different approaches: character by character processing, line by line processing, and buffer-based processing. It shows that Python scripts and the `sed` commands maintain approximately constant memory usage across all file sizes for both transliteration and detransliteration. For Python scripts, several factors contribute to this consistency: Reading in Chunks: The script processes the file character by character rather than loading the entire file into memory. This approach reduces the memory footprint, as only a small portion of the file is held in memory at any given time; Internal Buffers: Python uses internal buffering mechanisms for file I/O operations. This can lead to consistent memory usage patterns, as the buffering strategy does not scale linearly with the size of the file; Efficient I/O: Reading one character at a time with `f.read(1)` ensures that only a small amount of data is held in memory. This efficient handling of file I/O operations contributes to the stable memory usage; Data Structure: The *ka2lat* and *lat2ka* dictionaries are relatively small data structures compared to the size of the input file. Their size does not significantly impact the overall memory usage. For `sed` the maximum RAM usage during execution did not exceed 3 MB, regardless of file size, due to: Streaming Data Processing: `sed` processes data in a streaming fashion. Instead of loading the entire file into memory, `sed` reads the input file line by line or in small chunks, applies the specified operations, and writes the result to the

output file. This means that `sed` only needs to keep a small portion of the file in memory at any given time, regardless of the total file size.

In contrast, `vim` shows a significant increase in memory usage as file sizes grow, with sharper rises for larger files. This behavior is due to: In-Memory Editing: `vim` loads the entire file into memory for editing. This behavior contrasts with `sed` and Python's streaming approach, where only portions of the file are processed at a time. For large files, this can lead to significant memory usage; Buffering and Undo History: `vim` maintains an undo history and buffer for the entire file, which adds to memory usage. This feature allows you to undo and redo changes, but it also increases memory consumption, especially for large files.

Table 2. Maximum memory usage for transliteration and detransliteration.

	Python				Sed				Vim			
File Size (MB)	10	100	1024	5120	10	100	1024	5120	10	100	1024	5120
Tr. RAM (MB)	10	10	11	11	3	3	3	3	44	352	3432	16507
Detr. RAM (MB)	10	11	11	11	3	3	3	3	34	250	2409	11460

Regarding CPU time, it's noteworthy that CPU usage was consistently high, nearly 99% on one core, across all file sizes in Python, `sed`, and `vim`, despite the presence of multiple cores. This is due to how these tools are designed to operate and utilize system resources. Both `sed` and `vim` operate in a single-threaded mode for their core processing tasks. This means that, even if the server has multiple cores, these tools do not take advantage of parallel processing for the main operations. They perform their tasks on one core and do not distribute the workload across multiple cores. `sed` and the `vim` do not have built-in parallelism for processing large files. They handle text operations sequentially on one core. The Python script also lacks explicit parallelization, so it runs on a single core by default. The operations being performed (substitution, searching, transliteration) are CPU-bound tasks. They involve intensive computation that is executed on a single thread.

Main key findings are the following:

1. Python scripts:
   - Performance and Scalability: The Python scripts (`ka2lat.py` for transliteration and `lat2ka.py` for detransliteration) showed consistent execution times across various file sizes. The processing time increased linearly with file size (10 MB, 100 MB, 1 GB, 5 GB), indicating potential efficiency challenges when handling larger datasets.
   - CPU Utilization: Across all file sizes, the Python scripts utilized nearly 99% of one CPU core. This high CPU usage is typical of the Python's single-threaded execution, which does not leverage multi-core processing.

   - Memory Usage: The maximum memory usage remained constant at around 11 MB, irrespective of the file size. This efficient memory management can be attributed to Python's internal buffering and chunked file processing, which minimizes the memory footprint by reading and processing the file character by character rather than loading the entire file into memory.
2. Sed command:
   - Performance Efficiency: The `sed` command, known for its efficiency in stream editing, exhibited competitive performance, especially in detransliteration tasks. It was significantly faster than Python for both transliteration and detransliteration.
   - CPU Utilization: Similar to Python, `sed` also utilized nearly all available CPU time on a single core. However, its stream processing model allowed it to handle large files efficiently.
   - Memory Usage: The memory usage for `sed` remained impressively low, not exceeding 3 MB, regardless of file size. This is due to `sed`'s ability to process data in a streaming fashion, handling only small portions of the file in memory at any time. This allows it to handle large files efficiently without requiring a proportional amount of RAM.
3. Vim text editor:
   - Performance: Vim's buffer-based processing showed the fastest processing times for transliteration, outperforming both Python and `sed`. However, it exhibited the slowest performance in detransliteration, especially for large files. This discrepancy is due to the computational expense of handling many substitutions, conditional checks, and context switches during detransliteration.
   - CPU Utilization: Like the other tools, Vim used nearly 99% of one CPU core across all file sizes.
   - Memory Usage: Unlike Python and `sed`, Vim's memory usage increased significantly with file size. For transliteration, memory consumption rose from 44 MB for 10 MB file to 16.5 GB for 5 GB file. For detransliteration, memory usage ranged from 34 MB for 10 MB file to 11.5 GB for 5 GB file. This high memory consumption is due to Vim's in-memory editing model. As an interactive text editor it loads the entire file into memory for editing, and maintains a buffer and undo history.

It is important to note that our data was cached during the experiments. Caching mechanisms play a crucial role in optimizing performance, resource utilization, and scalability in data processing experiments. Their significance becomes more pronounced as the size of the data and the complexity of the computations increase.

Comparison and Implications:

- The performance measurements underscored the significance of choosing an appropriate method based on the task's nature and data volume.

- Vim's outstanding performance, attributed to its buffering mechanism, emphasized its suitability for handling substantial transliteration workloads, particularly when memory usage is not a constraint.
- The experiment highlighted the practical advantages of shell commands, such as `sed`, in terms of processing speed, especially in scenarios involving extensive data processing when detransliterating. These commands offer a lightweight and efficient solution for tasks that do not require the overhead of a full editor or more complex scripting languages.
- Despite Python's ease of use and flexibility, its longer execution time when processing text character by character compared to `sed` suggests that it may not be the best choice for extremely large files. However, Python's capabilities for more complex operations make it a strong candidate for tasks that go beyond simple text processing.
- The results underscore the importance of understanding the trade-offs between memory usage, processing speed, and the complexity of tasks when selecting tools for text processing and transliteration.

Table 3 displays the time required, in seconds, to execute various standard shell commands commonly employed in text processing. It includes scenarios involving chained commands with pipes, as text processing in the shell typically involves a series of commands.

The presentation showcases the performance for two distinct files: one containing Georgian texts with occasional Latin character fragments and the other with its transliterated counterpart, identified as '`f`' and '`f_tr`' in the table 3. The command column specifies '`f[_tr]`,' with brackets denoting optional parts of the command line semantics. This indicates that the command is initially executed for the file '`f`' and subsequently for the file '`f_tr`'.

Additionally, it's important to note that '`>/dev/null`' is appended for all commands. This redirection to `/dev/null` ensures that the command's output is not displayed on the terminal or saved to a file. This practice proves beneficial when the output itself may consume significant time or resources to process, allowing us to focus exclusively on measuring the command's execution time without interference from output handling.

Table 3. The experiment results

File size (MB)	10		100		1024		5120	
Command (`>/dev/null`)	Time (seconds)					factor		
`cat f[_tr]`	0.007 0.005	1.40	0.029 0.014	2.07	0.176 0.071	2.47	0.940 0.288	3.26
`sort f[_tr]`	0.199 0.019	10.47	3.534 0.181	19.5	18.113 1.246	14.53	69.555 5.492	12.66
`wc -l f[_tr]`	0.009 0.005	1.8	0.037 0.018	2.1	0.246 0.100	2.5	1.079 0.418	2.6
`uniq f[_tr]`	0.039 0.019	2.1	0.255 0.108	4	2.434 0.964	2.5	12.344 4.713	2.6
`grep -v ‘[ა-ჰ]’ f` `grep -v ‘[a-z]’ f_tr`	0.015 0.004	3.8	0.054 0.005	10.8	0.449 0.005	90	2.165 0.005	433
`sed ‘/[ა-ჰ]/d’ f` `sed ‘/[a-z]/d’ f_tr`	0.018 0.012	1.5	0.084 0.052	1.6	0.761 0.321	2.4	3.831 1.497	2.6
`awk -F" " '{print $NF}'` `f[_tr]`	0.038 0.029	1.3	0.264 0.175	1.5	2.422 1.682	1.4	12.163 8.378	1.5
`tr -d [0-9] < f[_tr]`	0.029 0.016	1.8	0.153 0.067	2.3	1.411 0.564	2.5	7.115 2.723	2.6
`paste f[_tr] f[_tr]`	0.075 0.032	2.3	0.639 0.246	2.5	6.241 2.356	2.6	31.007 11.946	2.6
`cat f[_tr] | sort | uniq`	0.216 0.038	5.7	3.887 0.336	11.6	53.079 3.907	15.6	289.736 21.963	13.2

`cat f[_tr] | sort | uniq |`
`grep -v '[ა-ლ]' | sed -n`
`'/^[მ-პ]/p' | tr -s " "`
	0.231
	0.050
		3.8	3.947
			0.372
				10.1	55.416
					3.991
						13.9	290.439
							21.920
								18.7

The demonstration is conducted across four different file sizes: 10 MB, 100 MB, 1 GB, and 5 GB. Furthermore, the green cell in the table displays a factor indicating the relative speed improvement when executing commands using the transliterated version of the file compared to the original content containing Georgian texts.

Considering the frequent execution of compiled commands in text processing workflows, achieving results approximately 18 times faster on transliterated files significantly expedites the overall process.

Another compelling indication of the efficacy of working with translated versions of large Georgian texts, as opposed to the texts containing Georgian letters and symbols, is evident in the comparison of translation/detransliteration times. Transliteration (working with Georgian text) using the `sed` command takes approximately 45 minutes, whereas detransliteration (working with Latin text) is completed in approximately 3.5 minutes, marking a notable 13-fold increase in speed. It's important to note that this process was accomplished using a single command, `sed`. When introducing additional commands and complexity, this enhanced performance is expected to scale even further.

The measurements were conducted on a pristine installation of Ubuntu 22.04 server minimal, devoid of any additional local or network services. No extraneous processes were running that could influence disk I/O. The presented table 3 reflects averaged data, ensuring that measurement times are reasonably accurate.

The technical specifications of the server are detailed in the Table 4:

Table 4. The technical specifications of the server

Processor	Intel(R) Xeon(R) Gold 5118 CPU @ 2.30GHz, 32 cores, model: 85
RAM	48 GB
Disk	100 GiB, with 76% free space

6 Conclusion

This study extensively explores character encoding intricacies, emphasizing its crucial role in natural language processing (NLP) tasks and addressing challenges specific to the Georgian language. The effectiveness of UTF-8 encoding in representing and storing Georgian characters is unveiled, exemplified by the Kartu-Verbs project's real-world application in constructing a comprehensive Georgian verb database. This underscores the necessity for efficient handling of extensive text files in NLP applications.

Performance measurements for standard shell commands emphasize efficiency gains when working with translated versions of large Georgian texts. The substantial speed increase, especially in scenarios involving extensive data processing, underscores the practical advantages of shell commands in text processing workflows.

Our method to transliterate Georgian texts proves efficient because it only processes the Georgian portions of the text, not other segments, reducing computational overhead. The use of a bijective mapping ensures that transliteration and detransliteration are both accurate and reversible. The flexible marker system provides a robust approach for handling Georgian text. Overall, Our approach provides a solid foundation for effectively transliterating and detransliterating for Georgian texts that can be adapted to other languages also that uses multi-bytes characters.

Various tools were compared, including a Python script, the `sed` command, and the `vim` text editor. This assessment involved examining the execution times across files of differing sizes, ranging from 10 MB to 5 GB (10 MB, 100 MB, 1 GB, 5 GB). The notable finding was the consistent superiority of Vim for transliteration, particularly evident as file sizes increased, though it also exhibits significant memory usage growth as file size increases. Processing the data in buffered mode with `vim` proved to be approximately 5 times faster compared to using only the `sed` command, which represents the second result. For detransliteration, `sed`

demonstrated superior performance in terms of execution time and memory usage compared to both the Python script and `vim`, processing conversions from Latin to Georgian about 10 times faster than the Python script, which represents the second result.

The experiment yielded valuable insights into the efficiency of handling single-byte texts compared to multi-byte texts. The superior performance advantage of using various shell commands for processing transliterated Georgian texts, particularly evident in the processing of larger files, becomes apparent – it processes transliterated texts approximately 20 times faster than texts with Georgian characters. The discussion on disk buffering and cache underscored their pivotal role in optimizing data processing, with potential implications for real-world applications. Overall, the experiment aimed to provide a comprehensive comparison of transliteration methods, considering both practicality and performance, thereby contributing valuable data for further exploration and optimization.

In the evolving digital landscape, where global communication diversifies, character encoding role, particularly in languages like Georgian, becomes increasingly crucial. The insights from this study contribute to optimizing data processing in NLP tasks, providing valuable considerations for researchers and practitioners. The study's focus on character encoding complexities and practical applications of transliteration methods deepens our understanding of NLP efficiency, emphasizing the importance of thoughtful method selection to enhance overall effectiveness, especially in languages with distinctive encoding requirements.

7 Acknowledgements

This research PHDF-22-1840 is supported by Shota Rustaveli National Science Foundation of Georgia (SRNSFG).

References

Jennings, T. (2010). *An annotated history of some character codes*, `https://www.sensitiveresearch.com/Archive/CharCodeHist/index.html`.

Strizver, I. (2011). *Confusing (and Frequently Misused) Type Terminology, Part 1*, `https://web.archive.org/web/20111225024213/https://www.fonts.com/aboutfonts/articles/situationaltypography/confusingtypeterms.htm`.

IBM (1954). *IBM Electronic Data-Processing Machines Type 702, Preliminary Manual of Information.*

Mackenzie C. E. (1980). *Coded Character Sets, History and Development*, IBM Corporation.

ISO (2020). *ISO/IEC 10646:2020, Information technology, Universal coded character set (UCS)*, `https://www.iso.org/standard/76835.html`.

Kozierok, C. M. (2017). *The TCP/IP Guide: A Comprehensive, Illustrated Internet Protocols Reference*, 1st Edition.

Unicode Terms (2024). *Glossary of Unicode Terms*, `https://www.unicode.org/glossary/`.

Unicode Standard (2024). *Components of The Unicode® Standard*, Version 15.1.0, https://www.unicode.org/versions/components-15.1.0.html.

Mackenzie, C. E. (1980). *Coded Character Sets*, History and Development (The Systems Programming Series), Addison-Wesley Publishing Company, Philippines.

Unicode Technical Reports (2023). *Unicode character database, Unicode® Standard Annex #44*, https://www.unicode.org/reports/tr44/tr44-32.html.

Mammadzade, S. (2018). *Transliteration and Transcription in Data Processing*, Problems of information society. №1, DOI: 10.25045/jpis.v09.i1.11., pp. 100–106.

Chinnakotla M. K., Damani Om. P., Satoskar Avijit. (2010). *Transliteration for Resource-Scarce Languages*, ACM Transactions on Asian Language Information Processing, Article 14, **vol. 9**, no. 4, p. 30, 2010.

Zhao, W. (2021). *Chinese-English Subtitle Translation Strategies from the Perspective of Communicative Translation Theory: A Case Study of China from Above by Knny*, The 8th International Conference on Education, Language, Art and Inter-cultural Communication.

Elizbarashvili, A., Ducassé, M.,Khachidze, M., and Tsintsadze, M. (2023). *From structured textual data to semantic linked-data for Georgian verbal knowledge*, Proceedings of the eLex2023 conference, Lexical Computing CZ, Brno, Czech Republic.

Spencer, G. A. (2001). *Digitization, Coded Character Sets, and Optical Character Recognition for Multi-script Information Resources: The Case of the Letopis' Zhurnal'nykh Statei*, International Conference on Theory and Practice of Digital Libraries.

Tsintsadze, M., Khachidze, M., Archuadze, M. (2022). *On Georgian Text Processing Toolkit Development*, Analysis of Images, Social Networks and Texts.

Soselia, D., Tsintsadze, M., Shugliashvili, L., Koberidze. I., Amashukeli, S., Jijavadze, S. (2018). *On Georgian Text Processing Toolkit Development*, 18th IFAC Conference on Technology, Culture and International Stability TECIS, *https://doi.org/10.1016/j.ifacol.2018.11.279*.

Received January 19, 2024 , revised September 9, 2024, accepted November 6, 2024