INSTITUTO TECNOLÓGICO Y DE ESTUDIOS SUPERIORES DE MONTERREY

CAMPUS MONTERREY

Materia:

TC2037 - Implementation of Computational Methods (Gpo 605)

E2 Applied parallel programming

Fernando Morán Fougerat	A01284623
Mónica Soberón Zubía	A01254488
Bernardo José Willis Lozano	A00837831

Profesor:

Edgar González Fernández

Monterrey, Nuevo León, México **Viernes 7 de junio del 2024**

Table of Contents

Team Members and Roles

Team Member	Role
Bernardo José Willis Lozano	Documentation Specialist
Fernando Morán Fougerat	Implementation Specialist
Mónica Soberón Zubía	Testing and Performance Analyst

Introduction

This documentation offers an overview and in-depth explanations of the functions and procedures implemented in the code for letter frequency analysis and substitution cipher operations. The code reads an encrypted text from a file, analyzes the letter frequency, and attempts to decrypt the text using letter frequency analysis. The decryption process can be executed in parallel using multiple CPU cores to enhance performance.

Libraries

We used the libraries string , collections, multiprocessing and time in order to correctly evaluate our texts, and get our expected results.
‘multiprocessing’: Enables parallel processing using multiple CPU cores.
‘time’: Used for measuring the execution time of the decryption process and parallel process.

Substitution Cipher Implementation

Here’s an explanation of the algorithm implemented in our file analisis.py:

Code Explanation:

‘analisis.py’:
This script contains the functions needed to perform letter frequency analysis, swap the positions of the letters using a key with a function called ‘swap_letters’, and decrypts the text using that key. And finally the script has the main function ‘substitution_analysis’ that takes a cipher and performs the decryption to an output file. Uses multiprocessing when reading chunks and counting letters of .txt file.

‘main.py’:
This script manages the execution of the decryption process, comparing parallel and sequential processing times.

Functions Used:

‘analisis.py’:

‘letter_frequency_table”

Contains the typical frequency of each letter in the English language, used to guide the decryption process.

‘encrypt_char(char, key)’

Encrypts a single character using the provided key.

‘swap_letters(key, letter1, letter2)’

Swaps two letters in the key to aid manual decryption.

‘substitution_cipher(text, key, ncores)’

Encrypts or decrypts text using a substitution cipher. Utilizes multiprocessing for parallel processing.

‘substitution_analysis(input_path, output_path, ncores)’

Performs the analysis and decryption on the text read from ‘input_path’ and writes the decrypted text to ‘output_path’.

‘main.py’:

‘process_file(args)’

Processes a single file by calling the ‘substitution_analysis’ function.

‘Parallel and Sequential Execution’

Part of the code that measures and compares the time taken to decrypt text files using both parallel and sequential processing methods.

How does the decryption process work?

Steps for the decryption process

1. The decryption process starts by reading the encrypted text from the specified input file.
2. The frequency of each letter in the encrypted text is counted.
3. The letters in the encrypted text are sorted.
4. A decryption key is generated by mapping the most frequent letters in the encrypted text to the most frequent letters in the English language.

5. The text is then decrypted using the key, and this step is parallelized to speed up the process.
6. Finally the decrypted text is written to the output file.

How does the parallel implementation work?

Multiprocessing:

The decryption process uses ‘multiprocessing’ library to parallelize the substitution cipher function. This works by splitting the text across multiple CPU cores, so it achieves faster processing times.

Pool:

This process is created to handle multiple parts of the text simultaneously across the available cores.

Comparative Analysis:

The script compares the performance of the parallel implementation with a sequential approach to demonstrate the efficiency gains achieved through the parallel process. The starting and end times are printed and so is the total execution time for comparison.

Test Cases

We made different types of tests, first we tested our function with a sequential approach against a parallel implementation, with the same 7 texts being tested. After that, we decided to check in our biggest text file how reducing their number of cores that allow our multi-process to occur affects our time. Finally, we made a test of all 6 initial txt files being run together by either method, to compare time.

The function was tested with a sequential approach and a parallel implementation on the same seven text files. The results are as follows:

Text	Parallel time	Sequential time
1	0.12415266036987305	0.15378165245056152
2	0.21088147163391113	0.12247467041015625
3	0.16807770729064941	0.9383270740509033
4	0.17751479148864746	0.10544276237487793
5	0.1313941478729248	0.07561016082763672
6	2.6711056232452393	2.7988228797912598

7	5.417501211166382	6.825813293457031
All texts	2.060619354248047	10.457871675491333

*Table 1. Parallel versus Sequential time comparison for each text and all texts**

* For this case, we couldn't use our original function, so we decided to turn it into a sequential approach (removing all multiprocessing uses as we couldn't run them at the same time if they had them, even if we tried and used an nCore of 1). Hence the creation of our other file, analisis_s.py.

The largest text file was tested using different numbers of cores to see how reducing the number of cores affects the processing time:

Text	Parallel time (all cores)	Parallel time (½ cores)	Parallel time (¼ cores)
7	5.417501211166382	5.828564882278442	6.080790042877197

Table 2. Different cores usage and parallel time comparison

Conclusions

- The parallel implementation significantly reduces the processing time for larger texts compared to the sequential approach. As you can see, in text 7, it took 5.418 seconds in parallel processing versus 6.826 sequentially.
- The parallel approach scales well with larger texts, achieving substantial reduction in time when it processes all texts together, as you can see in *Table 1*.
- Going to *Table 2*, reducing the number of cores available for parallel processing makes the time increase its value.
- It is important to take into consideration that the parallel time, even with fewer cores, still outperforms the sequential approach.
- The results suggest that using all available cores yields the best performance. If you see at *Table 2*, as the number of cores decreases, the time increases.
- As we ran our code from a virtual machine, the outputs could be slower from the ones we could get if we ran this locally, but still the results are strong enough to successfully see a difference in our results. (For example, in replit, the first run in a new environment was always the slowest, and after that we had faster results in both sequential and parallel approaches in following tests).
- Results in our deciphered .txt files were not 100% perfect, however that is due to statistics of letter repetition not being 100% followed in different texts, however our results are close enough that one person can identify by eye the few letters to change to correctly solve the cipher. We initially had added a loop where the user was shown the first 15 lines of the deciphered txt file, and was given the option to swap 2 letters. We believe that our code was very precise, and with few uses of this loop, one user could easily decipher any english .txt that has been decrypted using a basic monoalphabetic substitution cipher.

References

Pico Cetef. (2013, 26 abril). *Cryptography 101 - Substitution Ciphers* [Vídeo]. YouTube. https://www.youtube.com/watch?v=1P8Xpxm76e8

English letter frequency table. (s. f.). Gist. https://gist.github.com/devxleo/9c4f7e917b0f68d7eba3

multiprocessing — Paralelismo basado en procesos — documentación de Python - 3.9.19. (s. f.). https://docs.python.org/es/3.9/library/multiprocessing.html

time — Acceso a tiempo y conversiones — documentación de Python - 3.10.13. (s. f.). https://docs.python.org/es/3.10/library/time.html