Big Data Analytics 2024 Project 1

Oboni Anower

13923163

June 2024

Introduction

This Big Data Analytics project focuses on developing efficient programming strategies for downloading videos from YouTube to use them for further analysis. The goal is to use parallel processing techniques such as Threading, Multiprocessing, and Asynchronous programming to perform simultaneous tasks by taking advantage of modern computational resources. The programming is done in Python, as it offers an extensive libraries and frameworks for parallel processing and multimedia analysis.

This report will discuss about the parallel programming using Threads to download videos, compare it's performance with a serial execution, and discuss space-time complexities for both. It will also discuss about parallel programming libraries used to complete five video analysis tasks. One of such tasks are Audio extraction from the videos, which will be used to compare the execution time taken for different processing methods. Once "*main.py*" is run, all the functionalities will be executed one after another, in order of their call (see Appendix 1).

Folder Structure

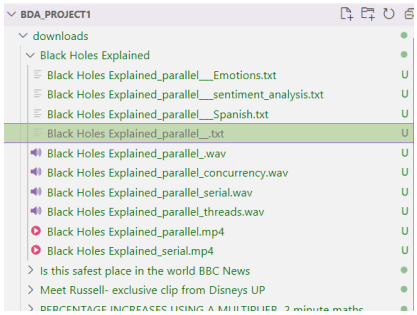

Image 1: dependant files in folder

For every YouTube Video, a folder is generated and named after the video title. The downloaded videos are saved in their respective folders. Once the `main()` function is run successfully, each folder will contain 11 files. Every task, starting from serial test download to subtasks will be downloaded to these folders.

GitHub Repo: https://github.com/0anower/BDA_Project1.git

Libraries

```
pip install pytube moviepy speechrecognition spacy nltk NRCLex Path
TextBlob googletrans==4.0.0-rc1
```

YouTube Video Downloads

Video URLs

"*video_urls.txt*" text file generated from a list containing 12 URLs.

Serial and Parallel Downloads

While both serial and parallel downloads are run using the same download function `video_download()`, each video in serial programming is downloaded sequentially and only one video is downloaded at a time. Although the program is much simpler, the process tends to be slower, especially if dealing with large volume of data.

Since downloading videos are I/O bound task from a shared resource (list of URLs in same text file), the download speed may often slow down due to the network's latency or low disc-space. However, the process can be sped up with the help of parallel programming like threads. A thread used over multiprocessing because instead of utilising multiple processors, threads share their memory address for a single process, significantly reducing the overheads.

```
THE NUN II | OFFICIAL TRAILER Download completed
Download started at: 02:07:25 .........Phil Lempert's 2 minute Speech Demo
Phil Lempert's 2 minute Speech Demo Download completed
Download started at: 02:07:28 .........Self Motivation | Brendan Clark | TEDxYouth@BarnstableHS
Self Motivation | Brendan Clark | TEDxYouth@BarnstableHS Download completed
Download started at: 02:07:31 .........Is this safest place in the world? BBC News
Is this safest place in the world? BBC News Download completed
Serial download finished in 34.80964469999992 seconds
```

Image 2: Serial video download

```
Download started at: 02:07:47 .........PERCENTAGE INCREASES USING A MULTIPLIER | *2 minute maths*
Download started at: 02:07:49 .........THE NUN II | OFFICIAL TRAILER
Download started at: 02:07:51 .........Phil Lempert's 2 minute Speech Demo
Black Holes Explained Download completed
THE NUN II | OFFICIAL TRAILER Download completed
PERCENTAGE INCREASES USING A MULTIPLIER | *2 minute maths* Download completed
Download started at: 02:07:53 .........Is this safest place in the world? BBC News
Phil Lempert's 2 minute Speech Demo Download completed
Download started at: 02:07:55 .........Self Motivation | Brendan Clark | TEDxYouth@BarnstableHS
The 'Busy' Life of the Sloth | BBC Earth Download completed
Self Motivation | Brendan Clark | TEDxYouth@BarnstableHS Download completed
Is this safest place in the world? BBC News Download completed
Parallel download finished in 24.178622099999984 seconds
```

Image 3: Parallel Video Download

Image 1 shows the time taken to execute the same `video_download()` to download each video serially, and concurrently using a thread pool. A thread pool is collection of pre-installed threads that are inactive but on standby. Thus, instead of creating a new thread every time for every download, the thread pool can be used to execute of large number of threads concurrently. It is also faster compared to serial execution. A ***semaphore***=*5* is used for parallel download, to ensure that no more than 5 threads can access the downloading function at the same time.

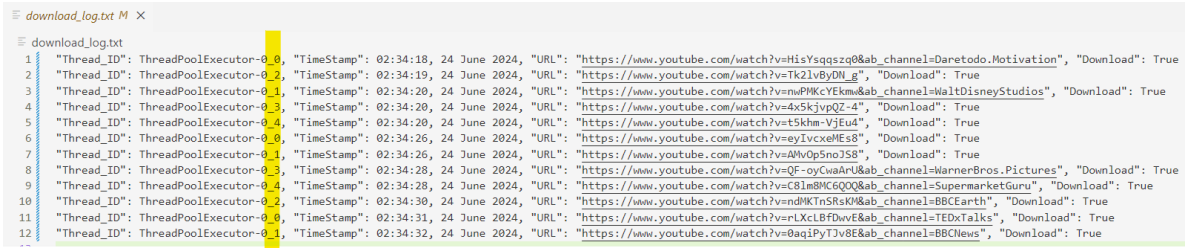

Image 4: download_log.txt

Although this project requires only 10-15 downloads, a concurrent thread pool is chosen because it reuses a fixed number of threads from the pool, in this case at most 5, which reduces overhead. This is because when a working thread in the pool completes its execution, that same thread can do another available task in the same process. Additionally, when one thread is terminated from the pool, a replacement will complete the following task. Image 3 is "*download_log.txt*", generated after the download completion, which shows that five threads work together concurrently. A ***mutex*** is used in to generate this log, to prevent multiple threads trying to log at the same time. Instead, it allows only a single thread to write at a time to maintain consistency.

Time and Space Complexities

Serial Execution

Image 5: Serial Programming

Time Complexity: YouTube videos are downloaded using `video_download()` which takes constant time O(1) for each video. The Since there are 12 videos in the List, the time complexity to loop through each video to download all 12 will be O(n).

Space Complexity: To download each of the 12 videos in the list, so the list will take space O(n). Although 12 videos will be downloaded, only 1 video is being downloaded at a time using single thread in a single process. Thus, the space complexity of download remains constant O(1). Thus, the total space complexity can be O(n) + O(1) = O(n)

Parallel Execution

Image 6: Parallel Programming using ThreadPoolExecutor

Time Complexity: Thread Pool Executor in the script can handle up to 5 downloads at time. Assuming constant time for each download, n=12 tasks are distributed into groups of 5. Since the download is concurrent, in each group it will take maximum *t time* to download 5 videos in parallel, considering the time to be linear. So, the time complexity will be O(n/5) = O(n)

Space Complexity: `parallel_downloads` is a list that keeps a reference to the submitted Future objects. Up to 12 Future objects will be created to store in the list because there are 12 videos URLs need to be processed. This makes a space complexity linear, of O(12) = O(n).

Analysis Subtasks

Among the recommended libraries, `multiprocessing` library was implemented as the choice of audio extraction and transcription because these involve decoding videos, extracting audio streams, and processing them, which makes them computationally intensive CPU bound tasks. This is because they distribute the workload across multiple CPU cores.

However, for sentiment analysis, translation, and emotion extraction, the decision was to use `threads`, as they perform fast and use less computational resources compared to multiprocessing. Creating and managing threads also cause less overhead than processes. Additionally, the communication between threads is faster compared to the communication between processes. When executions using Threads and Multiprocessing are compared, it is seen that threads usually tend to perform faster than processes.

For text translation, Google's translation API interface from `googletrans` library is used which can autodetect languages and translate as desired.

Finally, the comparison between different processes such as serial, multiprocessing, threading, and concurrency are experimented for the audio extraction functionality.

Comparison

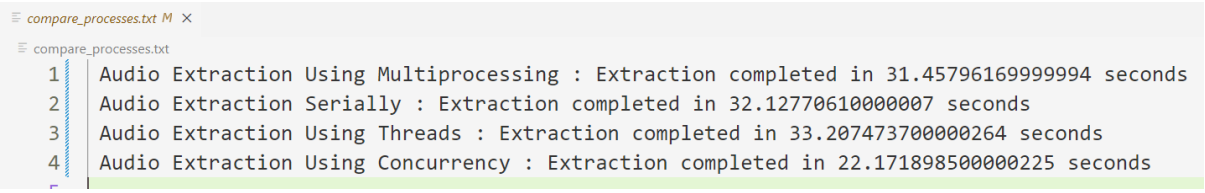

Image 7: Time duration for each process of Audio Extraction

Image 5 shows a text file "*compare_processes.txt"* which is generated from **task 5** at the end of each audio extraction process, showing the time taken for each of the execution. It shows that concurrency using process pool is the fastest, followed by multiprocessor execution. In this case, threads took the longest because they tend to slow down CPU-bound programs. Concurrent ProcessPoolExecutor takes the shortest time because a pool of processes asynchronously executes functions and call them independently using multiple cores.

Appendix

Appendix 1: Folder Structure once the main() function is run successfully.