YouTube Data Collection Using Parallel Processing

Joseph Kready, Shishila Awung Shimray, Muhammad Nihal Hussain, Nitin Agarwal
Department of Information Science
University of Arkansas at Little Rock (UALR)
Little Rock, USA
{jkready, sxawungshim, mnhussain, nxagarwal}@ualr.edu

***Abstract*—Several studies have identified social media platforms as significant data sources to study human behaviors and gain situational awareness about various events or crises. YouTube, being one of the largest social media platforms, provides a Data API that enables data collection on YouTube channels and videos which can be used in these studies. Current sequential methods for processing YouTube Data API requests are time consuming. In this paper we developed an implementation that utilizes Python's multiprocessing to process YouTube Data API request in parallel. Our tests indicate multiprocessing improves the performance by 400%. These improvements reduce computation time through utilization of multi-threaded CPU architecture.**

Keywords-YouTube; Parallel Processing; Social media data collection; Data Crawling

I. INTRODUCTION

There are over two billion people using social media [1]. Their activities and contributions on social media platforms generate data that provides insights into social behaviors. These datasets are at the heart of most social science research. Improving the data collection processes for the top social media platforms lowers the barrier of entry for future research. YouTube is the second largest social media platform, where users spend on average 40 minutes a day watching and commenting on videos. All this activity generates an estimated total of 10 Exabytes of data [2] making YouTube a treasure trove for researchers to use for novel social behavioral studies. The information collected can help researchers study information diffusion, identify digital communities [3] and study topic/interest trends, or identify deviant users and their inorganic behaviors [4].

However, collecting data with the current implementations of sequential YouTube API processing, is a time-consuming process. Moreover, the long processing time of standard sequential processing methods can result in limitations to the amount of data collectable.

Accessing data from any social media platform using their respective API involves several steps. For example, in YouTube Data API there is json conversions, feature engineering, data storage, etc. Many of these steps wait for I/O operations which bottlenecks the data collection performance. In order to maximize YouTube data collection efforts and overcome I/O bottlenecks, we propose parallelized YouTube data collection.

Parallelization is a method of decreasing the time to complete a task by dividing the project into subparts and running them in parallel. With parallelization, multiple API data requests can be submitted and processed at the same time. This allows larger sets of social media data to be processed in the same compute time.

The rest of the paper is organized as follows: Section 2 reviews the literature summarizing the methods for parallel processing. In section 3, we explain our implementation of multiprocessing to collect data from YouTube. Section 4 discusses the performance of our parallel data collection implementation against a baseline. We conclude with intended future work in section 5.

II. RELATED WORK

Social media platforms such as Twitter and YouTube provide a myriad of datasets that can be accessed through APIs for research purposes. Study on the social network based on the vaccination policies in Italy was conducted by accessing YouTube data through Netwizz, a tool for extracting data from the YouTube platform via the YouTube API v3 [5]

For data-intensive computing applications, today's hardware allows execution of parallel programs to increase performance. Python is widely used to develop such scalable applications that can implement parallelization because of its readability and availability of the various scientific libraries. Python distribution comes with a multithreading model that allows simultaneous execution of threads with shared memory. In the presence of only one core (or process) the OS creates the illusion of running multiple threads in parallel when in fact, it switches between the threads quickly, which is referred to as time division multiplexing [6]. Another optimization option in Python is the native multiprocessing module which supports spawning of processes in symmetric multiprocessing machines using an API similar to the threading module, with explicit calls to create processes, argument passing, execution, synchronization, result collection, etc. Another method [7] is to use the module to accelerate execution time of the concurrent inserts from asynchronous data streams. The authors also compare the performance of single thread synchronous inserts, single thread concurrent inserts and multi process concurrent inserts, concluding that the module can be used to efficiently for application that focuses on real-time text analyses and storage.

978-1-7281-7445-7/20/$31.00 ©2020 IEEE
DOI 10.1109/IPDPSW50202.2020.00185

III. METHODOLOGY

To begin data collection from YouTube, you must create a YouTube Data API key through the Google Developers Console [8]. New Data API keys come with a quota limit of 10,000 requests per day which allows you to access public YouTube data. Each request you send to the YouTube API will have a cost against your daily quota [9].

YouTube Data API is accessible through various programming languages. For this research, we focus on Python data collection using the Requests library for making API requests. For more information on building your own YouTube API Client [10].

To submit a request to the YouTube Data API you must create a URL containing the API URL, API key, Content ID, and data part. Submitting a proper request to the Data API will return a json response, which must be parsed for the relevant data and then saved as JSON, CSV, or SQL for further studies. One thing to note here is that all the data collected using YouTube's Data API is to be stored securely and in accordance with their terms of service [11].

The request process involves many I/O operations: sending and awaiting API response, re-sending requests for multiple pages, and saving data to file. Running a typical YouTube Data API request sequentially across millions of content IDs will result in slow performance as the singular Python processes will sit inactive awaiting I/O operations. The solution to reduce processing times is to utilize Multiprocessing in Python, which can create separate Python processes and make simultaneous YouTube Data API requests. Each individual process will still be limited by I/O operations, but the new pool of processes can work on the processing steps simultaneously.

When beginning to implement parallel processing in Python, you must first overcome the Global Interpreter Lock (GIL). The GIL is a part of CPython and forces all Python processes to go through a lock, allowing only 1 tread to execute at a time. The GIL was implemented to improve performance in single-threaded workloads while still allowing for parallel processing in C libraries like NumPy [12]. However, if you wish to parallelize your Pythonic code you must use the multiprocessing library, or a version of Python that does not have a GIL (Jython, Ironpython, PyPy-STM) [13].

Another issue to overcome when developing parallel processing in Python is the TCP Time wait delay. Making an API request or SQL server connection uses an available port on your system. On newer windows operating systems there are about 16,000 open ports [14]. Once a successful request is made, these ports go into a TIME_WAIT state, which can last 2 minutes on windows systems [15]. It's common for parallelized scripts that make HTTP connects to exhaust all available ports. A solution to this problem is to reduce the amount of time spent in TIME_WAIT state for the web sockets on your specific operating system. This allows the web sockets to become usable shortly after a connection is complete, keeping your system from exhausting the available ports.

As you begin to build parallel Python functions, you must consider how information will be shared between each process. Pickling is a method used by Python's Multiprocessing Library to convert your Python objects into byte streams [16]. This allows for processes to receive and send Python objects. Standard pickling has its limitations, such as a single function argument and limited pickable objects. A solution to Python pickling is Dill, an extension of pickling that has better support for converting and sending arbitrary classes and functions as byte streams [17]. The developers of Dill have also forked Python's multiprocessing library to support Dills directly. This can be found in the pathos library [18] and offers more functionality than Python's Multiprocessing library.

When building scripts for processing YouTube data, it is important to take into account that the API accepts 2 types of content IDs, channels and videos, and will provide a different response based on the content ID. The API's response for channels can include the id of the channel, its title, description, join date. Response for videos can include the video id, title, category, description, and date the video was uploaded to YouTube. At the heart of your data collection scripts, you must have a function that can take a content ID, send a request for data, and process/store the data appropriately.

When designing functions to support parallel processing, it's important to focus on For loops [Figure 1]. For a typical sequential process function, you would iterate through each content ID and process them 1-by-1.

```
## - Single Process - ##
def single_process_video(video_ids):
    for video_id in video_ids:
        process_video(video_id)
```

Figure 1. Looping in sequential data processing method.

To parallelize this code, you need to create a pool. A processing pool, as defined by the multiprocessing docs [19], is an object which offers a convenient means of parallelizing the execution of a function across multiple input values, distributing the input data across processes (data parallelism). The pools will stay active and open for jobs until pool.close() is called. For YouTube data processing, we use a map function which iterates through the list of video_ids and parallelizes it across the multiple processes. In the Figure 2, we use the iterative unordered map [20], which processes each item in the list in an unordered fashion while returning an iterator. In this study, we chose the uimap because unordered processing reduces wait times at the end of a batch, and the iterator allows for progress bars using libraries such as tqdm [21].

```
## - Parallel Process - ##
from pathos.multiprocessing import ProcessPool as Pool

def parallel_process_video(video_ids):
    #Creating a processing pool of 5 processes
    process_pool = Pool(nodes=5)

    #Mapping each video_id onto the process_video function
    process_pool.uimap(process_video, video_ids)
```

Figure 2. Looping in parallel processing method.

IV. PERFORMANCE

To conduct performance tests we selected a channel, FPSRussia, which has significant content but is also inactive to ensure consistency between each run. Our YouTube Data API crawler collected information on the channel, videos of the channel, and comments. At the time of processing there were 115 videos and 940,330 comments. We ran 4 tests ranging from a single process to 20 processes.

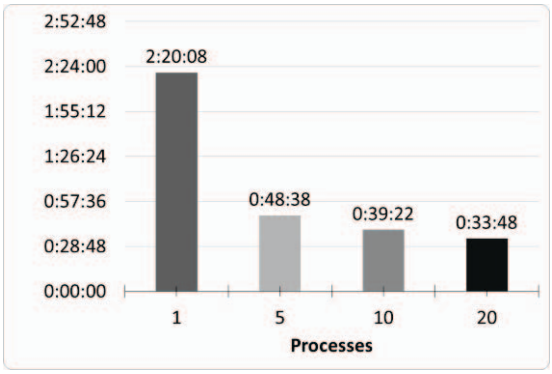

Figure 3. Performance results for 1, 5, 10 and 20 processes

Our performance tests show (Figure 3) a 400% decrease in processing time from 1 to 20 processes while the biggest gain in performance came between 1 and 5 processes. With more than 5 processes there is only marginal improvements, which could be due to I/O bottlenecks. Our performance tests show that parallelizing YouTube API Crawler, even with just a few extra processes, will dramatically reduce processing time.

V. CONCLUSION AND FUTURE WORK

The Information generated by social media platforms continues to increase. With the growing number of researchers conducting social studies, there must be a focus on improving data collection tools. In support of social studies researchers, we have developed parallelized YouTube data collection. By using parallel processing for YouTube data collection, we were able to see a 400% decrease in processing time, allowing for more data collection in shorter periods of time. Such improvements to performance result in faster processing time, allowing for research on larger scopes of YouTube Data.

Our methodology for parallelizing data collection is not limited to YouTube and could be applied to other social platforms such as Blogs, Twitter, or Reddit. For example, a blog crawler must visit a list of sites and scrape relevant data. This process is similar to sending and processing YouTube Data API requests and could be parallelized to optimize data collection.

There are also optimizations to make on the parallelized YouTube crawler. Often when crawling, the processing pool can sit idle at the end of a job waiting for just a single long tail process to complete. One method to overcome this is to replace the unordered iterative pools with asynchronous pools. Asynchronous pools would allow the workers to move onto the next task instead of sitting idle.

Asynchronous pools might also open options for improved parallel processing. Within the YouTube data collection, there are dependent and independent tasksf pools, one could structure the crawler to work on these independent tasks at the same time, improving processing speed.

ACKNOWLEDGMENT

This research is funded in part by the U.S. National Science Foundation (OIA-1920920, IIS-1636933, ACI-1429160, and IIS-1110868), U.S. Office of Naval Research (N00014-10-1-0091, N00014-14-1-0489, N00014-15-P-1187, N00014-16-1-2016, N00014-16-1-2412, N00014-17-1-2605, N00014-17-1-2675, N00014-19-1-2336), U.S. Air Force Research Lab, U.S. Army Research Office (W911NF-16-1-0189), U.S. Defense Advanced Research Projects Agency (W31P4Q-17-C-0059), Arkansas Research Alliance, and the Jerry L. Maulden/Entergy Endowment at the University of Arkansas at Little Rock. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding organizations. The researchers gratefully acknowledge the support.

REFERENCES

[1] "The rise of social media," *Our World in Data*. [Online]. Available: https://ourworldindata.org/rise-of-social-media. [Accessed: 09-Mar-2020].

[2] "What is the total size (storage capacity) of YouTube, and at what rate is it increasing? How is Google keeping up with the increasing demands of Youtube's capacity, given that thousands of videos are uploaded every day? - Quora." [Online]. Available: https://www.quora.com/What-is-the-total-size-storage-capacity-of-YouTube-and-at-what-rate-is-it-increasing-How-is-Google-keeping-up-with-the-increasing-demands-of-Youtube%E2%80%99s-capacity-given-that-thousands-of-videos-are-uploaded-every-day. [Accessed: 09-Mar-2020].

[3] M. N. Hussain, K. K. Bandeli, S. Tokdemir, S. Al-khateeb, and N. Agarwal, "Understanding digital ethnography: socio-computational analysis of trending YouTube videos," in *The Eight International Conference on Social Media Technologies, Communication, and Informatics*, 2018.

[4] M. N. Hussain, S. Tokdemir, N. Agarwal, and S. Al-Khateeb, "Analyzing Disinformation and Crowd

Manipulation Tactics on YouTube," in *2018 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, 2018, pp. 1092–1095.
[5] A. Porreca, F. Scozzari, and M. Di Nicola, "Using text mining and sentiment analysis to analyse YouTube Italian videos concerning vaccination," *BMC Public Health*, vol. 20, no. 1, pp. 1–9, 2020.
[6] N. Singh, L.-M. Browne, and R. Butler, "Parallel astronomical data processing with Python: Recipes for multicore machines," *Astron. Comput.*, vol. 2, pp. 1–10, 2013.
[7] G.-P. Heine, T. Woltron, and A. Wöhrer, "Towards a Scalable Data-Intensive Text Processing Architecture with Python and Cassandra," *DATA Anal. 2018*, p. 25, 2018.
[8] "YouTube Data API," *Google Developers*. [Online]. Available: https://developers.google.com/youtube/v3. [Accessed: 18-Feb-2020].
[9] "YouTube Data API (v3) - Quota Calculator," *Google Developers*. [Online]. Available: https://developers.google.com/youtube/v3/determine_quota_cost. [Accessed: 09-Mar-2020].
[10] "Client Libraries | YouTube Data API," *Google Developers*. [Online]. Available: https://developers.google.com/youtube/v3/libraries. [Accessed: 09-Mar-2020].
[11] "YouTube API Services Terms of Service," *Google Developers*. [Online]. Available: https://developers.google.com/youtube/terms/api-services-terms-of-service. [Accessed: 09-Mar-2020].
[12] "multithreading - Why Was Python Written with the GIL?," *Software Engineering Stack Exchange*. [Online]. Available: https://softwareengineering.stackexchange.com/questions/186889/why-was-python-written-with-the-gil. [Accessed: 09-Mar-2020].
[13] "Has the Python GIL been slain?" [Online]. Available: https://hackernoon.com/has-the-python-gil-been-slain-9440d28fa93d. [Accessed: 09-Mar-2020].
[14] Dansimp, "Troubleshoot port exhaustion issues - Windows Client Management." [Online]. Available: https://docs.microsoft.com/en-us/windows/client-management/troubleshoot-tcpip-port-exhaust. [Accessed: 09-Mar-2020].
[15] "TIME_WAIT and its design implications for protocols and scalable client server systems - AsynchronousEvents." [Online]. Available: http://www.serverframework.com/asynchronousevents/2011/01/time-wait-and-its-design-implications-for-protocols-and-scalable-servers.html. [Accessed: 09-Mar-2020].
[16] "pickle — Python object serialization — Python 3.8.2 documentation." [Online]. Available: https://docs.python.org/3/library/pickle.html. [Accessed: 09-Mar-2020].
[17] *dill: serialize all of python*. .
[18] *pathos: parallel graph management and execution in heterogeneous computing*. .
[19] "16.6. multiprocessing — Process-based 'threading' interface — Python 2.7.17 documentation." [Online]. Available: https://docs.python.org/2/library/multiprocessing.html. [Accessed: 09-Mar-2020].
[20] "pathos module documentation — pathos 0.2.6.dev0 documentation." [Online]. Available: https://pathos.readthedocs.io/en/latest/pathos.html. [Accessed: 09-Mar-2020].
[21] *tqdm/tqdm*. tqdm developers, 2020.