Parallel Programming: Email Address Web Scraper

Paolo Espiritu, Jason Jabanes
De La Salle University - Manila
Manila, Philippines
paolo_edni_v_espiritu@dlsu.edu.ph, jason_jan_jabanes@dlsu.edu.ph

***Abstract*—In this paper, a web scraper for De La Salle University's staff directory page will be implemented using parallel programming concepts such as multithreading and multiprocessing. The performance of these two variations will be evaluated by how much information is gathered by increasing the number of threads or increasing the scraping time of the web scraper.**

***Index Terms*—Web scraper, Selenium, Beautiful Soup, Parallel Programming, Python**

I. Introduction

A. Project Specifications

Email address web scraping is a process of extracting email addresses from websites. It has been a valuable tool for many businesses and organizations. However, the traditional approach of using single-threaded programs to scrape emails can be slow and inefficient when dealing with large amounts of data. This paper presents a parallel programming approach to email address web scraping that utilizes multiple threads and processes in order to improve performance and scalability. Additionally, it should be noted that this project is limited to only using the official De La Salle University website for scraping.

This paper will discuss the steps necessary to build an email address web scraper in Python, including selecting appropriate packages and libraries such as BeautifulSoup, Selenium, and requests, writing code that extracts the desired emails from pages on a website and other necessary information, and finally storing those results into a text file.

This project will use two parallel programming approach namely multithreading and multiprocessing, whereby multiple processes or threads can be executed simultaneously. Multiprocessing involves the simultaneous execution of multiple independent processes on a single computer system, typically using multiple CPUs to increase processing power. Multithreading is the ability for an operating system process to split itself into several sub-processes (threads) that run independently but share common resources such as memory space [1].

B. Program Input and Output

The following arguments must be provided as input to the program:

- `base_url`: Homepage URL of DLSU website
- `scrape_time`: Scraping time in minutes
- `num_threads`: Number of threads / process to use

When the program is finished, it should also output the following:

- Text file that contains email and its associated name, office, department or unit in CSV format
- Text file that contains statistics of the website: URL, number of pages scraped and number of email addresses found

II. Program Implementation

The programming language that will be used to create the e-mail address web scraper is Python. Two implementations of the web scraper will be discussed in this paper. The initial implementation utilized the `threading` library to instantiate multiple threads that will scrape each personnel page simultaneously. To further optimize the performance of the web scraper, the `multiprocessing` module will be used instead of `threading` to create multiple processes that will execute the same task. A thorough discussion regarding this approach is presented in II-D2.

A. Parallel Programming

`threading` and `multiprocessing` are two libraries that allow Python programs to take advantage of parallel processing. `threading` is a library that was utilized which allows multiple threads (tasks) to run concurrently within the same program. `multiprocessing` library ,on the other hand, was also utilized to spawn multiple processes at once.

Lock variables `shared_resource_lock_email` and `shared_resource_lock_pages` were used to ensure mutual exclusion on shared variables namely `num_emails_found` and

num_pages_scraped respectively. These variables are used to count the overall number of email addresses found and number of pages scraped that each thread was able to provide in the allotted amount of time.

Two queue variables were utilized for this project. The queue variable, personnel_id, was used to store the unique ids used to access their individual webpages. Another queue variable, personnel_details was used to store each of the personnel's scraped details such as full name, department, and e-mail address. Queue objects do not need locks or semaphores because they are designed to be thread-safe. This ensures that multiple threads accessing a queue object will never conflict with each other as each thread can process one item from the front of the queue before moving onto another thread's item.

B. Web Scraping

Upon analyzing the HTML structure of each webpage within the DLSU website, it is evident that the formatting differs from one page to another. With this, it will only be possible to collect e-mail addresses with additional information from each webpage if a specific scraper will be coded for each varying webpage. Hence, the e-mail address web scraper will instead focus on the contents of the staff directory page of the DLSU website (https://www.dlsu.edu.ph/staff-directory/) since this page contains abundant information about DLSU personnel which includes their full name, e-mail address, and department. In addition, the HTML structure of each personnel page in the staff directory is also consistent which makes it easier to scrape relevant information. However, the staff directory as well as individual personnel pages were being dynamically loaded by JavaScript functions which means the scraper has to wait before extracting the necessary information. Given this, it is significant to explore possible libraries that will help resolve the problem to successfully and efficiently scrape the DLSU staff directory page.

The Python libraries used for scraping data are Selenium and BeautifulSoup. Selenium is used to scrape complex pages that performs dynamic loading at a cost of higher computational resources [2]. This module is important since the relevant data in the staff directory and individual personnel pages will only be available once the page has fully loaded. Thus, it allows the scraper to wait for the JavaScript functions to finish first before extracting the necessary data. Due to the computational requirements of Selenium, the requests library is used to optimize data extraction from the staff directory page which will be further discussed in II-D1.

After waiting for the page to be rendered, the BeautifulSoup module is utilized for creating a parse tree to effectively extract data from parsed HTML and XML documents. Since the relevant data is found in HTML format, the module was used to locate various HTML elements to extract it. Although Selenium may also be used to do the same task, it is essential to use it only when necessary to lessen resource usage [2].

C. Code Implementation

The e-mail web scraper starts by asking for inputs as shown in I-B. After the user provides the necessary inputs for the program, the staff directory will be loaded by using the base_url. Upon visiting the page, the information for each personnel is being loaded via JavaScript; hence, there is a need to use the Selenium library to wait for the page to load before scraping it. When the personnel list is fully rendered, it can be observed that there is also a *Load More* button to retireve more personnel information from the DLSU database which is also controlled by JavaScript functions. With this, Selenium is utilized to click the *Load More* continuously until it is no longer available or it exceeded a set percentage of the scrape_time. When the task of Selenium finishes, the page will then be parsed by BeautifulSoup to efficiently extract data from HTML elements. Since each personnel page is loaded using the *value* embedded with the button element, the *value* attribute was scraped from each button element of each personnel in the staff directory page. Finally, a url is created with each button value scraped from the staff directory page then it will be added to the global queue. The pseudocode for this process is shown in Algorithm 1.

However, this algorithm is extremely slow as it is heavily dependent on the Selenium library which requires vast computational resources. Thus, an optimization technique will be introduced in II-D1 which utilized the requests module of Python.

Following that, the number of threads supplied by the user is used to create and start the threads.The creation of multiple threads for the e-mail address scraping task is shown in Algorithm 2.

After creating the threads, Algorithm 3 illustrates how each thread will process the urls from the global queue. Since each personnel page is loaded using JavaScript, Selenium will be used to wait for the page to load. The timeout set for waiting is 2 minutes and if the page is not loaded within the time provided, the url will be put back into the queue. On the other hand, if the page successfully loaded, BeautifulSoup will parse its HTML content to scrape the necessary data from the personnel

Algorithm 1 Web Scraping Staff Directory v1

```
input_personnel ← global queue of individual personnel page url
base_url ← DLSU home page url
scrape_time ← time limit of web scraper
driver ← Chrome webdriver object
soup ← BeautifulSoup object

driver ← loads staff directory page using base_url
while Load More is clickable AND current run time does not exceed 20% of scrape_time do
    click Load More
end while
soup ← parse driver page
locate HTML elements to get information using soup
personnel_list ← append button value of each personnel in staff directory
for each p ∈ personnel_list do
    personnel_page ← create url with button value
    input_personnel ← append url
end for
```

Algorithm 2 Thread Creation

```
threads ← list of threads
num_threads ← number of threads to create
scrape_time ← time limit of web scraper
for i = 0 to num_threads do
    thread ← Personnel_page(i)
    append thread to threads
    Start(thread)
end for
for thread in threads do
    Join(thread)
end for
```

page. With this, the full name, e-mail, and department can be extracted and stored in a dictionary. This will be placed inside a global queue variable to store the outputs of different threads. Moreover, the variable `num_emails_found` is incremented only if the personnel has e-mail information available. After scraping the personnel page successfully, the variable `num_pages_scraped` is also incremented. In order for threads to safely modify these variables, locks such as `shared_resource_lock_email` and `shared_resource_lock_pages` are used for the e-mail counter and scrape page counter respectively.

Algorithm 3 Web Scraping Personnel Page

```
final_personnel ← global queue of scraped personnel information
num_pages_scraped ← no. of pages scraped
num_emails_found ← no. of e-mails found
shared_resource_lock_pages ← lock for updating num_pages_scraped
shared_resource_lock_email ← lock for updating num_emails_found
driver ← Chrome webdriver object
soup ← BeautifulSoup object

while input_personnel is not empty do
    personnel ← get url from input_personnel
    driver ← loads page using url in personnel
    wait for page to load (time out in 2 minutes)
    if page loaded then
        soup ← parse driver page
        locate HTML elements to get information using soup
        personnel_info ← store full name, e-mail, and department in a dictionary
        if e-mail exists then
            acquire(shared_resource_lock_email)
            num_emails_found+ = 1
            release(shared_resource_lock_email)
        else
            pass
        end if
        final_personnel ← put personnel_info to store results of different threads
        acquire(shared_resource_lock_pages)
        num_pages_scraped+ = 1
        release(shared_resource_lock_pages)
    else
        input_personnel ← put back personnel into the queue
    end if
end while
```

D. Optimization Strategies

1) `requests` library for scraping staff directory: Due to the demanding computational requirements of `Selenium`, it was necessary to explore another approach to efficiently scrape the button values in the staff directory page. With the use of the `requests` library, the JSON response when the page has loaded can be exploited by using the DLSU API url. The url can be obtained by performing the following steps on the staff directory page:

Step 1: Open Developer Tools.
Step 2: Go to Network Tab and check Fetch/XHR.
Step 3: Check the Response tab of an item to see if it contains the important information.

Step 4: Go to the Headers tab of the selected item and copy the Request URL excluding the payload.

By utilizing the JSON response of the staff directory page, the e-mail address web scraper will be faster since it will no longer need to interact with the web browser with `Selenium`. With `requests`, the program will be able to receive the data directly from the server without waiting for the JavaScript functions to render the page. Algorithm 4 exhibits the pseudocode for the optimized web scraping of the staff directory page.

Unfortunately, the same optimization technique cannot be done when scraping individual personnel pages since there are is no response that shows the full name, e-mail, and department of each personnel. Thus, `Selenium` and `BeautifulSoup` will still be used for scraping information from each personnel page.

Algorithm 4 Web Scraping Staff Directory v2

```
input_personnel ← global queue of individual personnel page url
payload ← initialized payload parameters for request
headers ← intialized headers parameters for request
url ← DLSU API url used in staff directory
session ← requests.Session() object
MAX_PAGES ← total number of pages in staff directory

for 1 to MAX_PAGES do
    session ← load url with payload and headers
    final ← store personnel attribute of session JSON response
    for each personnel ∈ final do
        personnel_page ← create url with personnel id value
        input_personnel ← append url
    end for
end for
```

2) Multithreading vs. Multiprocessing: The initial implementation of the e-mail web scraper made use of multithreading since `threading` is best suited for I/O-bound tasks [3]. This include tasks performed by the program such as HTTP requests and writing to a file [4]. However, there is a limitation for Python when it comes to `threading` because of the Global Interpreter Lock (GIL) [3]. It is a lock that only permits one thread to control the Python interpreter at any given time. Thus, this indicates that only one thread can ever be in a state of execution which prevents the threads from running in parallel [5]. Since there are also CPU-bound tasks in the program such as loading JSON data and parsing scraped HTML content [6], it was necessary to investigate a `multiprocessing` approach for the program as it may further improve the results of the web scraper. Algorithm 5 shows the pseudocode of how processes were created in the program.

With this, the multithreading and multiprocessing implementations will be compared in the results section of this paper.

Algorithm 5 Process Creation

```
processes ← list of processes
num_processes ← number of processes to create
scrape_time ← time limit of web scraper

for i = 0 to num_processes do
    process ← Personnel_page(i)
    append process to processes
    Start(process)
end for
for process in processes do
    Join(process)
end for
```

III. Results

The experiment setup involves observing and comparing the performance of multiple threads and multiple processes based on increasing scrape time and increasing number of threads/processes.

The setup for varying scrape time are as follows:

- Website - https://www.dlsu.edu.ph/
- Scrape time in minutes - [1, 2, 3, 4, 5]
- Number of threads/processes - 8

While the setup for varying number of threads/processes are as follows:

- Website - https://www.dlsu.edu.ph/
- Scrape time in minutes - 2
- Number of threads/processes - [1, 2, 4, 8, 16]

It can be seen from the experimental setup for varying scrape time that the number of threads/processes remains constant while the opposite is applied to varying number of threads where scrape time remains a constant value. Moreover, scrape time and number of threads/processes vary across the supplied range of values per test.

A. Increasing Scrape Time

Table I shows the results for both multithreading and multiprocessing when scrape time is increased for each test. The results of the comparison between multiple threads and multiple processes showed that increasing the scrape time led to an increase in number of pages scraped and number of e-mail

addresses found. It is also observed that multiprocessing scraped more number of pages and found more number of e-mail addresses when compared to multithreading, that is except for when scrape time was set to 3 minutes where they both had equal amounts of number of pages scraped and number of e-mail addresses found.

TABLE I: Increasing Scrape Time

		Threads		Processes	
#	Input	Pages	Emails	Pages	Emails
1	1, 8	21	12	25	16
2	2, 8	33	19	40	24
3	3, 8	56	34	56	34
4	4, 8	66	40	69	41
5	5, 8	71	42	77	46

B. Increasing Number of Threads/Processes

Table II shows the results for both multithreading and multiprocessing when number of threads/processes is increased for each test. Similar to the previous experiment, increasing the number of threads or processes resulted in more pages being scraped and e-mail addresses found. In particular, when comparing a single thread versus two threads, four threads, eight threads, and sixteen threads, it was found that there was a significant increase in time for each successive step. Once more, similar to the previous experiment, when increasing the number of threads and processes, it was found that multiprocessing had a better performance compared to multithreading in terms of number of pages scraped and e-mail addresses found. It can be concluded from these results that when it comes to achieving true parallelism, multiprocessing is more effective than multithreading.

TABLE II: Increasing number of threads/processes

		Threads		Processes	
#	Input	Pages	Emails	Pages	Emails
1	2, 1	6	3	8	5
2	2, 2	13	8	14	9
3	2, 4	25	16	30	17
4	2, 8	33	19	47	30
5	2, 16	48	31	62	37

IV. Conclusion

In conclusion, multithreading and multiprocessing are two powerful tools for email address web scraping. They allow the process to be completed in a fraction of the time it would take otherwise, while still maintaining accuracy. Multithreading and multiprocessing also reduce resource utilization when compared to regular single-threaded processes, allowing for more efficient use of hardware resources. It is worth noting that all threads and processes are synced to prevent interference. In order to accomplish this, synchronization objects like locks and queues are used. By utilizing these techniques, web scrapers can quickly and accurately gather large amounts of data from websites with minimal effort.

According to the results, it is clear that multiprocessing is the superior approach for email address web scraping. This is due to its ability to simultaneously execute multiple processes at once, allowing for faster and more efficient data collection than multithreading can provide. Additionally, multiprocessing requires fewer resources than multithreading, making it both a cost-effective and time-effective solution. As such, when considering an approach to email address web scraping, multiprocessing should be strongly considered as the preferred option.

References

[1] S. Dutta, "An overview on the evolution and adoption of deep learning applications used in the industry," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 8, no. 4, p. e1257, 2018.

[2] G. Pornaras, "Selenium vs. beautiful soup: A full comparison," https://www.blazemeter.com/blog/selenium-vs-beautiful-soup-python#:\~:text=The\%20main\%20difference\%20between\%20Selenium,the\%20scope\%20of\%20the\%20project., 2021, accessed: 2022-12-07.

[3] B. Solomon, "Async io in python: A complete walkthrough," https://realpython.com/async-io-python/, 2019, accessed: 2022-12-08.

[4] Baeldung, "Guide to the "cpu-bound" and "i/o bound" terms," https://www.baeldung.com/cs/cpu-io-bound, 2022, accessed: 2022-12-08.

[5] A. Ajitsaria, "What is the python global interpreter lock (gil)?" https://realpython.com/python-gil/, 2018, accessed: 2022-12-08.

[6] Scrapfly, "Web scraping speed: Processes, threads and async," https://scrapfly.io/blog/web-scraping-speed/#:~:text=In\%20web\%20scraping\%2C\%20we\%20primarily,\%2Dbound\%20and\%20CPU\%2Dbound.&text=We\%20also\%20encounter\%\20CPU\%20tasks,language\%20parsing\%2C\%20and\%20so\%20on.&text=In\%20web\%20scraping\%2C\%20we\%20encounter\%20both\%20of\%20these\%20performance\%20challenges., 2022, accessed: 2022-12-08.