Designing an Efficient Multiprocess Email Address Scraper for the De La Salle University Website Staff Directory

Mark Edward M. Gonzales
Department of Software Technology
De La Salle University
Manila, Philippines
mark_gonzales@dlsu.edu.ph

Hans Oswald A. Ibrahim
Department of Software Technology
De La Salle University
Manila, Philippines
hans_oswald_ibrahim@dlsu.edu.ph

***Abstract*—With the volume of data in the Web, scraping tools enable users to automate the bulk collection of data for subsequent use and analysis. The performance of these tools, *i.e.,* the amount of data that they can collect in a given period of time, is an important consideration, and conventional serial programming is inadequate to meet performance expectations. This paper presents a multiprocess email address scraper for the De La Salle University website staff directory. It models the scraping task as a multiple producer – multiple consumer problem. The producers map to different department directories, retrieve the personnel IDs from their assigned department directory, and store them in a synchronized queue. Concurrently, the consumers get IDs from the queue and visit the individual web pages of the staff members to scrape the necessary details. Running our proposed approach with five threads achieves a $7.22\times$ superlinear speedup compared to serial execution. Further experiments show that it achieves better scalability and performance than baseline parallel programming approaches that scrape from the root directory.**

***Index Terms*—Web scraping, parallel programming, multiprocessing, synchronized queue**

I. Introduction

Regarded as the information superhighway, the Internet serves not only as the global connection of computers but also facilitates the unprecedented volume, velocity, and variety of data — both structured and unstructured — on the Web. The first step to utilizing and analyzing these data is to collect them from potentially multiple web pages. Scraping tools automate this bulk collection of data. Scraped data have been extensively used to derive insights related to multiple areas, such as pedagogy [1], consumer analytics [2], scientific literature [3], [4], and weather forecasting [5].

Email scraping is a particular form of web scraping that focuses primarily on email addresses. Provided that it is done within an established ethical or legal framework, it can help in reaching out to wider target audiences and conducting cost-effective marketing campaigns [6]. In the educational setting, it can be employed to automatically and expeditiously create department-wide or college-wide mailing lists for sending centralized announcements or advisories. Email scraping also reduces errors due to typographical oversights from manual email encoding.

An important consideration in the design of scrapers is their performance, *i.e.,* the amount of data they can collect in a given period of time [7]. The conventional serial programming techniques are inadequate to meet users' performance expectations, especially as the scale increases. By capitalizing on multiple concurrently executing processes, parallel programming techniques provide a way to speed up scraping by several orders of magnitude.

This paper presents a multiprocess email address scraper for the De La Salle University website staff directory. Combining both functional and data decomposition, our proposed approach models the scraping task as a multiple producer – multiple consumer problem. The set of personnel IDs in the staff directory is divided by department. Multiple producer subprocesses are mapped to different department directories; each producer retrieves the personnel IDs from its assigned department directory and stores them in a synchronized queue. Concurrently, the IDs are dequeued by consumer subprocesses, which use them to visit the staff members' individual web pages and scrape the required details from there.

Executing this proposed approach with five threads achieves a $7.22\times$ superlinear speedup compared to serial execution. Additional experiments also show that it achieves better scalability and performance than baseline approaches that also employ parallel programming but scrape from the root staff directory instead of the department directories.

The rest of our paper is organized as follows. Section II explains the program implementation. The results of the experiments conducted are reported and discussed in Section III . Finally, concluding remarks are given in Section IV.

II. Program Implementation

This section discusses the dependencies of our program and its input parameters and output files, the anatomy of the DLSU website staff directory, our application of parallel programming techniques in our baseline approaches and in our proposed approach, and details related to the scraping statistics and timer.

TABLE I
INPUT PARAMETERS OF THE PROGRAM. CURRENTLY, OUR PROGRAM IS LIMITED TO AND HAS BEEN OPTIMIZED FOR SCRAPING EMAIL ADDRESSES FROM THE DLSU WEBSITE STAFF DIRECTORY.

Parameter	Description
`url`	(Dependent on mode, default = https://www.dlsu.edu.ph/staff-directory/) URL of the website to be scraped
`mode`	(Optional, default = 1) 1 refers to our proposed approach (Section II-F); 2 and 3 refer to our first and second baseline approaches, respectively (Sections II-D and II-E)
`scraping_time`	Maximum scraping time in minutes
`num_threads`	(Optional, default = 5) Number of threads. Setting this to 1 is equivalent to serial execution. Otherwise: • If `mode` is set to 1, then `num_producers` and `num_consumers` will be asked. • If `mode` is set to 2 or 3, then there is automatically 1 producer, `num_threads`−2 consumers, and 1 writer.
`num_producers`	Number of producer threads in mode 1. Ignored if mode is set to 2 or 3, or if `num_threads` is set
`num_consumers`	Number of consumer threads in mode 1. Ignored if mode is set to 2 or 3, or if `num_threads` is set

A. Dependencies

Our email scraper was written in Python 3.8, and the `multiprocessing` [8] library was employed for spawning parallel processes without the single-thread restriction imposed by the Global Interpreter Lock. While `multiprocessing` technically spawns subprocesses, the distinction between threads and subprocesses is ignored from this section onwards in the interest of simplicity.

Since the DLSU website is dynamically loaded, automated browser interaction is necessary for the retrieval of the pertinent web pages. To this end, the open-source library Selenium 4.7.2 [9] was used. During our program's first run, it automatically installs the latest compatible version of Chrome WebDriver [10] on the machine using the Webdriver Manager library [11]. Our scraper runs the Chrome WebDriver in headless mode to control the browser programmatically while keeping its graphical user interface invisible.

B. Running the Program

Our program executes purely on the command-line interface and can be configured with multiple parameters, as enumerated in Table I. It produces two text files as the output:

- A text file that contains the email address and its associated name, office, department, or unit (hereinafter referred to as the *department*) in comma-separated values format
- A text file that contains statistics about the scraping, particularly the number of pages scraped, the number of email addresses found, and the URLs scraped

C. DLSU Website Staff Directory Anatomy

The staff directory page displays at most 24 staff members at a time, with the picture, name, position, and department of each staff member displayed in a card. To display more staff members, the Load More button at the bottom of the page has to be clicked (Figure 1).

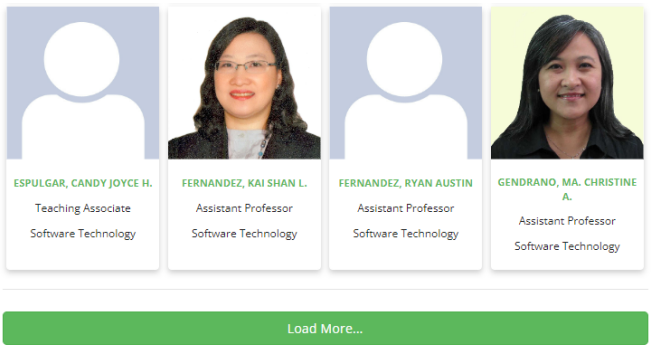

Fig. 1. DLSU Staff Directory. Clicking the Load More at the bottom of the page dynamically fetches at most 24 entries from the server (or the cache) and appends them to the display.

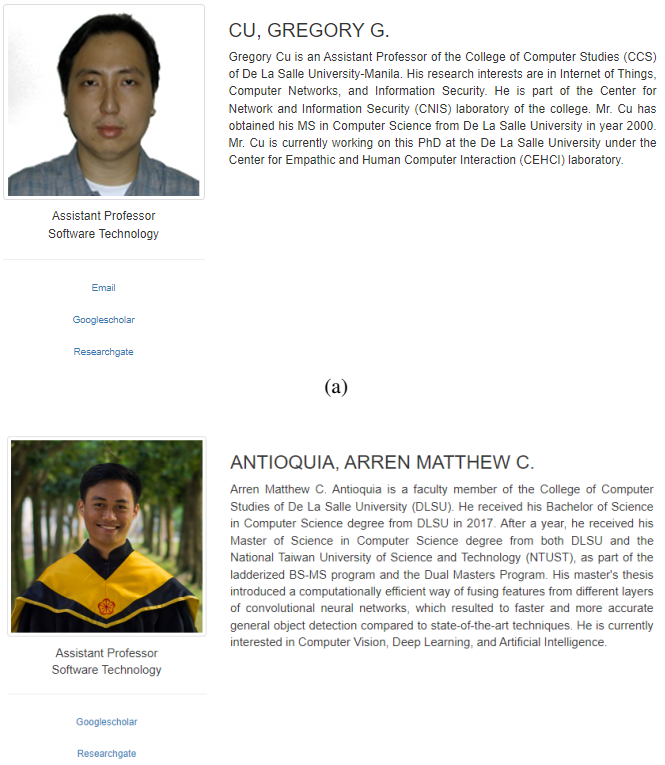

Fig. 2. Individual Web Page of Staff Member. The email address of a staff member is located on their individual web page; it is the first among several provided profile links (Figure 2a). However, some staff members do not have their emails recorded on the DLSU website (Figure 2b.

Since the email address of a staff member is not included in their card, their individual web page has to be visited. As seen in Figure 2a, their email address can be obtained by fetching the value to which the Email hyperlink points. Their name and department are also found on this page. However, it is to be noted that some staff members do not have their email addresses on the DLSU website; an example is shown in Figure 2b.

Algorithm 1 Clicking the Load More Button. Since the button does not have any ID attribute, it is fetched via its class names, which uniquely identify it among the other page elements. Moreover, since it is located at the bottom of the page, it has to be scrolled into view before it can be clicked.

Require:
`driver` – Web driver
`DELAY` – Maximum wait time for condition

Constants:
`LOAD_MORE_CLASS = ".btn.btn-success.btn-lg.btn-block.text-capitalize"`

```
1: load_more = WebDriverWait(driver, DELAY).until(EC.element_to_be_clickable(
      (By.CSS_SELECTOR, LOAD_MORE_CLASS)))
2: driver.execute_script("arguments[0].scrollIntoView();", load_more)
3: load_more.click()
```

Algorithm 2 Visiting a Staff Member's Web Page. The web page is directly visited by appending the personnel parameter (query) and ID of the staff member to the URL, effectively performing a `GET` request. The personnel ID is not exposed in the front-end; it is fetched from a specific value attribute found in the source code.

Require:
`driver` – Web driver
`DELAY` – Maximum wait time for condition
`URL` – URL of the staff directory with the `GET` parameter `?personnel=` appended

```
1: personnels = WebDriverWait(driver, DELAY).until(
      EC.presence_of_all_elements_located((By.NAME, "personnel")))
2: for personnel in personnels:
3:    personnel_id = personnel. get_attribute("value")
4:    driver.get(URL + personnel_id)
```

Since the elements of the staff directory and individual web pages are dynamically loaded, another important consideration when scraping is waiting for the target elements (*e.g.,* load button) to load. To this end, the explicit waiting functionality of Selenium, which forces the web driver to freeze the calling thread until the expected condition is satisfied [12]. A maximum wait time was set to prevent our scraper from waiting indefinitely. The scripts are provided in Algorithms 1 to 3.

D. Parallel Programming: First Baseline Approach

The first baseline approach models the scraping task per se as a single producer – multiple consumer problem. The high-level block diagram, as well as the motivation for this parallel programming approach, is given in Figure 3.

Formally, let PERSONNELQUEUEPRODUCER be a producer subprocess that repeatedly clicks the Load More button (*i.e.,* it executes Algorithm 1) until half of the user-specified scraping time elapses. Afterwards, it fetches all the personnel IDs (line 1 of Algorithm 2) and stores them in a synchronized queue PERSONNELQUEUE. The `multiprocessing` library offers an implementation of this synchronized data structure.

Let the consumer subprocesses be the set of all PERSONNELQUEUECONSUMER$_i$ (where i ranges from 1 up to the number of consumer subprocesses). Each of them concurrently gets a personnel ID from PERSONNELQUEUE, visits the individual web page of the associated staff member (line 4 of Algorithm 2), and extracts the required details from the web page. The required details are then bundled together into an object SCRAPEDDETAILS and placed in a synchronized queue WRITEQUEUE. A single subprocess WRITERTHREAD concurrently gets an instance of SCRAPEDDETAILS from WRITEQUEUE and writes its contents to a text file.

Some design decisions in the creation of this pipeline may have to be justified. First, a personnel ID should be fetched at most once; otherwise, it will be duplicated in the text file. Having a single PERSONNELQUEUEPRODUCER removes the complexity of having to keep track which personnel ID has already been fetched.

Second, since this baseline approach involves fetching personnel IDs from the root of the staff directory, it is possible that the PERSONNELQUEUEPRODUCER spends all the user-specified scraping time repeatedly clicking the Load More Button, as there are more entries to be loaded. This will result in no URLs scraped. Clearly, there are several solutions to this issue; the workaround presented here, which is to repeatedly click the button for only half the specified scraping time, is deliberately kept simple for experimentation purposes.

Algorithm 3 Extracting the Email, Name, and Department. Since these details are not marked by ID attributes, the name is fetched by looking for the sole third-level heading tag. The email address is fetched by looking for the email anchor tag via its class names and extracting the hyperlink destination. The department is fetched by looking for the unordered list below the staff member's picture (Figure 2) via its class names and getting the second list item enclosed inside a span (the first list item is the position of the staff member in the department).

Require:
`driver` – Web driver
`DELAY` – Maximum wait time for condition

Constants:

```
EMAIL_CLASS = ".btn.btn-sm.btn-block.text-capitalize"
POSITION_CLASS = "list-unstyled.text-capitalize.text-center"
```

```
1: email = driver.find_element(By.CSS_SELECTOR, EMAIL_CLASS)
```

Check if the profile link captured is an email

```
2: if "mailto" in email.get_attribute("href"):
3:     name = WebDriverWait(driver, DELAY).until(EC.visibility_of_element_located(
           (By.TAG_NAME, "h3")))
4:     position = WebDriverWait(driver, DELAY).until(EC.visibility_of_element_located(
           (By.CLASS_NAME, POSITION_CLASS)))
5:     position_info = position.find_elements(By.TAG_NAME, "li")
```

Go through the details in the unordered list located below the staff member's picture

```
6:     affiliation = []
7:     for info in position_info:
8:         if "span" in info.get_attribute("innerHTML"):
9:             affiliation.append(
                   info.get_attribute("innerHTML")[len("<span>"):-len("</span>")])
```

Store the fetched staff member details

```
10:    personnel_email = email.get_attribute("href")[len("mailto:"):]
11:    personnel_name = name.get_attribute("innerHTML")
12:    personnel_dept = affiliation[1]
```

Third, writing to a text file is an expensive operation, as it requires communicating with the disk. This motivated the delegation of this task to a dedicated WRITERTHREAD; in this manner, once the PERSONNELQUEUECONSUMER has fetched the required details and enqueued them to WRITEQUEUE, it can dequeue another personnel ID from PERSONNELQUEUE instead of spending time on file I/O. Having only a single WRITERTHREAD also avoids the complications of the multiple writers problems. There is no detriment to efficiency, as only one writer is allowed to write on a file at any given time.

E. Parallel Programming: Second Baseline Approach

The second baseline approach is a slight modification to the first baseline approach, as seen in the high-level block diagram in Figure 4. It follows the same logic as the first baseline approach. However, instead of repeatedly clicking the Load More button for half of the user-specified scraping time before proceeding to scraping per se, the second baseline approach clicks Load More, immediately fetches the loaded personnel IDs and puts them to the queue, and repeats this cycle until the user-specified scraping time elapses.

A caveat in the implementation of this approach is that Line 1 of Algorithm 2 returns the personnel IDs of *all* the staff members displayed, not only the newly displayed ones. To this end, a variable local to PERSONNELQUEUEPRODUCER keeps track of the index of the last personnel ID from the previous iteration. The validity of this strategy is justified by the deterministic, in-order behavior of Selenium's retrieval of the personnel IDs, which follows from the web driver having only a single thread of execution [13].

F. Parallel Programming: Our Proposed Approach

Our proposed approach models the scraping task per se as a multiple producer – multiple consumer problem. Instead of scraping from the root staff directory, it concurrently scrapes from the staff directories of the departments. The high-level block diagram, as well as the motivation for this parallel programming approach, is given in Figure 5.

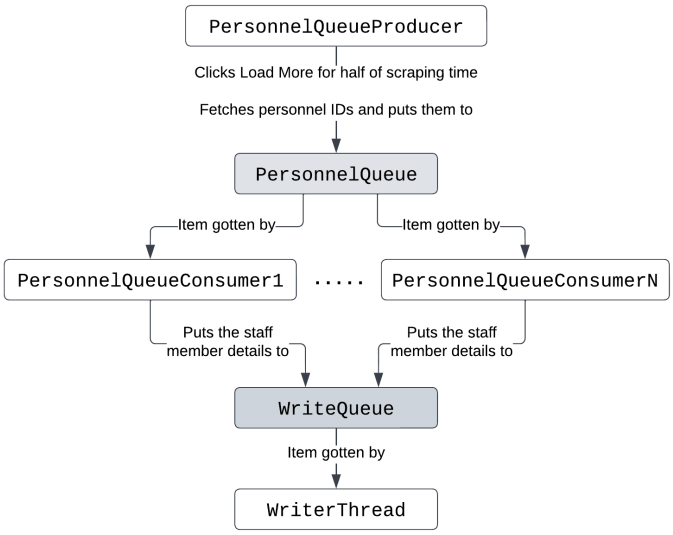

Fig. 3. Block Diagram of First Baseline Approach. The motivation for this approach is that Algorithm 2 can be parallelized by assigning a producer to fetch all the personnel IDs from the root of the staff directory and having multiple consumers visit the individual web pages of the staff members and extract the required details.

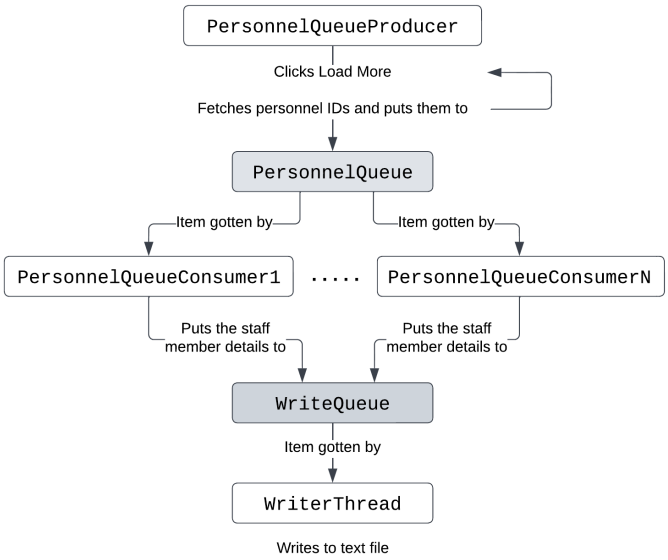

Fig. 4. Block Diagram of Second Baseline Approach. This approach follows the same pattern as in the first baseline, but makes a simple yet critical change to the activity of the `PersonnelQueueProducer` by alternating loading more entries and fetching the personnel IDs, and repeating this cycle until the scraping time elapses.

Formally, the main thread first stores the department codes in a synchronized queue DEPARTMENTQUEUE; our program provides the option to either fetch the department codes from the filter drop-down at the top of the root staff directory page or opt for the faster strategy of hard-coding them (since the University's departments are not expected to change often). The experiments reported in this paper assume the latter.

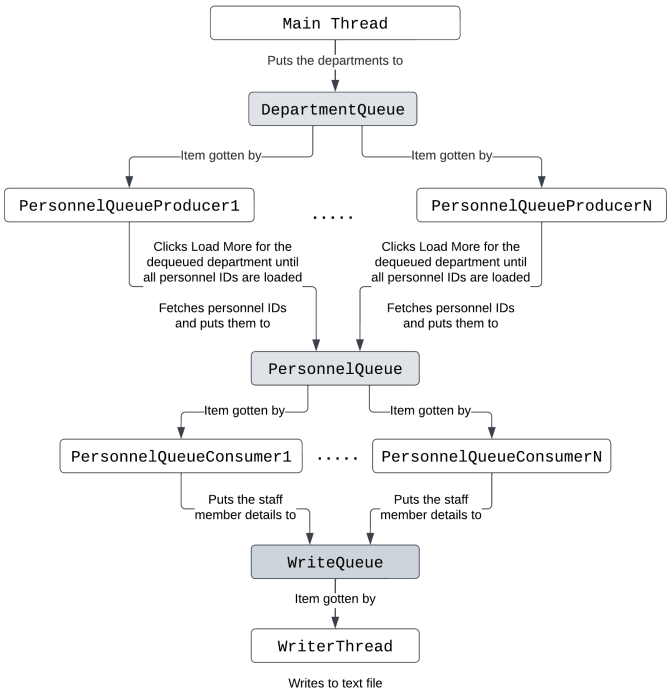

Fig. 5. Block Diagram of Our Proposed Approach. The motivation for this approach is the observation that consumer threads may become idle if there is only a single producer. In this regard, the main feature of this approach is the addition of another layer of domain decomposition to the baseline approaches by dividing the set of personnel IDs in the staff directory by department. This transposes the scraping model into that of multiple producers and multiple consumers

Let the producer subprocesses be the set of all PERSONNELQUEUEPRODUCER$_i$ (where i ranges from 1 up to the number of producer subprocesses). Each of them concurrently gets a department from DEPARTMENTQUEUE and visits this department's staff directory. It repeatedly clicks the Load More button until all the entries have been loaded. Afterwards, it fetches all the personnel IDs and stores them in a synchronized queue PERSONNELQUEUE. It is reasonable for PERSONNELQUEUEPRODUCER$_i$ to fetch the IDs only after all the entries have been loaded since, on average, the Load More button is expected to be clicked only twice.

The functions of the consumer subprocesses (*i.e.,* the set of all PERSONNELQUEUECONSUMER$_i$) and the WRITERTHREAD are the same as in the baseline approaches (explained in detail in Section II-D).

G. Statistics

The scraping statistics (*i.e.,* the number of pages scraped and the number of email addresses found) are also updated in a parallel fashion. In particular, a single manager object (`multiprocessing.Manager`) manages the shared integer variables (`multiprocessing.Value`) corresponding to the statistics. By default, an under-the-hood lock makes each shared integer process-safe [8]; thus, incrementing its value is straightforward.

H. Timer

A timer is required to keep track of the elapsed scraping time. This task is assigned to a separate subprocess `TimerThread`, whose activity is fired at program startup but immediately suspended for the specified scraping time. The manager object described in the previous section (Section II-G) manages the shared integer flag `terminated`, initialized to 0. Once `TimerThread` wakes, it sets the `terminated` to 1, indicating that the scraping time has elapsed.

All the other subprocesses have several breakpoints in their execution to check the value of `terminated`. For instance, PERSONNELQUEUEPRODUCER$_i$ has breakpoints at the start of its activity, before retrieving the personnel IDs, and before enqueueing them. PERSONNELQUEUECONSUMER$_i$ has breakpoints at the start of its activity, before dequeueing a personnel ID, before visiting the associated staff member's web page, before fetching each required information, and before enqueueing them to the WRITEQUEUE. Our implementation guarantees that the value of `terminated` is checked before the next run of a subprocess.

Once the scraping time has elapsed, the contents of all the synchronized queues are also forcibly flushed `close()` method [8], and Python's garbage collection follows.

III. Results and Discussion

To test the performance of the proposed multiprocess email address scraper, experiments were conducted on a local machine with the following specifications:

- Processor: AMD Ryzen™ 7 4800H with Radeon™ Graphics
- Processor Base Frequency: 2.9 GHz
- Memory: 16.0 GB
- Storage: 512 GB Solid-State Drive
- Operating System: Microsoft Windows 10 (64-bit)

The scraping time for all the experiments was fixed at three minutes. Although attempts were made to conduct the experiments when the internet connectivity of the testing machine was fairly stable and the speed of the DLSU website was relatively consistent, some confounding factors, such as those related to the network and the website server, remain.

A. Program Performance

The first set of experiments sought to compare our proposed approach with a serial approach and with the two baseline approaches in terms of the number of URLs scraped. This was selected over the number of emails scraped as the representative performance measure since some staff members do not have their email addresses on the DLSU website.

As seen in Table II, parallelism improved the program's performance. For the first baseline, the speedup after using five threads is $2.83\times$, which is below the ideal $5\times$ speedup. However, for the second baseline and our proposed approach, using five threads registered superlinear speedups, clocking in $8.22\times$ and $7.22\times$ speedups, respectively. These results indicate that email scraping is a task that strongly benefits from the application of parallel programming techniques.

The speedup tapers to linear or sublinear levels as the number of threads in the parallel approaches is increased in increments of five. This is expected since multiple subprocesses introduce a degree of overhead from synchronization mechanisms [14]; for instance, the synchronized queue and value objects implement complex locking semantics under the hood [8], [15]. Introducing additional subprocesses increases the overhead, as more subprocesses have to be synchronized. This synchronization overhead is absent from serial programs, although at the expense of significantly lower performance.

Comparing the performances of the parallel programming approaches, the second baseline approach clocked in the highest performance at fewer threads. However, when the number of threads increased from 15 onwards, it was outperformed by our proposed approach. This result can be attributed to the difference between their producer-to-consumer ratios. In the second baseline, which models the scraping task as a single producer – multiple consumer problem, increasing the number of threads will only increase the number of consumers. These consumers are, thus, left idle while the producer is waiting for more entries to load after clicking the Load More button. Empirical observations showed that loading new entries and displaying them on the staff directory page can be noticeable bottlenecks when the website server slows down.

On the other hand, our proposed approach avoids this problem by implementing another layer of domain decomposition. Instead of collecting personnel IDs from the root staff directory, the set of personnel IDs is divided by department. Multiple producers are assigned to concurrently retrieve personnel IDs, with each producer working on a particular department directory. Since there are multiple producers enqueueing, the idle time of the consumer threads is minimized. Consequently, this strategy also permits better scalability as more departments are added to the website; more threads can also be assigned to the task.

In relation to this, our proposed approach entails finding the optimal balance between the number of producers and consumers. The second set of experiments sought to compare the

TABLE II
COMPARISON OF SERIAL AND PARALLEL APPROACHES. THE VALUES IN THE FIRST ROW PERTAIN TO THE NUMBER OF THREADS AND THE IDEAL SPEEDUP. EACH CELL HAS TWO VALUES: THE NUMBER OF URLS SCRAPED AND THE SPEEDUP VIS-À-VIS THE PREVIOUS NUMBER OF THREADS. THE PRODUCER-TO-CONSUMER RATIO IN OUR APPROACH WAS TUNED VIA GRID SEARCH.

Number of Threads	1	5 5.00×	10 2.00×	15 1.50×	20 1.33×	25 1.25×
Serial	18	N/A	N/A	N/A	N/A	N/A
First Baseline	N/A	51 2.83×	79 1.54×	94 1.18×	142 1.51×	214 1.50×
Second Baseline	N/A	**148** 8.22×	**272** 1.83×	325 1.19×	379 1.17×	405 1.07×
Ours	N/A	130 7.22×	233 1.79×	**352** 1.51×	**382** 1.09×	**477** 1.25×

TABLE III
COMPARISON OF DIFFERENT PRODUCER-TO-CONSUMER RATIOS. ALTHOUGH THERE APPEARS TO BE A DOWNWARD TREND IN PERFORMANCE AS THE NUMBER OF PRODUCERS INCREASES, MORE EXPERIMENTS ARE NECESSARY TO FORM GENERALIZATIONS. THE MOST INFORMED APPROACH TO OPTIMIZE PRODUCER-TO-CONSUMER RATIO MAY BE TO DYNAMICALLY ADJUST IT DEPENDING ON THE LOAD.

Num. of Producers	Num. of Consumers	Num. of URLs Scraped
3	22	477
8	17	395
13	12	341
18	7	238

performances of different producer-to-consumer ratios when their sum is fixed at 25 threads. As seen in Table III, setting the number of producers to 3 and the number of consumers to 22 yielded the highest number of URLs scraped. However, it should be emphasized that this experiment alone is insufficient to formulate generalizations.

Nevertheless, some principles can be gleaned. An overly high number of producers will only populate `PersonnelQueue` but will not translate to a higher performance since the consumers are the subprocesses that scrape the needed information from the individual web pages. Likewise, an overly high number of consumers will result in extended idle times as they wait for the producers to populate `PersonnelQueue`.

B. Memory Usage

The application of parallel programming techniques carries effects on the memory usage of the email scraper. Figure 6 shows the plots when the memory usage of the three parallel approaches is profiled every 0.1 seconds. It can be observed that memory consumption increases as the number of subprocesses increases. This is expected since each process is assigned its own virtual memory, which maps to a physical address space [16].

The first baseline approach has the least memory consumption, with a peak memory usage of around 9 GiB at 25 threads, although it also registers the lowest performance (Table II). The second baseline and our proposed approach have roughly the same memory consumption, with a peak memory usage of around 10 GiB at 25 threads; moreover, for these approaches, increasing the number of threads by five translates to a near-uniform rise of roughly a gibibyte of memory usage.

The difference in the memory consumption of the first baseline compared to the other two approaches may be ascribed to the peak number of elements in `PersonnelQueue`. The first baseline populates this queue at a slower rate since its cycle of retrieving personnel IDs, clicking the Load More button, and waiting for the new entries to load is strictly confined to only half of the user-specified scraping time. This limitation is not present in the second baseline and in our proposed approach.

On another note, Figure 7 shows the memory profile of our proposed approach under different producer-to-consumer ratios. The overall shapes and peaks of their plots are similar. However, it can be observed that the actual time at which the memory usage of the case with 18 producers and 7 consumers zeroes out is noticeably later than the others. This may be ascribed to the longer time needed by Python to finish its automatic garbage collection after the signal to terminate the

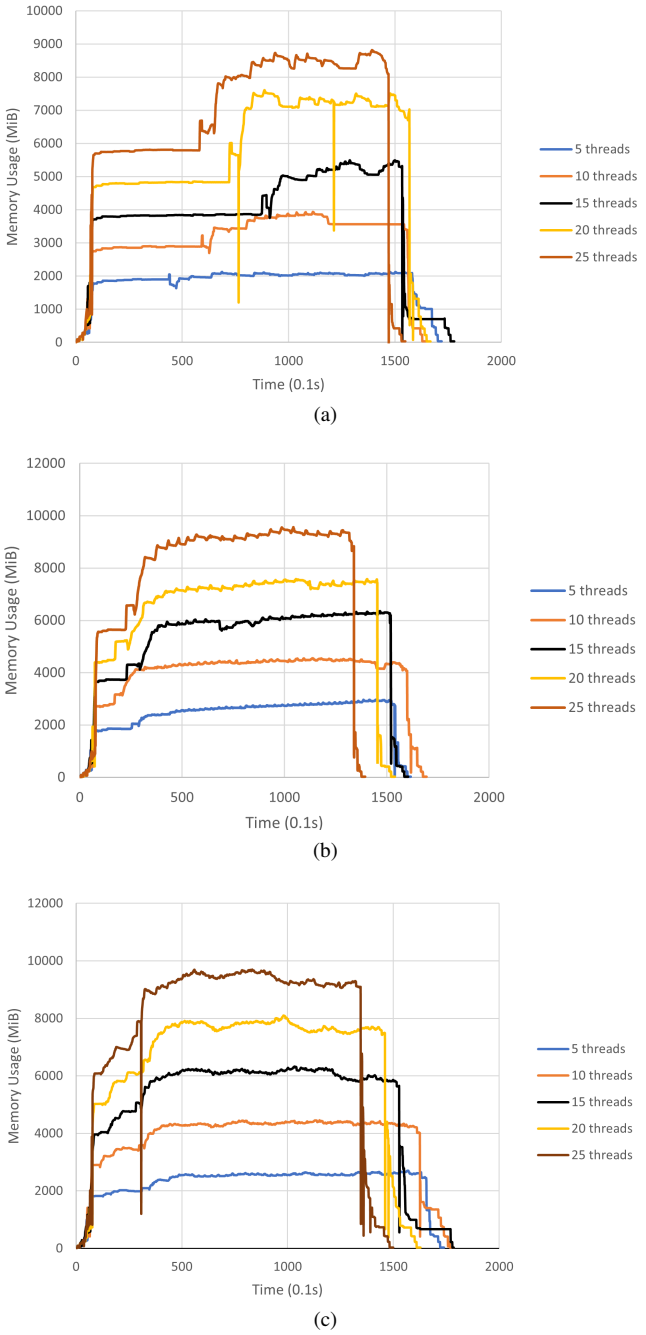

Fig. 6. Memory Usage of the Parallel Approaches. Figure 6a plots the total memory usage of the first baseline; Figure 6b, the second baseline; and Figure 6a, our proposed approach. Although the user-specified scraping time is fixed, the exact times at which the actual memory usage zeroes out differ due to Python's garbage collection, which involves flushing the remaining items in the synchronized queues. Meanwhile, the sudden drops in memory usage may be ascribed to network errors or episodes where the DLSU website fails to give a response on time. Our implementation of the program has mechanisms to automatically recover from these exceptions (*e.g.,* dynamically increasing the timeout window for explicit wait conditions).

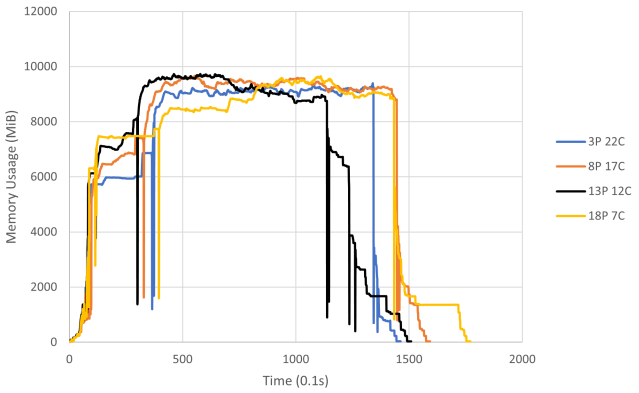

Fig. 7. Memory Usage of Our Proposed Approach Under Different Producer-to-Consumer Ratios. In the legend at the right side of the plot, *P* stands for producers, and *C* stands for consumers. Memory profiling was done via the `memory-profiler` library [17].

program has been received; the longer time may be caused by the need to flush more items from the synchronized queue, as there are around $2.5\times$ more producers than consumers.

IV. Conclusion

This paper presents the design, implementation, and analysis of a multiprocess email address scraper for the De La Salle University website staff directory. Our proposed approach models the scraping task as a multiple producer – multiple consumer problem. The set of personnel IDs in the staff directory is first divided by department, and multiple producers are mapped to different department directories. Each producer retrieves the personnel IDs from its assigned department directory and stores them in a synchronized queue. Concurrently, the IDs are dequeued by consumer subprocesses, which use them to visit the staff members' individual web pages, scrape their names, email addresses, and departments, and store these details in another queue. A dedicated subprocess gets the details from this queue and writes them on the output file.

The division of the scraping problem into multiple subproblems and assigning dedicated threads to accomplish each subproblem is evidential of functional decomposition. The division of the staff directory by department and having different producers work on these departments — as well as having various consumers perform a common activity but on different web pages — are applications of domain decomposition.

Experiments showed that running our proposed approach with five threads achieves a $7.22\times$ superlinear speedup compared to serial execution. Furthermore, it achieves better scalability and performance than baseline parallel programming approaches that scrape from the root directory, as having multiple producers minimizes the idle time of the consumers and having multiple consumers expedites scraping the needed information from the web pages.

Future directions include implementing a dynamic approach to adjust the producer-to-consumer ratio in our proposed approach depending on the load. The program may also be extended to function on distributed systems.

References

[1] S. Lunn, J. Zhu, and M. Ross, "Utilizing web scraping and natural language processing to better inform pedagogical practice," in *2020 IEEE Frontiers in Education Conference (FIE)*, 2020, pp. 1–9.

[2] Petrović and I. Stanišević, "Web scrapping and storing data in a database, a case study of the used cars market," in *2017 25th Telecommunication Forum (TELFOR)*, 2017, pp. 1–4.

[3] W. S. Alkaberi, R. H. Aljuhani, and H. M. Alamoudi, "Web scraper application for extracting scientific journals data," in *The 5th International Conference on Future Networks amp; Distributed Systems*, ser. ICFNDS 2021. New York, NY, USA: Association for Computing Machinery, 2022, p. 220–224.

[4] D. Pratiba, A. M.S., A. Dua, G. K. Shanbhag, N. Bhandari, and U. Singh, "Web scraping and data acquisition using google scholar," in *2018 3rd International Conference on Computational Systems and Information Technology for Sustainable Solutions (CSITSS)*, 2018, pp. 277–281.

[5] Fatmasari, Y. N. Kunang, and S. D. Purnamasari, "Web scraping techniques to collect weather data in south sumatera," in *2018 International Conference on Electrical Engineering and Computer Science (ICECOS)*, 2018, pp. 385–390.

[6] G. Karatas, "Email scraping: overview, use cases, challenges best practices," September 2022. [Online]. Available: https://research.aimultiple.com/email-scraping/

[7] R. Diouf, E. N. Sarr, O. Sall, B. Birregah, M. Bousso, and S. N. Mbaye, "Web scraping: State-of-the-art and areas of application," in *2019 IEEE International Conference on Big Data (Big Data)*, 2019, pp. 6040–6042.

[8] Python Software Foundation, "multiprocessing — process-based parallelism." [Online]. Available: https://docs.python.org/3/library/multiprocessing.html

[9] "Selenium." [Online]. Available: https://github.com/SeleniumHQ/Selenium

[10] Google, "Chrome webdriver." [Online]. Available: https://chromedriver.chromium.org/

[11] S. Pirogov, "Webdriver Manager for Python." [Online]. Available: https://github.com/SergeyPirogov/webdriver_manager

[12] Software Freedom Conservancy, "Waits." [Online]. Available: https://www.selenium.dev/documentation/webdriver/waits/#explicit-wait

[13] Software Freedom Conservancy, "Frequently asked questions for Selenium 2." [Online]. Available: https://www.selenium.dev/documentation/legacy/selenium_2/faq/

[14] P. S. Pacheco and M. Malensek, "Chapter 2 - parallel hardware and parallel software," in *An Introduction to Parallel Programming (Second Edition)*, second edition ed., P. S. Pacheco and M. Malensek, Eds. Philadelphia: Morgan Kaufmann, 2022, pp. 17–88.

[15] Python Software Foundation, "queue — A synchronized queue class ." [Online]. Available: https://docs.python.org/3/library/queue.html

[16] A. Silberschatz, H. F. Korth, and S. Sudarshan, *Database System Concepts, Seventh Edition*. McGraw-Hill Book Company, 2020.

[17] F. Pedregosa and K. Altendorf, "Memory profiler." [Online]. Available: https://github.com/pythonprofilers/memory_profiler