DSC520: High-Performance Scientific Computing
Final Project Report

Instructor: Dr. Alfa Heryuduno

Submitted by:
Abhijot Bedi
Master's in Data Science
ID 01908371

UMass | Dartmouth

Project Title: Parallelizing Web Scraping to Extract Books Data from Amazon(dot)com

Abhijot Bedi
Master's in Data Science
University of Massachusetts Dartmouth

Submitted 15 December 2021

Abstract: Parallel processing is one of the important subjects today in scientific computing. Researchers and Data Scientists work with a tremendous amount of data. Using parallel processing systems or methods can save a significant amount of time while receiving the same results. In the last decade, data has become a treasure in the industry. There can be valuable insights generated by data and those help companies in various applications. The more data, the better.

But with more data comes more computational time consumption, running the programs for such applications. This is when parallelization becomes valuable. Many industry leaders have access to supercomputers with a number of cores far superior to what other mid-level and startup companies have.

With this experimental project, we will observe how even a small number of cores easily available on commercial computer systems can help save a lot of time performing web scraping.

Keywords: Parallelization, Speedup, Average Efficiency, Cores

UMass | Dartmouth

Contents

1 Introduction
2 Project Methodology
3 Project Analysis and Results
4 Numerical Results from Parallelization
5 Conclusion
6 References
7 Appendix

---

1. Introduction

Parallel Processing can be used to speed up the execution of programs by dividing the program into multiple fragments and executing them simultaneously [1]. This project aims to parallelize a web scraper programmed to extract book listing from the popular e-commerce website, Amazon(dot)com, performing an automated genre search and webpage navigation designed in Python programming language. We will be performing this extraction process in both sequential and parallel settings utilizing different numbers of cores.

In today's date, continuous advances in parallel programming allow us to use parallel computing in different areas of knowledge [2]. The objective of this project is to reduce the computational time it takes to scrape data from Amazon(dot)com. We have used two versions of a program creating different experimental settings - Serial and Parallel.

The motivation behind this project came from the idea of how 16% of the products Amazon sells are books which goes up to 52 million in numbers [4]. Now that's a lot of data. We will be looking at how we can parallelize a simple web scraping problem to collect book data with respect to their genre and store it in datasets.

In the next section, we discuss the methodology and work-around of the project followed by the analysis of the two program versions we mentioned above. We will also be comparing the results between different numbers of core utilization and visualization.

2. Methodology

The code is programmed on an Apple Macbook Air M1 series, consisting of 8 CPU cores.

Although the program is scraping the book data (prices, ratings, and webpage links) from one website, it deals with multiple URLs when navigating through the website fetching items.

In order to perform scraping, we have used two libraries in the Python programming language:

Selenium: for automating webpage navigation using chromedriver.
BeautifulSoup: for extraction of the book items using html.parser and find_all.

We have used three more libraries to support this extraction process and store the results in.csv format in order to create a dataset for each book genre searced:

Pandas: for using the Dataframe method to convert it into a .csv file.
Concurrent.futures: for parallelizing the program using ProcessPoolExecutor.
Time: to record the computational time of the program.

The first step was to set up the webdriver using Selenium. The default browser on this system is Chrome, so we have used a chromedriver compatible with the chrome browser version. Coupled with BeautifulSoup, we have performed a keyword search for every book genre name defined in a list, manipulating a base URL based on Amazon(dot)com standards.

In most of the book search results on Amazon, it was observed that the minimum page numbers are 7. Therefore, we have performed an iteration in the range of 1 to 8, to make sure the program extracts information not just from the first page of the search results but all of them. We created a soup object and parsed the html page source, followed by the extraction of the items based on the appropriate tags which were identified using inspect element feature on the chrome browser.

After the extraction process, we append the results in a list which was later stored in a data frame and exported to a .csv format file to create individual datasets with respect to the keywords.

Parallelization:

For this experiment, three parallelization cases were considered manipulating the program to distribute the keywords list item equally in the maximum workers defined. We have performed this project for 2, 4, and 8 cores using the ProcessPoolExecutor method and map distributed keywords from the list to the main to follow the aforementioned program flow.

3. Program Analysis and Results

The project is divided into four cases in the *sequential version* of the program. We take:

I. 8 genre names in the list to perform the books search on Amazon.
II. 16 genre names.
III. 24 genre names.
IV. 32 genre names.

Whereas, in the *parallelized version* of the program we take three cases for each case mentioned in the *sequential version.* For example:

8 genre names are,

1. distributed between 2 workers/cores.
2. distributed between 4 workers/cores.
3. distributed between 8 workers/cores.

The result of running the program for a list of 32 genre names ***serially*** can be referred from figure 1.

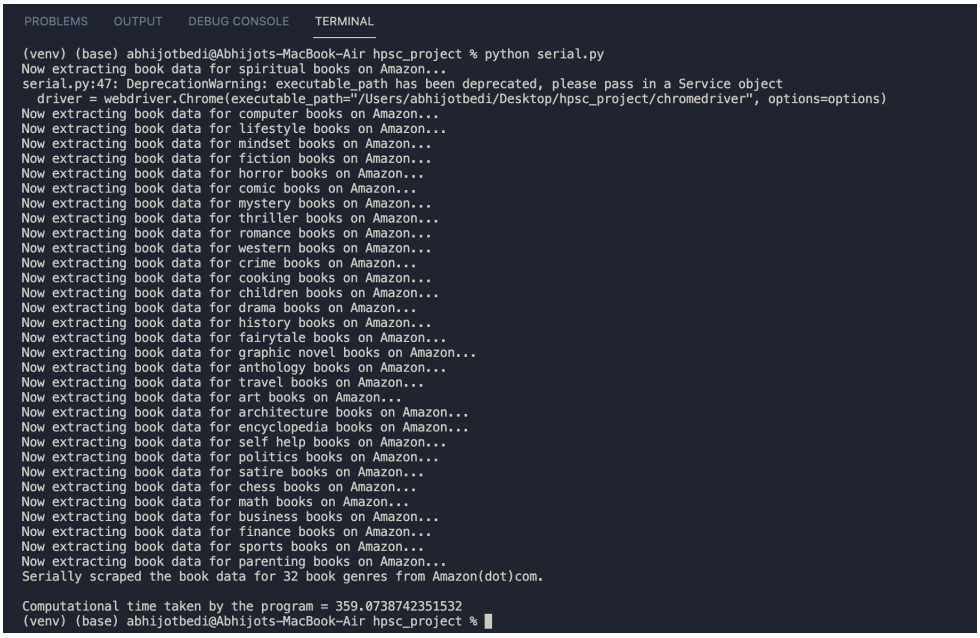

Figure 1. Terminal snapshot after running the program for 32 genres serially

It can be observed that the computational time recorded after running the program serially is 359.0738742351532 seconds, whereas we are drastically saving time and achieving the same result performing parallelization (fig. 2). Three runs are in the order of 2, 4, and 8 workers.

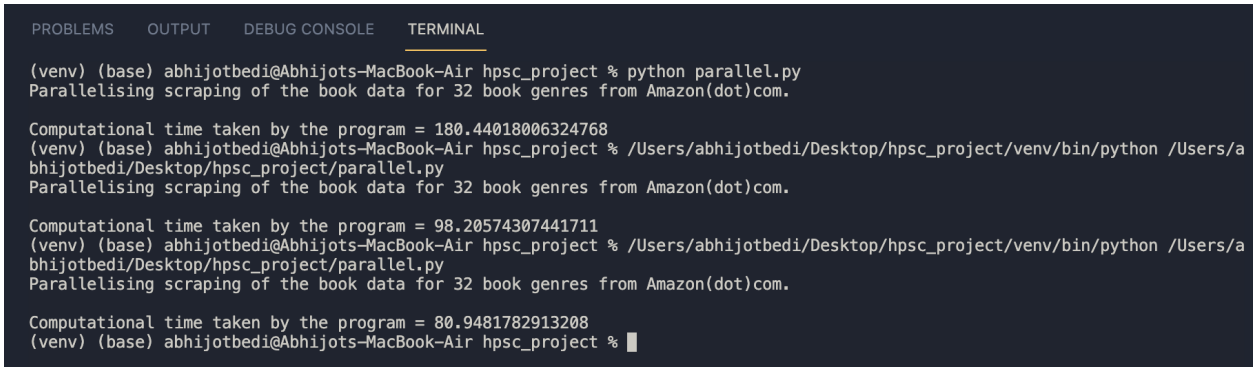

Figure 2. Terminal snapshot after running the program in multi-processing using 2, 4, and 8 cores

4. Numerical Results

In order to determine how well and efficiently parallelizing the program is helping reduce the computational time, we will be using two formulas:

$$S_p = \frac{t_1}{t_p} \qquad E_p = \frac{S_p}{p}$$

where,

t1 is the time taken by the program to run sequentially.
Tp is the time taken by the program to run after parallelization.
Sp is the Speedup.
p is the number of workers.
Ep is the Average Efficiency.

Results achieved by this project have been calculated using these formulas.

	Serial	Parallel (2 cores)	Parallel (4 cores)	Parallel (8 cores)
8	80.53906894	50.95419168	29.865098	20.11667204
16	179.7604578	96.01601028	53.96597195	43.53361678
24	262.354116	129.6935463	68.28163886	51.07261205
32	359.0738742	180.4401801	98.20574307	80.94817829

Figure 3. The computational time recorded from serial and parallel programs

We can observe the time taken by the programs (serial and parallel) in figure 3.

	Parallel (2 cores)	Parallel (4 cores)	Parallel (8 cores)
8	1.580617144	2.696762252	4.003598049
16	1.872192536	3.33099639	4.129233247
24	2.022877186	3.842235194	5.136884632
32	1.98998845	3.656342929	4.435848735

Figure 4. Speedup calculated for each parallelization case (2, 4, and 8 cores)

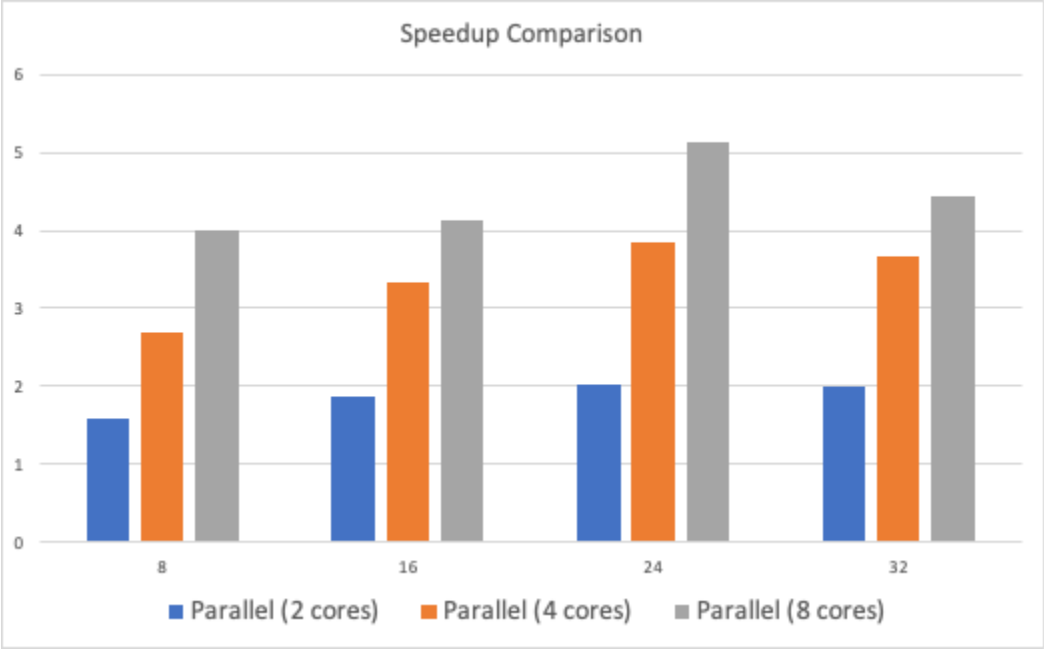

Figure 5. Speedup comparison between 2, 4, and 8 cores categorized in four different cases of the number of genres (8, 16, 24, and 32)

	Parallel (2 cores)	Parallel (4 cores)	Parallel (8 cores)
8	79.0308572	67.4190563	50.04497562
16	93.60962678	83.27490976	51.61541558
24	101.1438593	96.05587986	64.21105791
32	99.4994225	91.40857321	55.44810918

Figure 6. Average Efficiency for each case

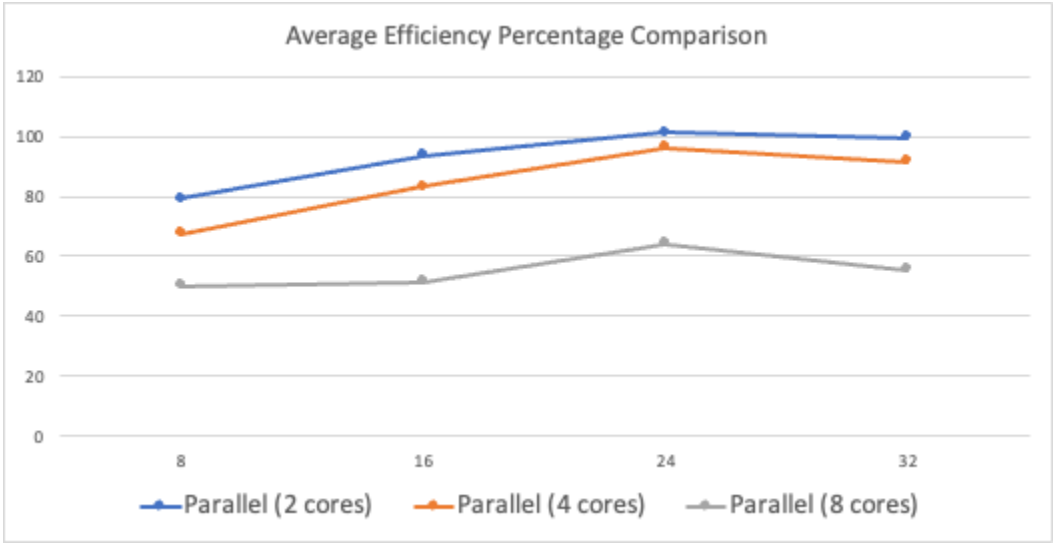

Figure 7. Visualization of the Average Efficiency compared between 2, 4, and 8 cores

5. Conclusion

From the project results, we can conclude that using parallel computing saves computational time. Parallelizing a program can be a time-saver. Researchers and many companies spend a lot of time gathering information. In this experimental setting, we were limited to 8 cores but we can acknowledge that increasing the number of cores on the system can provide even more improvements in saving time.

A significant change in results can be observed when we increased our number of processors working on the program. Web scraping has a lot of industrial applications in e-commerce, marketing, competition research, and even in creating database pipelines. These applications work with a huge amount of data and parallelization can be an effective approach to implement as we have learned from this project.

6. References

[1] RupaYashwantaNagpure and Sandhya Dahake, “Research Paper on Basic Parallel Processing,” IOSR Journal of Engineering, 2278-8719, PP 77-83.

[2] T. de Jesus Oliveira Duraes, P. Sergio Lopes de Souza, G. Martins, D. Jose Conte, N. Garcia Bachiega and S. Mazzini Bruschi, "Research on Parallel Computing Teaching: state of the art and future directions," *2020 IEEE Frontiers in Education Conference (FIE)*, 2020, pp. 1-9, doi: 10.1109/FIE44824.2020.9273914.

[3] Article: https://medium.datadriveninvestor.com/speed-up-web-scraping-using-multiprocessing-in-python-af434ff310c5

[4] Article: https://www.scrapehero.com/how-many-books-does-amazon-sell/

7. Appendix

Serial Program Code

```
from bs4 import BeautifulSoup
from selenium import webdriver
import time
import pandas as pd
import time

# search term based url extraction
def fetch_link(product_keyword):
   base = 'https://www.amazon.com/s?k={}'
   product_keyword = product_keyword.replace(' ', '+')

   link = base.format(product_keyword)
   link += '&page={}'

   return link

# extraction fuction
def prod_extraction(product):

   tag = product.h2.a
   product_name = tag.text.strip()

   product_link = 'https://www.amazon.com' + tag.get('href')

   try:
       price = product.find('span', 'a-offscreen').text
   except AttributeError:
       return
```

```
    try:
        rating = product.i.text
    except AttributeError:
        rating = ''

    result = (product_name, price, rating, product_link)
    return result

# main function
def main_func(product_keyword):
    # setup driver
    options = webdriver.ChromeOptions()
    options.add_argument("--ignore-certificate-errors")
    options.add_argument("--incognito")
    options.add_argument("--headless")
    options.add_argument("--no-sandbox")
    options.add_argument("--disable-dev-shm-usage")
    driver =
webdriver.Chrome(executable_path="/Users/abhijotbedi/Desktop/hpsc_project/chromedriver
", options=options)

    product_listing = []
    product_listings = []
    # results = 0

    # fetching link for a specific product search
    url = fetch_link(product_keyword)

    for page_num in range(1, 8):
        # driver is getting the link
        driver.get(url.format(page_num))

        # begin extraction for every product
        soup = BeautifulSoup(driver.page_source, 'html.parser')
        results = soup.find_all('div', {'data-component-type': 's-search-result'})
        # print(len(results))
        # time.sleep(5)
        for product in results:
            product_listing = prod_extraction(product)


            product_listings.append(product_listing)
```

```
    # print(len(product_listings))
    # after extraction closing the driver
    driver.close()

    # storing the books info in a csv
    df = pd.DataFrame(product_listings)
    df.to_csv(product_keyword + '.csv', header= ['Book Name', 'Price', 'Rating',
'Link'], index=False)

# program starts
if __name__ == '__main__':
    start_time = time.time()

    # giving a list of book genres to extract books on
    keyword_list = ['spiritual books', 'computer books', 'lifestyle books', 'mindset
books',
            'fiction books', 'horror books', 'comic books', 'mystery books',
            'thriller books', 'romance books', 'western books', 'crime books',
            'cooking books', 'children books', 'drama books', 'history books',
            'fairytale books', 'graphic novel books', 'anthology books', 'travel
books',
            'art books', 'architecture books', 'encyclopedia books', 'self help books',
            'politics books', 'satire books', 'chess books', 'math books',
            'business books', 'finance books', 'sports books', 'parenting books'
        ]

    # iterating in the list
    for keys in keyword_list:
        print("Now extracting book data for %s on Amazon..." % keys)

        # calling the main function
        main_func(keys)

    # Recorded computational time
    print("Serially scraped the book data for %d book genres from Amazon(dot)com." %
len(keyword_list))
    print("\nComputational time taken by the program = %s" % (time.time() -
start_time))
```

UMass | Dartmouth

Parallel Program Code:

```
from bs4 import BeautifulSoup
from selenium import webdriver
# import csv
import time
import pandas as pd
import concurrent.futures

# search term based url extraction
def fetch_link(product_keyword):
    base = 'https://www.amazon.com/s?k={}'
    product_keyword = product_keyword.replace(' ', '+')

    link = base.format(product_keyword)
    link += '&page={}'

    return link

# extraction fuction
def prod_extraction(product):

    tag = product.h2.a
    product_name = tag.text.strip()

    product_link = 'https://www.amazon.com' + tag.get('href')

    try:
        price = product.find('span', 'a-offscreen').text
    except AttributeError:
        return

    try:
        rating = product.i.text
    except AttributeError:
        rating = ''

    result = (product_name, price, rating, product_link)
    return result

# main function
def main_func(product_keyword):
```

Submitted by: Abhijot Bedi

```
    # setup driver
    options = webdriver.ChromeOptions()
    options.add_argument("--ignore-certificate-errors")
    options.add_argument("--incognito")
    options.add_argument("--headless")
    options.add_argument("--no-sandbox")
    options.add_argument("--disable-dev-shm-usage")
    driver =
webdriver.Chrome(executable_path="/Users/abhijotbedi/Desktop/hpsc_project/chromedriver
", options=options)

    product_listing = []
    product_listings = []
    # results = 0

    # fetching link for a specific product search
    url = fetch_link(product_keyword)

    for page_num in range(1, 8):
        # driver is getting the link
        driver.get(url.format(page_num))

        # begin extraction for every product
        soup = BeautifulSoup(driver.page_source, 'html.parser')
        results = soup.find_all('div', {'data-component-type': 's-search-result'})
        # print(len(results))
        # time.sleep(5)
        for product in results:
            product_listing = prod_extraction(product)


            product_listings.append(product_listing)

    # print(len(product_listings))
    # after extraction closing the driver
    driver.close()


    df = pd.DataFrame(product_listings)
    df.to_csv(product_keyword + '.csv', header= ['Book Name', 'Price', 'Rating',
'Link'], index=False)
```

```
    # with open(product_keyword + '.csv', 'w', newline='', encoding='utf-8') as f:
    #     writer = csv.writer(f)
    #     writer.writerow(['Book Name', 'Price', 'Rating', 'Link'])
    #     writer.writerows(product_listings)

if __name__ == '__main__':
    start_time = time.time()
    keyword_list = ['spiritual books', 'computer books', 'lifestyle books', 'mindset
books',
            'fiction books', 'horror books', 'comic books', 'mystery books'
        ]

    # parallelising the extraction using ProcessPoolExtractor using concurrent.futures
    with concurrent.futures.ProcessPoolExecutor(max_workers=8) as executor:
            result = executor.map(
                main_func,
                keyword_list,
            )

    # Recorded computational time
    print("Parallelising scraping of the book data for %d book genres from
Amazon(dot)com." % len(keyword_list))
    print("\nComputational time taken by the program = %s" % (time.time() -
start_time))
```