Fatima Jinnah women University

Parallel And Distributed Computing

Project Report

Project Title:

Distributed Web Crawler Using Dask

Submitted by:

- Ommamah Latif (032)
- Rafia Islam (034)
- Sana Banaras (037)
- Sania Arif (038)
- Warda Yousaf(043)

Submitted to:

Mam Madiha Wahid

Due Date:

8-Dec-2025

Contents

Table of Figures:

Distributed Web Crawler Using Dask: A Parallel Computing Approach

Abstract

Web crawling plays a fundamental role in data collection, indexing, and search engine operations. Traditional crawlers execute sequentially, fetching one link at a time, which becomes inefficient when processing large datasets or multiple websites. This project addresses this challenge by designing a **Distributed Web Crawler** using **Dask**, a parallel computing library for Python.

The crawler uses **Requests** to retrieve HTML content, **BeautifulSoup** for parsing and extracting text, and **Dask Bag** for parallel distributed execution. Additional modules like **Pandas**, **Matplotlib**, and **WordCloud** support data processing and visualization. The project demonstrates how parallel computation significantly reduces web crawling time and improves scalability. Experimental results show that Dask-based crawling outperforms sequential crawling by efficiently utilizing available CPU cores.

Introduction

What is Web Crawling?

Web crawling is the automated process of systematically browsing the web, downloading content, and extracting data. Web crawlers serve as the backbone of search engines, academic research tools, and data-driven applications.

Key operations include:

- Sending HTTP requests
- Receiving HTML responses
- Parsing and extracting content
- Storing or analyzing data

Limitations of Sequential Crawling

Traditional crawlers are sequential:

URL 1 → URL 2 → URL 3 → ...

This becomes slow due to:

- Network latency
- CPU idle time while waiting for responses
- Inability to scale with increased workloads

Parallel Computing in Web Crawling

Parallel computing enables:

- Running multiple crawl operations at the same time
- Efficient utilization of CPU cores
- Reduced total runtime

A distributed crawler assigns *multiple URLs to multiple workers*, completing the job significantly faster.

Why Use Dask for Distributed Crawling?

Dask is chosen because:

- It performs parallel computing using all system cores
- It processes collections using **Dask Bag**, ideal for unstructured tasks
- It scales to clusters (but also works on a laptop)
- It integrates with common Python libraries

Thus, Dask offers an excellent approach to building a scalable distributed web crawler.

Problem Statement

Sequential crawling suffers from the following issues:

1. **Slow Performance**
   Fetches pages one by one, wasting time.
2. **Lack of Scalability**
   Does not utilize CPU parallelism.

3. **Not suitable for large datasets**
   Crawling hundreds of pages becomes impractical.
4. **No analytical capability**
   Raw HTML needs parsing and word extraction.

Goal of This Project

To design and implement a **distributed web crawler** that:

- Fetches multiple URLs in parallel
- Extracts readable text
- Performs word-frequency analysis
- Visualizes results
- Demonstrates efficiency gains of parallel computing

Tools & Technologies

This project uses a combination of Python libraries and parallel computing tools to build a scalable distributed web crawler. Each tool plays a specific role from crawling to parsing, processing, and visualization

Python

Python is the primary programming language used in this project because it provides:

- Simple syntax for quick development
- Powerful libraries for networking, parsing, data analysis, and visualization
- Direct integration with Dask for parallel computing

Python supports all modules used in the crawler such as requests, BeautifulSoup, pandas, and matplotlib.

Jupyter Notebook

Jupyter Notebook is used as the development and testing environment. It allows:

- Step-by-step execution of code

- Easy debugging
- Instant visualization of results
- Interactive development

It is ideal for data-driven projects like web crawling and analysis

Dask

Dask is the **core technology** of this project.

Why Dask?

- Enables parallel execution using all CPU cores
- Removes Python's GIL limitation using distributed scheduling
- Works on a single machine or a full cluster
- Integrates with other Python libraries easily

Dask Bag

This project uses **Dask Bag**, which is ideal for:

- Unstructured, independent tasks
- Processing a list of URLs in parallel
- Applying map, filter, reduce easily
- Handling large datasets efficiently

This allows the crawler to fetch multiple URLs **simultaneously**, reducing total crawling time.

Requests Library

Requests is used for:

- Sending HTTP GET requests
- Downloading HTML content from websites

- Handling network errors and timeouts

It forms the **first stage** of crawling: retrieving raw HTML.

BeautifulSoup (bs4)

Why We Use It

BeautifulSoup is used to:

- Parse messy HTML
- Remove tags, scripts, and CSS
- Extract clean readable text
- Convert HTML into plain words (tokens)

It forms the **second stage** of crawling: parsing and text extraction.

Pandas

Pandas is used for:

- Creating DataFrames
- Cleaning and organizing text data
- Sorting words by frequency
- Preparing data for visualization

It is essential for word-frequency analysis.

Matplotlib

Matplotlib is used to create:

- Bar charts
- Frequency comparison plots

It helps visually compare results between sequential and parallel crawling.

WordCloud

WordCloud generates:

- Visual representations of most repeated words
- Attractive visual summaries of crawled content

This gives users an instant overview of extracted text.

System Architecture

The system is designed as a pipeline, where each stage processes the output of the previous stage.

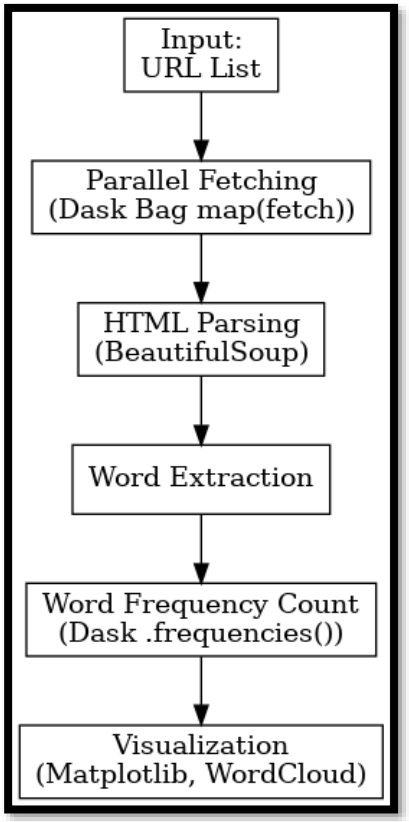


Figure 1

Dask parallelizes the entire process.

Architecture Explanation

Input: URL List

The system begins with a predefined list of URLs that need to be crawled.A list of URLs is provided as input. Each URL represents one webpage to be fetched and processed.

Parallel Fetching (Dask Bag map(fetch))

In this stage, Dask Bag distributes the URLs across multiple workers (CPU cores).

- Each worker runs the fetch() function
- HTML content from many URLs is downloaded simultaneously
- This parallel approach greatly reduces crawling time compared to sequential processing

This is the core performance advantage of the project.

HTML Parsing (BeautifulSoup)

Once the HTML is fetched:

- BeautifulSoup parses the HTML structure
- It removes unnecessary elements such as JavaScript, CSS, and HTML tags
- It extracts clean readable text from each webpage

This step converts raw HTML into meaningful text data.

Word Extraction

The cleaned text is:

- Converted to lowercase
- Split into individual words (tokens)
- Filtered to remove empty strings or symbols

This step prepares the data for frequency calculations.

Frequency Count (Dask .frequencies())

Dask performs a distributed frequency calculation on all extracted words.

- Counts how many times each word appears
- Merges results from different workers
- Produces a final frequency dictionary

This helps identify the most common words across all webpages.

Visualization (Matplotlib, WordCloud)

Visual output is generated in two forms:

- Bar charts showing the most frequent words
- Word clouds providing visual summaries
- These graphs help validate and present results effectively.

Key Strengths of the Architecture

- Fully parallel pipeline using Dask
- Scalable (can run on multi-node cluster)
- Fault-tolerant (errors in one URL don't break pipeline)
- Modular (each stage is independent)
- Optimized for CPU utilization

Methodology

Installing Libraries

Methodology refers to the systematic process, techniques, and steps used to design, implement, and evaluate a research project or system. The complete step-by-step workflow of how the project was developed.

The following libraries were installed in Jupyter Notebook

- dask → parallel computations
- requests → HTTP requests
- beautifulsoup4 → HTML parsing
- pandas → data manipulation
- matplotlib → plotting
- wordcloud → visualization

```
pip install dask requests beautifulsoup4 pandas matplotlib wordcloud
```

Figure 2

URL Dataset Preparation

A predefined list of URLs was created:

```
urls = [
    'https://byjus.com/cbse/essay-on-computer/',
    'https://www.example.org',
    'https://www.ibm.com/think/topics/parallel-computing'
]
```

Figure 3

Purpose:

- Input dataset for crawling
- Each URL becomes one parallel task

Fetch Function Development

The **fetch()** function retrieves raw HTML content from each URL:

```
def fetch_page(url):
    try:
        resp = requests.get(url, timeout=REQUEST_TIMEOUT, headers=REQUEST_
        if resp.status_code == 200:
            return resp.text
        else:
            logging.warning(f"Non-200 status for {url}: {resp.status_code}
            return ""
    except requests.RequestException as e:
        logging.warning(f"Request failed for {url}: {e}")
        return ""
```

Figure 4

Purpose:

- Sends HTTP GET request
- Handles network delays using timeout

- Ensures pipeline continuity by returning empty text on failure

HTML Parsing & Text Extraction

The **parse()** function converts HTML into clean text:

```
def parse_words(html):
    if not html:
        return []
    soup = BeautifulSoup(html, "html.parser")
    for s in soup(["script", "style", "noscript"]):
        s.decompose()
    text = soup.get_text(separator=" ")
    words = [w.lower() for w in text.split() if w.isalpha()]
    return words
```

Figure 5

Purpose:

- Removes HTML tags, scripts, CSS
- Extracts readable plain text
- Converts text into a list of lowercase words (tokens)

This prepares the data for frequency computation.

Sequential Crawling

A sequential crawler was developed to measure performance before applying parallelism:

```
start_seq = time.time()
all_words_seq = []
for url in urls:
    html = fetch_page(url)
    words = parse_words(html)
    all_words_seq.extend(words)
```

Figure 6

Why needed?

- Establishes baseline performance
- Demonstrates inefficiency of one-by-one crawling

Distributed Crawling Using Dask Bag

Dask Bag was used to execute crawling tasks in parallel:

```
bag = db.from_sequence(urls, npartitions=4)   # increase partit
html_pages = bag.map(fetch_page)
words_bag = html_pages.map(parse_words).flatten()
words_bag = words_bag.filter(lambda w: w not in STOP_WORDS)
word_counts_dask = dict(words_bag.frequencies().compute())
```

Figure 7

- **from_sequence(urls)** → Splits URL list into distributed tasks
- **map(fetch)** → Multiple workers download HTML simultaneously
- **map(parse)** → Parsing also happens in parallel
- **flatten()** → Converts list of lists into a flat list of words
- **frequencies().compute()** → Dask performs distributed word counting

This step utilizes all CPU cores, reducing crawl time significantly.

Data Storage & Preparation

The outputs of both sequential and Dask-based crawlers were stored in:

- Python dictionaries
- Pandas DataFrames

This enabled simple sorting, filtering, and preparing data for visualization.

Visualization

Two types of visualizations were generated:

Bar Charts (Matplotlib)

Shows top N most frequent words for quick comparison.

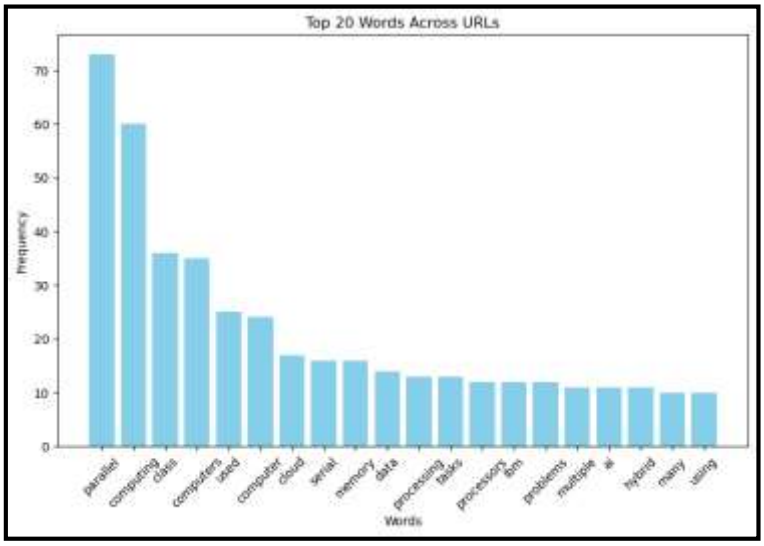


Figure 8

WordCloud

Visual representation of highest-frequency words extracted from all pages.

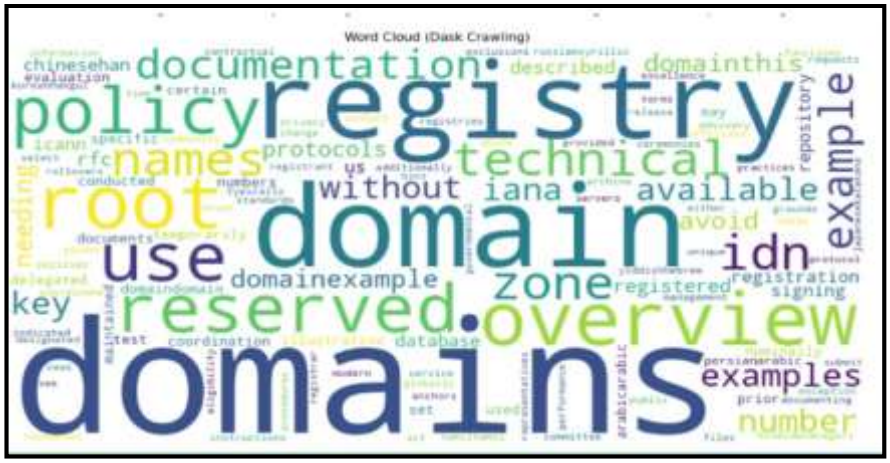


Figure 9

These visualizations validate the correctness of the crawler and support result interpretation.

Performance Evaluation

Two types of measurements were taken:

Execution Time

- Sequential crawling time
- Dask crawling time
- Speedup achieved (Parallel / Sequential)

Frequency Comparison

- Top words extracted in each mode
- Consistency of parsing results

Complexity Analysis

- **N** = number of URLs
- **C** = number of CPU cores

Sequential Time Complexity

$T = O(N \times fetch\_time \times parse\_time)$

Parallel Time Complexity

$T = O((N / C) \times fetch\_time)$

This demonstrates theoretical speedup under parallel execution.

Fetch the Wikipedia homepage and print the HTTP status code.

Error	Meaning
403	site blocked your request
429	too many requests
requests.exceptions.ConnectionError	internet / DNS issue

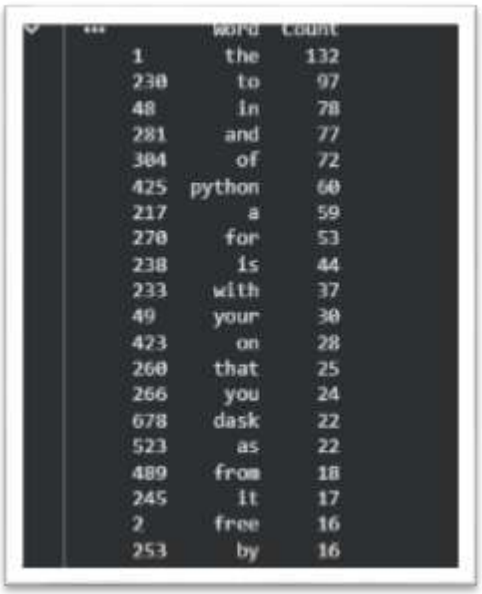
What this header does?

- **User-Agent** → makes your crawler look like a normal browser

- **Accept-Language** → preferred language (optional)
- **Accept** → types of content your client can understand

Workflow Summary

1. Fetch pages (Dask parallel).
2. Extract links → normalize → add to URL queue.
3. Parse content → count words or extract data.
4. Store HTML/content in a database.
5. Repeat for new URLs (multi-level crawling).



	word	count
1	the	132
230	to	97
48	in	78
281	and	77
304	of	72
425	python	60
217	a	59
270	for	53
238	is	44
233	with	37
49	your	30
423	on	28
260	that	25
266	you	24
678	dask	22
523	as	22
489	from	18
245	it	17
2	free	16
253	by	16

Figure 10

```
import pandas as pd

# Convert to Dataframe
df = pd.DataFrame(list(filtered_word_counts.items()), columns=['Word', 'Count'])

# Sort by count descending
df = df.sort_values('Count', ascending=False).head(20)  # Top 20 words
print(df)
```

```
...          Word  Count
375     python     60
614       dask     22
1         free     16
0    wikipedia     14
337       news     14
399       code     13
943        hrs     12
468       data     12
366        use     12
867        bbc     11
430     events     11
358  community     11
376        psf     10
294        get     10
503       easy     10
437         us     10
4     articles      9
666     import      8
37      simple      8
349   software      8
```

Figure 11

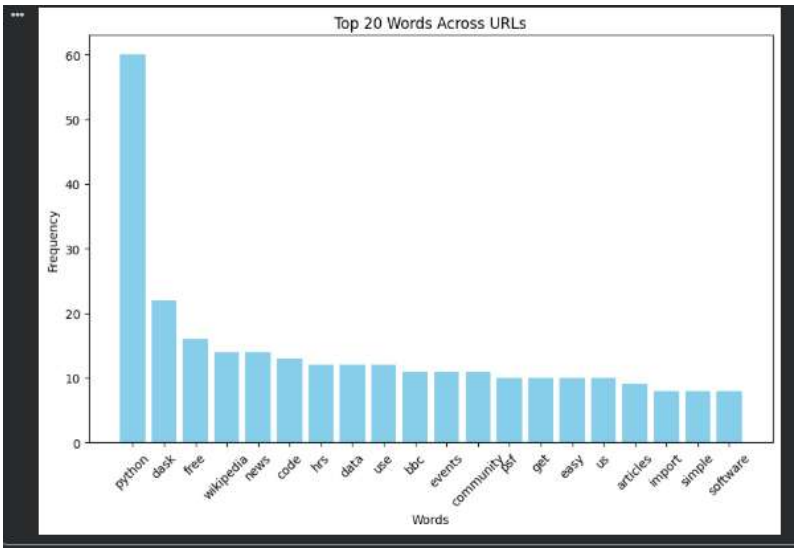


Figure 12

fetch_page(url)

- Sends an HTTP GET request to the given URL.
- Adds a User-Agent header to mimic a browser.
- Uses timeout=5 to avoid hanging requests.

- Returns the page HTML if status code is 200, otherwise returns an empty string.

parse_words(html)

- Uses BeautifulSoup to parse HTML.
- Extracts all visible text from the page.
- Splits text into words.
- Keeps alphabetic words only and converts them to lowercase.
- Returns a list of words.

Dask Bag Processing

- db.from_sequence(urls, npartitions=10) → creates a Dask bag from the list of URLs, split into 10 partitions for parallel processing.
- bag.map(fetch_page) → fetches all pages in parallel.
- html_pages.map(parse_words).flatten() → parses words from each page and flattens them into a single sequence.
- words_bag.frequencies().compute() → counts the frequency of each word across all pages and computes the result.
- dict(word_counts) → converts the result to a standard Python dictionary.

Stopword Removal

- Uses NLTK stopwords: stop_words = set(stopwords.words('english')).
- {word: count for word, count in word_counts_dict.items() if word not in stop_words} → filters out common English words like "the", "and", etc.

Pandas DataFrame

- pd.DataFrame(list(filtered_word_counts.items()), columns=['Word', 'Count']) → converts the dictionary to a DataFrame with columns Word and Count.

- .sort_values('Count', ascending=False).head(20) → sorts by frequency descending and selects top 20 words.

Plotting with Matplotlib

- plt.bar(df['Word'], df['Count'], color='skyblue'): creates a bar chart of the top words.
- plt.xticks(rotation=45): rotates x-axis labels for readability.
- plt.show(): displays the plot.

Save Results

- df.to_csv('word_frequency_results.csv', index=False) : saves the top words and their counts to a CSV file.

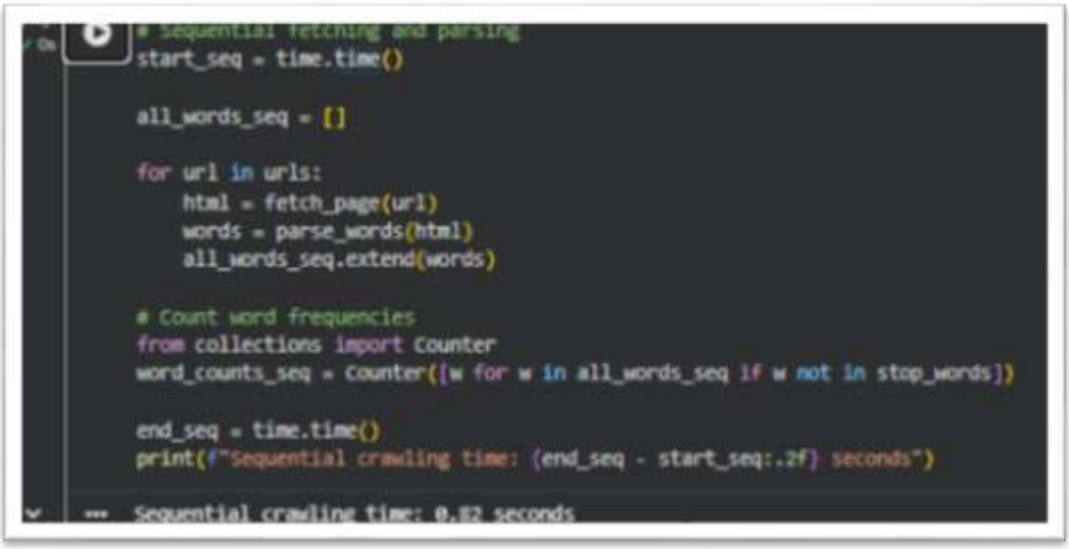


Figure 13

```
import dask.bag as db

# Start timer
import time
start_dask = time.time()

# Create Dask Bag
bag = db.from_sequence(urls)
html_pages = bag.map(fetch_page)
words_bag = html_pages.map(parse_words).flatten()

# Count word frequencies
word_counts_dask = dict(words_bag.frequencies().compute())

# Remove stopwords (if used)
word_counts_dask = {w: c for w, c in word_counts_dask.items() if w not in stop_words}

end_dask = time.time()
print(f"Dask parallel crawling time: {end_dask - start_dask:.2f} seconds")

... Dask parallel crawling time: 0.61 seconds
```

Figure 14

Feature	Sequential Crawling	Dask Parallel Crawling
Execution Mode	One URL at a time	Multiple URLs in parallel
Speed	Slower	Faster
Resource Utilization	Low	High (CPU + network)
Scalability	Limited	High (multi-core / multi-node)
Fault Tolerance	Simple	Automatic retries, distributed safe
Complexity	Simple	More setup required (Dask client, workers)
Best Use	Small-scale or testing	Large-scale crawling / scraping

Figure 15

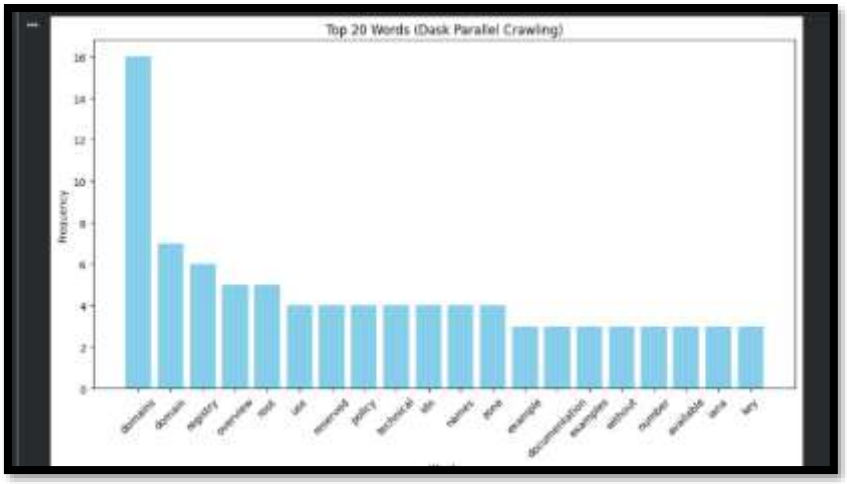


Figure 16

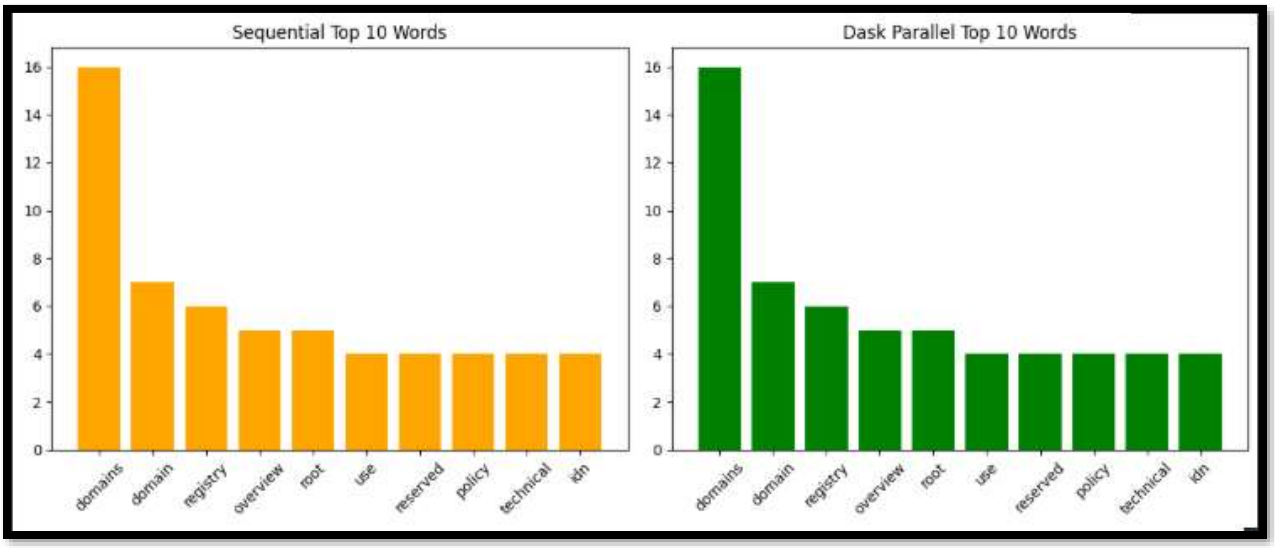


Figure 17

Data Preparation

- df_top20: Top 20 words from Dask crawling, sorted by frequency.
- df_seq_top10: Top 10 words from sequential crawling.
- df_dask_top10: Top 10 words from Dask crawling.
- Purpose: Create clean pandas DataFrames for plotting.

Top 20 Words Bar Chart

- ax1.bar: Plots a vertical bar chart of top 20 words.
- set_title, set_xlabel, set_ylabel → Labels and title.
- tick_params(rotation=45): Rotates x-axis labels for readability.
- Purpose: Shows the most frequent words from Dask crawling**.**

Sequential vs Dask Top 10 Comparison

- ax2 → Bar chart for top 10 sequential words (orange).
- ax3 → Bar chart for top 10 Dask words (green).

Purpose:

Side-by-side comparison of word frequencies between sequential and parallel crawling.

Word Cloud

- WordCloud(...).generate_from_frequencies(word_counts_dask) → Creates a visual cloud weighted by word frequency.
- ax4.imshow(..., interpolation='bilinear') → Displays the word cloud.
- ax4.axis('off') → Removes axes for a clean visual.
- Purpose: Intuitive visual representation of overall word frequency.

Layout and Presentation

- plt.subplot2grid((3,2), ...) → Organizes subplots in a 3-row, 2-column grid.
- colspan=2 → Makes the top and bottom plots span both columns.
- plt.tight_layout() → Automatically adjusts spacing to prevent overlap.
- Purpose: Combines all plots into a single professional figure for reports or slides**.**

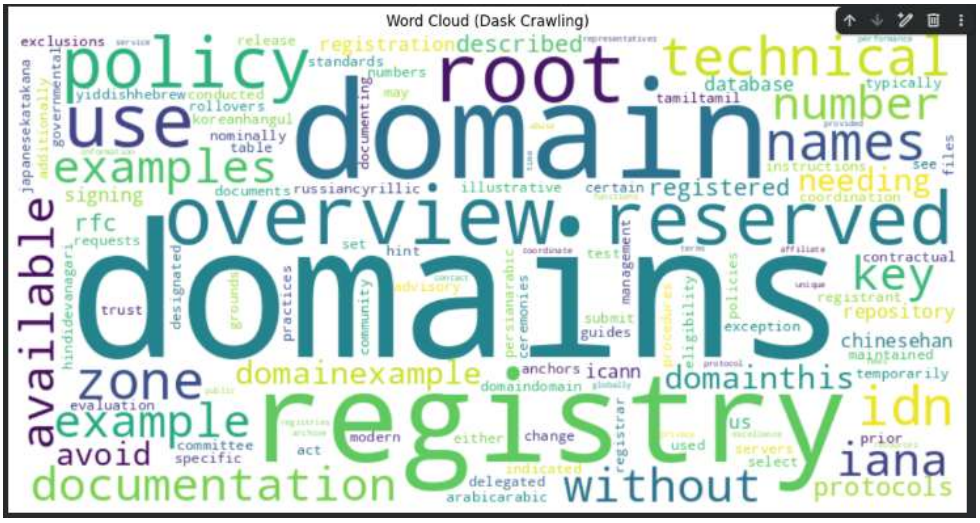


Figure 18

Challenges Faced

During the development of the distributed web crawler, several challenges were encountered. One of the main difficulties was handling multiple URLs simultaneously and ensuring that tasks were correctly distributed across workers using Dask Bag.

Another challenge was cleaning meaningful text from HTML pages. Web pages often contain unwanted elements such as tags, scripts, and navigation content, which required additional effort to extract readable text using BeautifulSoup.

The project also faced issues with broken or unreachable links, where some URLs returned errors due to network issues or restricted access. Proper exception handling was implemented to ensure that the crawler continued execution without failure.

Finally, understanding the Dask Bag execution model and its lazy evaluation mechanism required time and experimentation, particularly in knowing when and how computations are triggered using the compute() function.

Conclusion

This project successfully demonstrates the practical application of parallel computing through the implementation of a distributed web crawler using Dask. The crawler efficiently fetched multiple

web pages in parallel, extracted readable text using BeautifulSoup, and performed word-frequency analysis on the collected data.

By leveraging Dask for parallel execution, the solution reduced crawling time and improved CPU utilization compared to a traditional sequential crawling approach. Additionally, meaningful visualizations were generated to provide insights into the extracted web content. Overall, the project highlights how Dask can be effectively used for distributed and parallel data processing tasks in real-world applications.

Future Work

In the future, this project can be enhanced by running the crawler on a multi-node distributed cluster to process very large datasets efficiently. The crawler can be extended to perform deep crawling by automatically following links instead of using a fixed URL list.

Further improvements include applying Natural Language Processing (NLP) techniques such as sentiment analysis to better understand the extracted content, and storing crawling results in a MongoDB database for persistent storage and analysis. Additionally, a real-time dashboard can be developed to visualize crawling progress and analysis results dynamically.

References:

1. Dask Documentation
   Dask: Parallel Computing with Python
   https://docs.dask.org/en/stable/
2. BeautifulSoup Documentation
   BeautifulSoup: Screen-scraping library
   https://www.crummy.com/software/BeautifulSoup/bs4/doc/
3. Python Official Documentation
   Python Programming Language
   https://docs.python.org/3/
4. Requests Library Documentation
   Requests: HTTP for Humans
   https://docs.python-requests.org/en/latest/

5. Matplotlib Documentation
   Matplotlib: Visualization with Python
   https://matplotlib.org/stable/

Youtube learning Links

6. Web Scraping Using Requests & BeautifulSoup
   https://www.youtube.com/watch?v=GjKQ6V_ViQE
7. Python Web Scraping with BeautifulSoup
   https://www.youtube.com/watch?v=ng2o98k983k