CMPE300 Project-2

Zeynep Ebrar Karadeniz, Celil Özkan

1 Program Description

Our program processes an input text file and applies a series of preprocessing operations. After preprocessing, our program computes term-frequency (total count of each word in the vocabulary) and document-frequency (number of distinct sentences in which each word in the vocabulary appears) values for predefined vocabulary.

We implemented 4 MPI communication patterns:

Pattern 1: End-to-End Processing in Worker Processes

Before implementing any of the parallel patterns, we had our first challenge: installing mpi4py. For example, for me, the OpenMPI package that I installed was for x86 (even though I have a Mac) because my conda environment was x86, and this caused linking errors. Also, PyCharm was not using the correct virtual environment, which prevented mpi4py from being recognized even after installation. After I configured a proper virtual environment and reinstalled OpenMPI with the correct architecture, I was finally able to install mpi4py and proceed to implement Pattern 1.

We started with implementing small functions like distributing input text into chunks and preprocessing sentences. After writing helper functions, we started the main task: parallelism. The manager (rank = 0) should first send the config (vocabulary and stopwords), then data chunks to each worker process. Then each worker process will preprocess and count term-frequency independently. At the end, it will send its output to the manager, which will aggregate all using the Counter() function.

After writing main functions, we implemented this part pretty quickly without having major challenges.

Pattern 2: Linear Pipeline

For Pattern 2, we implemented a linear pipeline where each worker is responsible for a different stage of the text-processing workflow. Unlike Pattern 1, where every worker performs the same task on different chunks, in Pattern 2 each rank corresponds to a fixed processing step. The manager (rank = 0) first sends the config (vocabulary and stopwords) to all four workers. Then it splits the input text into ten chunks and sends each chunk to Worker 1, which begins the pipeline.

Worker 1 (rank = 1) lowercases the text and forwards it to Worker 2. Worker 2 (rank = 2) removes punctuation using a manual loop over every character. Worker 3 (rank = 3) removes stopwords from each sentence. Worker 4 (rank = 4) computes the term frequency using the filtered words and updates the Counter. When the manager finishes sending data, it sends a termination signal (None), which flows through the pipeline informing each worker. Once Worker 4 receives the termination signal, it sends the final term-frequency values back to the manager.

A challenge we encountered was how we would inform the other processes that the data was finished. We first tried using tags, but we could not get that approach to work reliably in the pipeline structure. Then we tried sending None as a termination signal and added an additional check so that each worker could detect when the received value was None. This allowed the termination signal to inform workers through the pipeline and made sure that Worker 4 could finish its work and return the final term-frequency values to the manager.

Pattern 3: Pipeline Parallelism (Multiple Independent Pipelines)

In Pattern 3, we implemented **multiple independent linear pipelines**. Each of these pipeline consists of four workers performing the algorithm just as we did in Pattern 2:

$$\begin{array}{cccc} & \text{Worker 1} \rightarrow \text{Worker 2} \rightarrow \text{Worker 3} \rightarrow \text{Worker 4} & & \\ \text{(Lowercase)} & \text{(Punctuation Removal)} & \text{(Stopword Removal)} & \text{(Term-Frequency)} \end{array}$$

The number of pipelines is determined by the total number of worker processes:

$$\text{num\_pipelines} = \frac{\text{size} - 1}{4}$$

The **manager process (rank = 0)** performs the following tasks:

1. Sends configuration data (vocabulary and stopwords) to all workers.
2. Divides the input text into large chunks, one per pipeline.
3. Splits each larger chunk into some smaller chunks and sends them to the first worker in each pipeline.

4. Receives partial calculated tf results from the last worker of each pipeline and aggregates them into a final term-frequency count.

Each **worker process** operates according to its stage in the pipeline:

- **Worker 1**: Converts text to lowercase.
- **Worker 2**: Removes punctuation manually by iterating over characters.
- **Worker 3**: Removes stop words.
- **Worker 4**: Computes term-frequency and accumulates results.

Workers pass processed data to the next stage, and the last worker sends the final pipeline results back to the manager. Termination is signaled by sending `None` through the pipeline.

Challenges encountered:

- I wanted to directly call **pattern 2** implementation since the **new pattern 3** task is only multiple pipeline version of the previously implemneted **pattern 2** code. However had some troubles with calling pattern2 code since it always assigns the first worker as manager. Then changed my design to write pattern 3 all by itself without calling patter 2.
- I forgot to pass the (`None`) word, which we introduced as **termination signal** through all stages initiallly caused some pipelines to hang indefinitely.

Pattern 4: Master–Worker with Task Parallelism (TF/DF Split)

Pattern 4 implements **end-to-end processing in worker processes with task parallelism**:

1. Each worker performs lowercase conversion, punctuation removal, and stopword removal on its assigned chunk of text.
2. After preprocessing, workers are split into two groups:
    - **Odd-ranked workers**: compute term-frequency (TF).
    - **Even-ranked workers**: compute document-frequency (DF).
3. Workers are paired to exchange preprocessed data asymmetrically to avoid deadlocks:
    - Odd-ranked workers send first, then receive.

- Even-ranked workers receive first, then send.

4. Each worker computes its assigned metric (TF or DF) on the combined data (own + partner's chunk) and sends the results back to the manager.

The **manager process**:

- Sends configuration and text chunks to all workers.
- Collects TF and DF counts from all workers and aggregates them into the final TF and DF results.

Challenges encountered:

- I mistakenly wrote the code to send data to the paired process first and then receive, which caused a deadlock scenario.
- I had problems starting the pattern with size = 9, since I don't have a machine with 9 cores. After adding the command –oversubscribe I could test it with larger n's.

2 Main Sample Output

Our results for the given sample:

```
Term-Frequency (TF) Result:
came: 30
come: 169
could: 52
day: 174
every: 180
go: 114
god: 61
great: 160
king: 2
know: 174
like: 127
little: 94
lord: 8
man: 197
```

```
one: 330
said: 26
see: 422
shall: 262
thee: 251
thou: 202
thy: 221
unto: 5
upon: 156
would: 92
ye: 32

Document-Frequency (DF) Result:
came: 23
come: 136
could: 45
day: 143
every: 151
go: 103
god: 51
great: 138
king: 2
know: 139
like: 110
little: 89
lord: 8
man: 156
one: 270
said: 24
see: 220
shall: 155
thee: 141
thou: 130
thy: 105
unto: 4
upon: 135
```

```
would: 76
ye: 26
```

3 Test Cases

We prepared five different test cases using samples from classic English literature and executed each test case 20 times for all patterns.

Test Case 1

This text is a sample from *A Tale of Two Cities* by Charles Dickens. The text contains 20 sentences, a vocabulary of 6 words, and 5 stopwords.

Command Example

```
 mpiexec -n 5 python solution.py
--text test_cases/text_1.txt
--vocab test_cases/vocab_1.txt
--stopwords test_cases/stopwords_1.txt
--pattern 4
```

Output

Term-Frequency (TF) Result:
age: 2, best: 1, epoch: 2, foolishness: 1, wisdom: 1, worst: 1
Document-Frequency (DF) Result:
age: 1, best: 1, epoch: 1, foolishness: 1, wisdom: 1, worst: 1

Test Case 2

This text is a sample from *Moby Dick* by Herman Melville. The text contains 22 sentences, a vocabulary of 5 words, and 6 stopwords.

Command Example

```
 mpiexec -n 5 python solution.py
--text test_cases/text_2.txt
--vocab test_cases/vocab_2.txt
```

```
--stopwords test_cases/stopwords_2.txt
--pattern 4
```

Output

Term-Frequency (TF) Result:
money: 1, sea: 1, ship: 1, soul: 1, water: 0
Document-Frequency (DF) Result:
money: 1, sea: 1, ship: 1, soul: 1, water: 0

Test Case 3

This text is a sample from *The Tell-Tale Heart* by Edgar Allan Poe. The text contains 21 sentences, a vocabulary of 6 words, and 5 stopwords.

Command Example

```
 mpiexec -n 5 python solution.py
--text test_cases/text_3.txt
--vocab test_cases/vocab_3.txt
--stopwords test_cases/stopwords_3.txt
--pattern 4
```

Output

Term-Frequency (TF) Result:
eye: 3, floor: 0, hear: 0, heart: 0, loud: 0, mad: 3
Document-Frequency (DF) Result:
eye: 3, floor: 0, hear: 0, heart: 0, loud: 0, mad: 3

Test Case 4

This text is a sample from *The War of the Worlds* by H.G. Wells. The text contains 20 sentences, a vocabulary of 6 words, and 5 stopwords.

Command Example

```
 mpiexec -n 5 python solution.py
--text test_cases/text_4.txt
```

```
--vocab test_cases/vocab_4.txt
--stopwords test_cases/stopwords_4.txt
--pattern 4
```

Output

Term-Frequency (TF) Result:
earth: 4, human: 1, mars: 4, men: 4, minds: 2, space: 2

Document-Frequency (DF) Result:
earth: 4, human: 1, mars: 4, men: 3, minds: 1, space: 2

Test Case 5

This text is a sample from *Alice's Adventures in Wonderland* by Lewis Carroll. The text contains 23 sentences, a vocabulary of 5 words, and 5 stopwords.

Command Example

```
 mpiexec -n 5 python solution.py
--text test_cases/text_5.txt
--vocab test_cases/vocab_5.txt
--stopwords test_cases/stopwords_5.txt
--pattern 4
```

Output

Term-Frequency (TF) Result:
alice: 9, dormouse: 2, hatter: 3, tea: 2, time: 1

Document-Frequency (DF) Result:
alice: 9, dormouse: 2, hatter: 3, tea: 2, time: 1

4 Work Sharing

Ebrar: Implemented Pattern 1 and 2 logic, added test cases.

Celil: Implemented Pattern 3 and 4 logic.

We reviewed and validated the test outputs and contributed to writing the final report together.