CMPE 300: Analysis of Algorithms

Project 2: MPI-Based Parallel NLP System

Efekan Kavalcı
efekan.kavalci@bogazici.edu.tr

Deadline: 8 December 2025, 23:59

1 Introduction

In this project, you will develop an MPI[1]-based program that takes a text file as input and applies a series of preprocessing operations. After preprocessing, your program will compute term-frequency (TF) and document-frequency (DF) values for a predefined vocabulary. These frequency measures are fundamental components of many natural language processing (NLP) systems such as those used for text analysis and vector-space representations.

Several MPI communication patterns will be implemented in this project. The patterns covered are listed below, and their detailed descriptions are provided in the subsequent sections.

- Pattern #1 – Parallel End-to-End Processing in Worker Processes
- Pattern #2 – Linear Pipeline
- Pattern #3 – Parallel Pipelines (Multiple Independent Pipelines)
- Pattern #4 - Parallel End-to-End Processing in Worker Processes with Task Parallelism

List of NLP Operations

- **Lowercasing:** Converts all characters in each sentence to lowercase.
- **Punctuation Removal:** Removes all punctuation symbols (e.g., commas, periods, question marks). You can use `string.punctuation` in python.
- **Stopword Removal:** Eliminates commonly used words such as “the”, “is”, “to”, and “in”, which are called stopwords.
- **Term-Frequency (TF) Counting:** Counts how many times each vocabulary word appears in the given set of sentences.
- **Document-Frequency (DF) Counting:** Counts in how many *distinct documents* each vocabulary word appears. You will consider sentences in the text as documents for the project.

[1]Message Passing Interface (MPI) Wikipedia page

Example Input

Consider the following three sentences as an example text and an example vocabulary that consists of three words (cat, dog, old):

```
The cat, the old cat, sat on the mat; the cat was tired.
A small cat chased the big dog! The cat didn't stop.
Dogs are bigger than cats, but a cat is faster.
```

After Preprocessing (Lowercasing, punctuation removal and stopword removal) (Using a simple stopword list: the, a, on, in, are, than, was, but)

```
cat old cat sat mat cat tired
small cat chased big dog cat didnt stop
dogs bigger cats cat faster
```

Term-Frequency (TF) Result Counts of each word in the vocabulary across *all sentences combined*:

```
cat: 6
dog: 1
old: 1
```

Document-Frequency (DF) Result Number of *distinct sentences* in which each word in the vocabulary appears:

```
cat: 3
dog: 1
old: 1
```

2 MPI Communication Patterns

2.1 Pattern #1: End-to-End Processing in Worker Processes (25 points)

The manager process divides the input text into balanced chunks and distributes them to worker processes that work in parallel. Each chunk should contain approximately the same number of sentences. For example, with 100 sentences and 5 worker processes, each chunk would contain about 20 sentences. A sentence should not be split across chunks.

Each worker performs the sequence of preprocessing operations: lowercasing, punctuation removal, stopword removal. Finally, the term-frequency counting operation will be done for the words in the vocabulary. You will not do document-frequency counting in this pattern. Workers return partial term-frequencies to the manager, which aggregates and prints the overall term-frequency results.

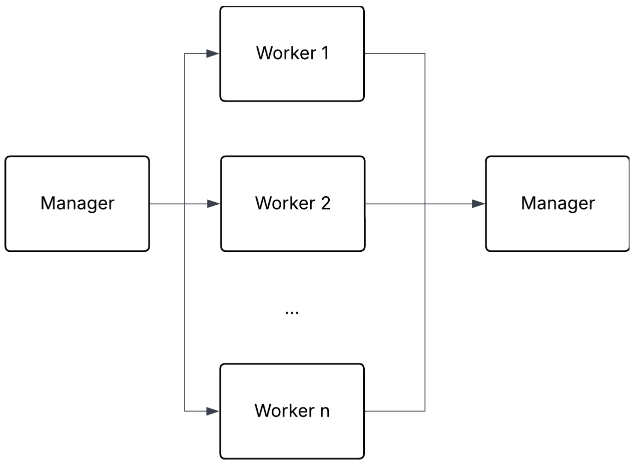

Figure 1: Diagram of Pattern #1

2.2 Pattern #2: Linear Pipeline (25 points)

In this pattern, each worker process is responsible for exactly one stage of the NLP processing pipeline. The data flows sequentially from one stage to the next, forming a linear chain of processing steps. Only term-frequency counting will be done, you will not do document-frequency counting in this pattern.

The manager process must send the input data in chunks, rather than sending the entire text at once. Sending all data at once prevents the pipeline from operating concurrently, since later stages cannot begin processing until earlier stages finish consuming their full input. Chunked communication ensures that each worker can start processing its portion while upstream ranks continue sending additional chunks.

Manager must send only to the Worker 1 in this pattern. Worker 1 receives a chunk from the manager, applies its operation (lowercasing), and sends the processed chunk to Worker 2. It then receives the next chunk from the manager and repeats the same steps until no chunks remain. Similarly, Worker 2 will receive from Worker 1 and send to Worker 3 and so on, resulting in the pipeline shown in Figure 2. Worker 4, which performs the final stage (term-frequency counting), accumulates the TF results as each chunk arrives. After all chunks have passed through the pipeline and each stage has completed its assigned operation, the manager obtains the final term-frequency results from Worker 4 and prints them.

The chunk size should be determined by dividing the total number of sentences by a value between 5 and 20. For instance, if the input text consists of 200 sentences and you choose to use a value of 10 to divide the text into chunks, then each chunk will be formed of 20 sentences. You can experiment with values outside this range, but the final value should be set within the given range.

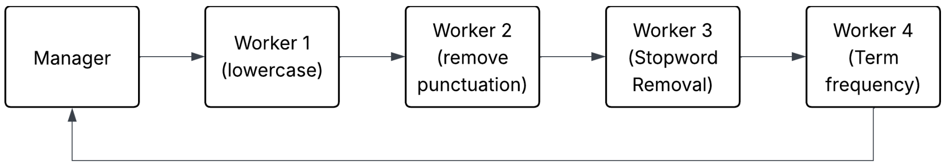
Manager
Worker 1 (lowercase)
Worker 2 (remove punctuation)
Worker 3 (Stopword Removal)
Worker 4 (Term frequency)

Figure 2: Diagram of Pattern #2

2.3 Pattern #3: Parallel Pipelines (Multiple Independent Pipelines)(25 points)

In this pattern, the system contains two or more independent linear pipelines operating simultaneously. Each pipeline consists of its own sequence of MPI ranks, and each worker within a pipeline performs a single, fixed stage of the NLP processing chain as in Pattern #2. Unlike the single-pipeline design of Pattern #2, here multiple pipelines run in parallel. You will not do document-frequency counting in this pattern.

The manager process is responsible for distributing data chunks across the available pipelines. Note that data partitioning occurs in two stages: first, the text is divided into larger chunks, one for each pipeline. These larger chunks should be obtained by dividing the total number of sentences by the number of pipelines. Then, within each pipeline, these chunks are further split into smaller pieces (by dividing the number of sentences in the chunk by a value between 5 and 20) to feed the workers, following the same chunking strategy used in Pattern #2.

Within each pipeline, data flows through the stages as in Pattern #2: each rank receives a chunk from the previous stage, applies its designated operation, and forwards the result to the next stage. Multiple pipelines operate in parallel.

After all chunks are processed by all pipelines, the final outputs are collected from the last stage of each pipeline and integrated by the manager. The manager then aggregates and prints the combined results.

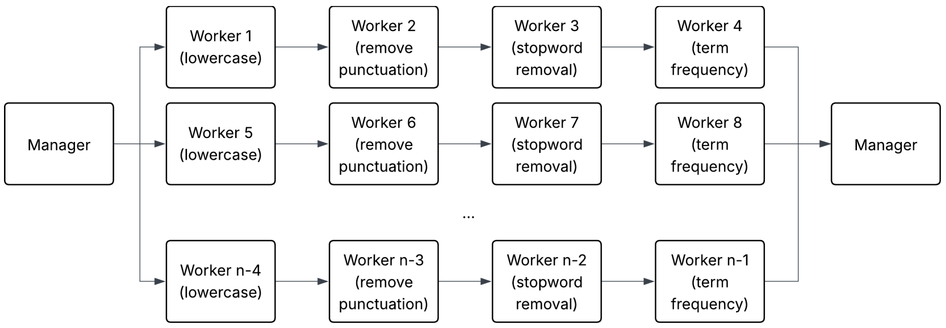
Manager
Worker 1 (lowercase)
Worker 2 (remove punctuation)
Worker 3 (stopword removal)
Worker 4 (term frequency)
Worker 5 (lowercase)
Worker 6 (remove punctuation)
Worker 7 (stopword removal)
Worker 8 (term frequency)
...
Worker n-4 (lowercase)
Worker n-3 (remove punctuation)
Worker n-2 (stopword removal)
Worker n-1 (term frequency)
Manager

Figure 3: Diagram of Pattern #3

2.4 Pattern #4: End-to-End Processing in Worker Processes with Task Parallelism (25 points)

In this communication pattern, each worker process first performs the three preprocessing operations (lowercasing, punctuation removal, stopword removal). This pattern introduces an additional computation: in addition to computing term-frequencies (TF), the system must also compute document-frequencies (DF).

The text is divided into balanced chunks as in Pattern #1, and these chunks are distributed to the worker processes. All worker processes apply the three preprocessing steps (lowercasing, punctuation removal, stopword removal). After preprocessing, unlike Pattern #1 where every worker performed the same task (term-frequency counting), here we will **split the two tasks across workers**.

Before splitting TF and DF tasks, workers must first exchange their data in pairs. With n worker processes, there will be n/2 such pairs. Within a pair, each process will send its own data to its partner and receive the data of its partner; and combine the two. After data exchange, both workers in each pair should have the combined data of the two. Then, the following split must be done for sharing term-frequency and document-frequency tasks (assuming rank 0 is the manager and worker processes start from rank 1):

- Even-ranked processes compute the **document-frequency (DF)**.
- Odd-ranked processes compute the **term-frequency (TF)**.

For example, Worker 1 performs TF counting on the merged data from Workers 1 and 2, while Worker 2 performs DF counting on that same combined data (see Figure 4). You must design the communication pattern so that:

1. No deadlocks occur. To avoid deadlocks during the send/receive phase, the communication pattern must be asymmetric. That is, one side (e.g., all even-ranked processes) must perform `send` first while the other side (all odd-ranked processes) performs `recv` first. If both sides attempt to send simultaneously, a possibility of a deadlock exists.
2. All TF and DF computations complete correctly for every worker.
3. The manager process collects the TF and DF results, aggregates and prints them.

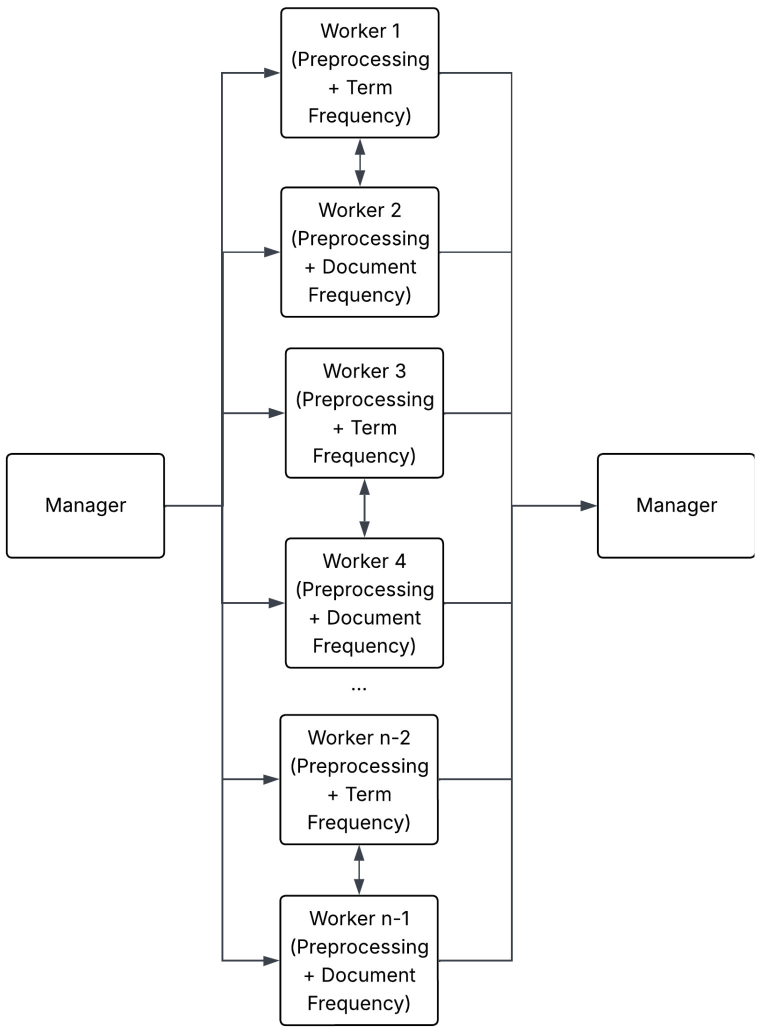
Worker 1 (Preprocessing + Term Frequency)
Worker 2 (Preprocessing + Document Frequency)
Worker 3 (Preprocessing + Term Frequency)
Worker 4 (Preprocessing + Document Frequency)
...
Worker n-2 (Preprocessing + Term Frequency)
Worker n-1 (Preprocessing + Document Frequency)
Manager
Manager

Figure 4: Diagram of Pattern #4

3 Implementation Guide & Submission

- The project will be implemented in Python using the `mpi4py`[2] library. You may choose an MPI implementation suitable for your operating system (e.g., OpenMPI, MS-MPI), `mpi4py` works on top of these standard MPI distributions. You will use the following core MPI functions:
  - `MPI_Comm_rank` → `comm.Get_rank()` in `mpi4py`
  - `MPI_Comm_size` → `comm.Get_size()` in `mpi4py`
  - `MPI_Send` → `comm.send(data, dest=..., tag=...)` in `mpi4py`
  - `MPI_Recv` → `comm.recv(source=..., tag=...)` in `mpi4py`

  Although the MPI standard provides many communication routines (e.g., `MPI_ISend`, `MPI_IRecv`, and various collective operations), **you are limited to using only `MPI_Send` and `MPI_Recv`** for point-to-point communication in this project. Non-blocking communication routines (such as `MPI_ISend` and `MPI_IRecv`) and all collective operations are prohibited.
- You should use the rank 0 process as the manager process, following the common practice.
- Your program must support different numbers of processes via the `-n` option of `mpiexec`. You can assume the following regarding -n (number of processes):
  - -n is at least 2 for Pattern #1 (1 manager and at least 1 worker process).
  - -n is exactly 5 for Pattern #2.
  - -n is in the form of $1+4i$, $i \geq 1$ for Pattern #3.
  - -n is in the form of $1+2i$, $i \geq 1$ for Pattern #4.
- Your program must support the following command-line arguments:
  - `--text`: Path to the input text file (in `.txt` format).
  - `--vocab`: Path to the vocabulary file (in `.txt` format).
  - `--stopwords`: Path to the stopwords file (in `.txt` format).
  - `--pattern`: Indicates which processing pattern to execute. Valid options are `1`, `2`, `3`, or `4`.

  You may use `argparse`[3] to parse command-line arguments.
- A sample text file, vocabulary file, and stopword file are provided in Moodle. You can use this sample set of files to see an example input and to test your program.
- An example command to run your program is: `mpiexec -n 9 python solution.py --text ./sample_text.txt --vocab ./vocab.txt --stopwords ./stopwords.txt --pattern 3`
- At the top of your main source file, include your names, student IDs, and any notes relevant for grading (e.g., known issues, partial functionality).

[2]mpi4py documentation
[3]argparse documentation

- Your program should be heavily commented and have explanatory names for the variables in the code.
- You should write a report that contains the following:
  - **Program Description:** Describe how you implemented your solution and the main challenges you encountered. Then explain, briefly but clearly, how your program satisfies the requirements of each pattern. This section should be a few pages long.
  - **Test Cases:** Prepare 5 different test cases by yourself, each test should contain the following:
    - A text file with varying lengths ($20 \leq \#$ of sentences ≤ 100)
    - A vocabulary ($5 \leq \#$ of words in vocabulary ≤ 15)
    - List of stopwords ($3 \leq \#$ of stopwords ≤ 10)

    Run each test case with all of the four patterns, i.e., there should be 20 runs total. In the report, provide the output and the command you used (i.e., `mpiexec -n 9 python ...`) for each of them.
  - **Work Sharing:** Explain the contributions of each group member and specify who worked on which parts of the project. Provide concrete details—avoid vague statements such as "we worked together."
- A zip file that contains your report, solution script and test cases must be submitted in the following structure (same with Project 1):

```
StudentNo1_StudentNo2.zip
-- StudentNo1_StudentNo2/
    |-- report.pdf
    |-- solution.py
    |-- test_cases/
        |-- text_1.txt
        |-- vocab_1.txt
        |-- stopwords_1.txt
        ...
        |-- text_5.txt
        |-- vocab_5.txt
        |-- stopwords_5.txt
```

- **Use of LLMs for writing the report or implementation is prohibited.**
- **The deadline is strict, late submissions will not be accepted.**