Distributed MapReduce Implementation Report

1. Basic Assumptions

- **Master-Worker Communication:**
  - Assumption: The master and workers communicate using gRPC.
  - Implementation:
    - gRPC services (`MapReduceService`) and RPCs (`Map`, `Reduce`, `Update`) are defined in `mapreduce/proto/mapreduce.proto`.
    - The master exposes a gRPC server, and workers connect to it as gRPC clients.
    - Workers request tasks (`Map`, `Reduce`), and the master responds with task details.
    - Workers send updates to the master (`Update`) upon completing their tasks.
- **Task Assignment:**
  - Assumption: The master assigns Map and Reduce tasks to workers.
  - Implementation:
    - Mappers request tasks by calling the `Map` RPC. The master assigns a unique Worker ID and an input file to each mapper. If no more map tasks are available, the master returns a Worker ID of -2.
    - Reducers request tasks by calling the `Reduce` RPC. The master assigns a unique Worker ID to each reducer. The master ensures that all map tasks are completed before assigning reduce tasks. If no more reduce tasks are available, the master returns a Worker ID of -2.
- **Data Handling:**
  - Assumption: Workers read input data, process it, and write intermediate/final outputs.
  - Implementation:
    - Mappers read their assigned input file, process the data (word count or inverted index), and generate intermediate files.
    - Reducers read the intermediate files, aggregate the data based on the task, and write the final output files.
    - Intermediate files are stored in the `datasets/intermediate` directory.
    - Output files are stored in the `datasets/output` directory.
- **Intermediate Data Partitioning:**
  - Assumption: Intermediate keys (words) are divided into buckets for `numReduce` reduce tasks.
  - Implementation:
    - Mappers hash each word to a reducer ID using the `sumOrdinals` function (`sum of ASCII values of characters in the word modulo numReduce`).
    - Each mapper creates `numReduce` intermediate files, one for each reducer, and writes the key-value pairs to the corresponding file based on the hash.

2. MapReduce Tasks Implementation

- **Word Count:**
  - Map Function:
    - Reads the input file.
    - Splits the content into words.

        - Counts the occurrences of each word.
        - Hashes each word to a reducer ID.
        - Writes intermediate files with "word count" pairs, partitioned by reducer ID.
    - Reduce Function:
        - Reads the intermediate files assigned to it.
        - Aggregates the counts for each word.
        - Writes the final output file with "word total_count" pairs.
- **Inverted Index:**
    - Map Function:
        - Reads the input file.
        - Splits the content into words.
        - Hashes each word to a reducer ID.
        - Writes intermediate files with "word filename" pairs, partitioned by reducer ID.
    - Reduce Function:
        - Reads the intermediate files assigned to it.
        - Collects the filenames for each word.
        - Writes the final output file with "word list_of_filenames" pairs.

3. I/O Format

- **Intermediate files:** Mappers create intermediate files in the `datasets/intermediate` directory. The files are named `intermediate-<worker_id>-<reduce_task_id>`. Each line in the intermediate file contains a key-value pair separated by a space. For Word Count, the format is "word count", and for Inverted Index, it's "word filename".
- **Output files:** Reducers create output files in the `datasets/output` directory. The files are named with the reducer's worker ID. Each line contains a key-value pair. For Word Count, the format is "word total_count", and for Inverted Index, it's "word list_of_filenames" (filenames separated by spaces).

4. Design Decisions

- The master waits until all map tasks are completed before assigning reduce tasks. This ensures that all intermediate data is available before reducers start processing.
- The number of reducers is configurable via a command-line argument, allowing for flexibility in partitioning the reduce workload.
- The `sumOrdinals` function is used for hashing words to reducer IDs. This is a simple hashing function for demonstration purposes. A more robust hashing function could be used for better distribution.
- The master uses in-memory maps (`mappers`, `reducers`) to track the status of workers and assigned tasks. This is suitable for this assignment's scope but might need to be replaced with a more persistent storage mechanism in a production environment.