SLR207 - Hadoop MapReduce from scratch

Emile Le Gallic

Table of contents

I. INTRODUCTION

1. Explanation of the Hadoop MapReduce Algorithm

In the Hadoop MapReduce algorithm:
- A **Master** node controls several **Slave** nodes.
- The Master node takes a text file as input and a list of hostnames corresponding to the Slave nodes. It outputs a sorted list of word occurrences in the input file.

The process involves **four** phases:

1. SPLITTING:
- The Master splits the file into several chunks, each of size (file_size)/n, where n is the number of Slaves. It then sends these chunks to the Slaves.

2. PROCESSING BY SLAVES:
- Each Slave executes the following steps:

a. Mapping:
- Each Slave takes its assigned chunk of the initial input file and produces a text file where each word from the chunk is on a new line.
  - Example:

```
[Chunk File]:
The cat is black

[Mapping Output]:
the
cat
is
black
```

b. Shuffling 1:
- For each word in the Mapping output file, the Slave computes an ID based on the word's hash, ranging between 1 and n (where n is the number of Slaves)
- The Slave creates several files, each containing words associated with a specific Slave ID, and sends these files to the corresponding Slaves (called Reducers).

c. Reducing 1:
- Each Slave processes the shuffling files received from other Slaves, creating a hash map of word occurrences from these files.
- The Slave then sends the minimum and maximum occurrences from its hash map to the Master.

3. GROUP REDUCTION:
- The Master computes reducing groups after collecting minimum and maximum occurrences from all Slaves.
- It determines the global minimum (min_g) and maximum (max_g) and creates n reducing groups, each responsible for a range of word occurrences.
- The range for each group i is defined as :

$$[min_g + i * \frac{(max_g - min_g)}{n}; min_g + (i+1) * \frac{(max_g - min_g)}{n}], \text{ for } i \in \{0,...,n\}.$$

4. Final processing by slaves:

- Each Slave performs the following steps:

a. Shuffling 2:

- Each Slave receives its assigned group and associated range, then processes the hash maps from the Reducing 1 phase.
- It creates files for each group containing words from the hash map whose occurrences fall within the group's range, and sends these files to the Slaves associated with those group IDs.

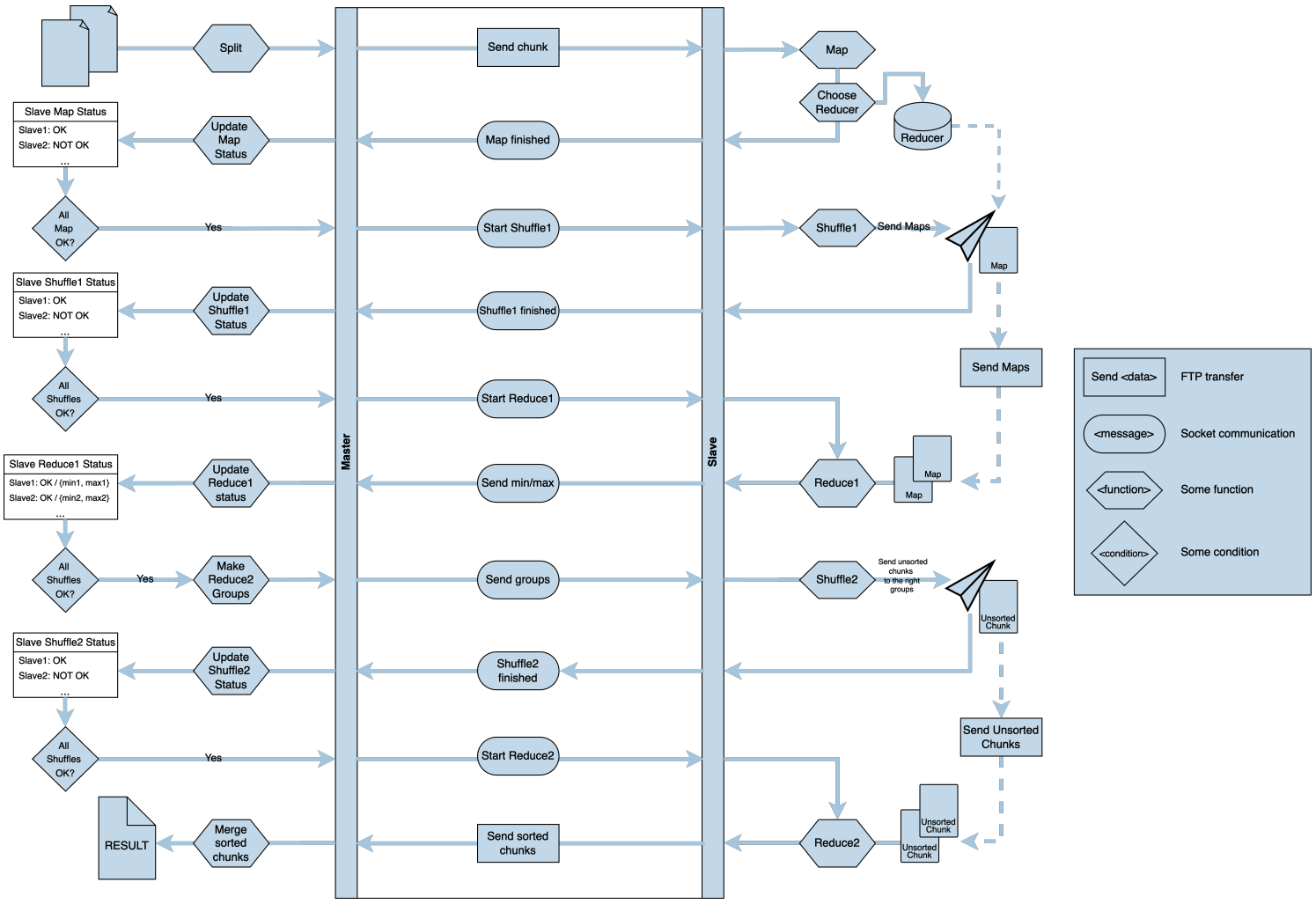
b. Reducing 2:

- Each Slave receives the files from other Slaves (from the Shuffling 2 phase) and sorts the words according to their occurrences.
- It then sends the sorted result back to the Master.

5. Aggregation:

- The Master receives all the sorted parts from each group and aggregates the results to build the final sorted list of word occurrences.

Workflow diagram



II. Implementation details

1. Master implementation

`Master.java`

The **`Master`** class orchestrates the entire MapReduce process, managing the communication with slave nodes, distributing work, and coordinating different phases of the MapReduce job (map, shuffle, reduce). It initializes communication handlers, splits the input file, sends tasks to slaves, and aggregates the final results.

Key responsibilities:

- **Reading slave hostnames:** It reads the hostnames of the slave nodes from a file.
- **Splitting the input file:** The **`splitAndSendChunks`** method splits the input file into chunks based on the number of slaves, ensuring that words are not split between chunks.
- **Sending tasks to slaves:** It uses **`CommunicationHandler`** to send file chunks and protocol messages to slave nodes.
- **Managing MapReduce phases:** It transitions between different MapReduce phases (map, shuffle, reduce) by sending corresponding commands to the slaves.
- **Aggregating results:** It gathers the results from slaves and compiles them into a final output file.

`CommunicationHandler.java`

The **`CommunicationHandler`** class handles the communication between the `Master` and each slave node. It establishes socket and FTP connections to send commands and transfer files.

Key responsibilities:

- **Connecting to slaves:** Establishes socket and FTP connections with the slave nodes.
- **Sending files:** Uses FTP to send file chunks to slaves.
- **Sending commands:** Uses sockets to send protocol messages to the slaves, instructing them to start different phases of the MapReduce process.
- **Receiving files:** Retrieves result files from slaves after the reduce phase.

`SocketThread.java`

The **`SocketThread`** class is responsible for maintaining the socket communication with a single slave node. It listens for messages from the slave and updates the **`Master`** about the progress of different phases.

Key responsibilities:

- **Listening for messages:** Continuously reads messages from the slave to monitor the progress of the MapReduce phases.
- **Updating Master:** Notifies the **`Master`** when a slave completes a phase (e.g., map, shuffle, reduce) by calling relevant update methods in the **`Master`**.

Detailed Implementation Points

File splitting

The **`splitAndSendChunks`** method in **`Master`** handles the splitting of the input file. Key points include:

- **Chunk calculation:** The file is divided into chunks based on the total file size and the number of slaves.
- **Avoiding word splits:** When splitting, the method ensures that chunks do not end in the middle of a word by checking if the last character of the buffer is a space. If it's not, it reads additional bytes until a space is found.

PROTOCOLS USED

- **FTP (File Transfer Protocol):** Used for transferring file chunks from the **`Master`** to the slaves and for retrieving result files from the slaves to the **`Master`**. FTP ensures reliable file transfer.
- **Sockets:** Used for communication between the **`Master`** and the slaves. Through sockets, the **`Master`** sends protocol messages (e.g., start map, start shuffle) and receives updates on the progress of each phase.

2. Slave implementation

`Slave.java`

The **`Slave`** class represents a worker node in a distributed MapReduce system.

KEY RESPONSIBILITIES:

- MapReduce phases:
  - *Map Phase:* Reads a chunk of data, splits it into words, and writes the words to an output file.
  - *Shuffle1 Phase:* Distributes the intermediate words to other slaves based on a hash function.
  - *Reduce1 Phase:* Aggregates word counts from the shuffle phase.
  - *Shuffle2 Phase:* Further groups the word counts and redistributes them.
  - *Reduce2 Phase:* Final aggregation and sorting of the word counts, preparing them for output.

`CommunicationHandler.java`

The **`CommunicationHandler`** class manages both FTP and socket communications for a slave node.

KEY RESPONSIBILITIES:

- **FTP Server management:**
  - *Initialization:* Sets up an FTP server for file exchanges between nodes.
  - *File transfers:* Handles uploading and downloading files via FTP.

- **Socket communication:**
  - *Initialization:* Establishes a server socket to listen for connections from the master.
  - *Message handling:* Sends protocol messages and data objects over the socket to the master.

`SocketThread.java`

The **`SocketThread`** class handles the socket communication in a separate thread.

KEY RESPONSIBILITIES:

- **Continuous communication:**
  - *Run method:* Listens for and processes messages from the master while running.

- **Message handling:**
  - *Command execution:* Executes commands based on received messages, such as starting the map, shuffle, and reduce phases.
  - *State updates:* Updates the state of the **`Slave`** class based on received messages.

Detailed Implementation Points

1. PROTOCOLS USED:
- **FTP:** For file transfer, ensuring that intermediate files are distributed among slave nodes.
- **Sockets:** For real-time communication and coordination between master and slaves.

2. SLAVE COMMUNICATION WITH MASTER:
- **Sockets:** Used for command and control messages (e.g., `START_MAP`, `START_SHUFFLE1`).
- **Initialization:** Master sends initialization commands which include the number of slaves, their IDs, and hostnames.

3. GATHERING SHUFFLE FILES FROM OTHER SLAVES:
- **FTP Protocol:** Used for transferring shuffle files between slaves.
- **Shuffle phase:** Each slave sends its shuffle files to the appropriate slave nodes using FTP. For example, `shuffle1` files are distributed based on a *hash function* to ensure balanced load distribution.

```
    public static int chooseReducer(String word, int numberOfSlaves) {
        int hash = Math.abs(word.hashCode());
        int reducer = hash % numberOfSlaves;
        return reducer;
    }
```

III. SETUP

To set up and run the Hadoop MapReduce system, follow these instructions:

1. Sequential Counter

The sequential counter is located in the sequential folder. To use it you can run :

```
java -jar ./target/sequential-1-jar-with-dependencies.jar <input file>
```

2. Slave Node

The slave JAR file is located at `slave/target/slave-1-jar-with-dependencies.jar`.

3. Master Node

The master JAR file is located at `master/target/master-1-jar-with-dependencies.jar`.
The `master/machines.txt` file lists the hostnames of the slave nodes.
The `master/data` folder stores some data samples used in tests.

4. Deployment and Execution

- To **deploy the slave JAR on the remote hosts**, edit the `deploy.sh` script to add your login and the path to the `.jar` and then run :

```
sh deploy.sh.
```

- To **start the MapReduce procedure**, use the following command:

```
java -jar ./target/master-1-jar-with-dependencies.jar <hosts file> <input file>
```

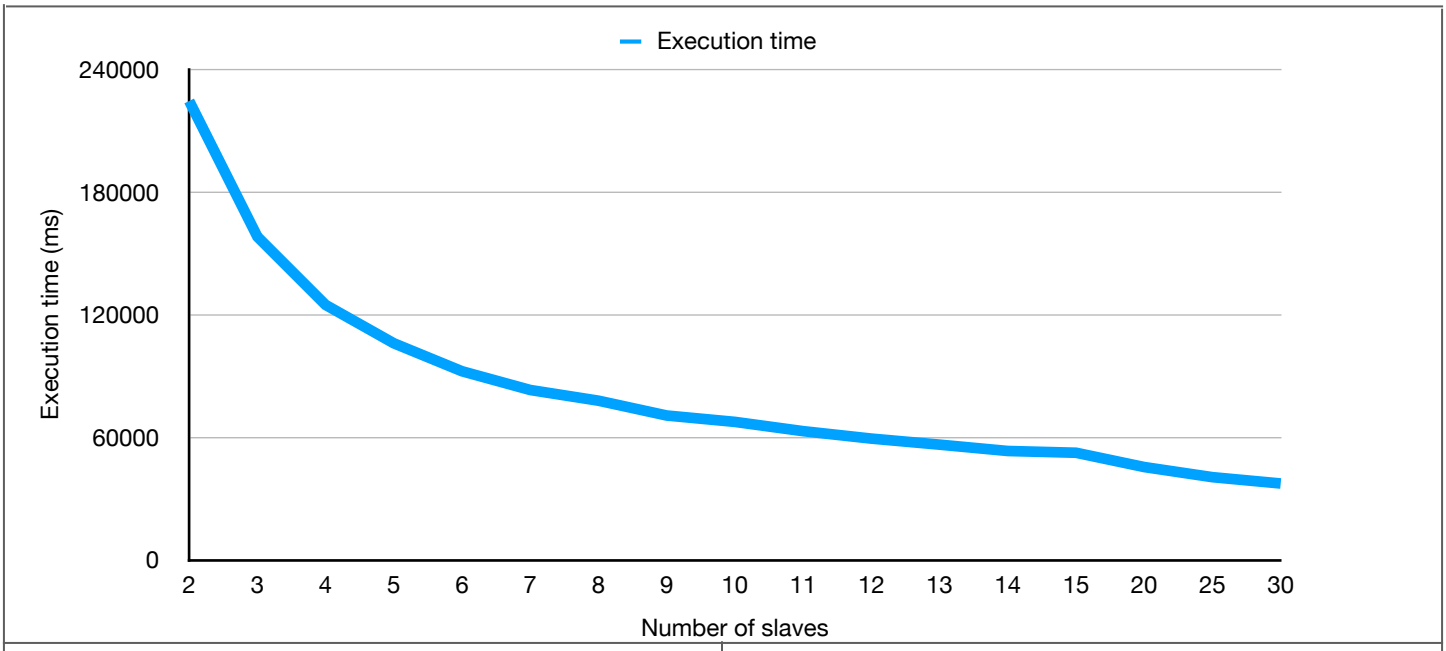
where `<hosts file>` is the file listing the hostnames of the slaves, one per line.

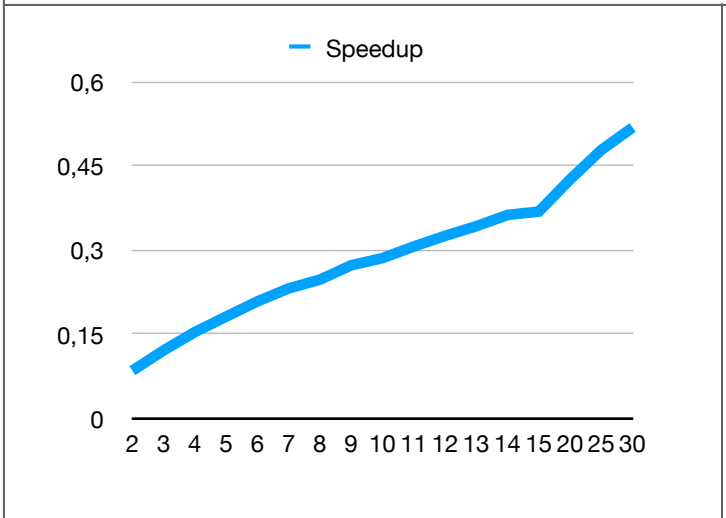
Example (run from the /master directory):

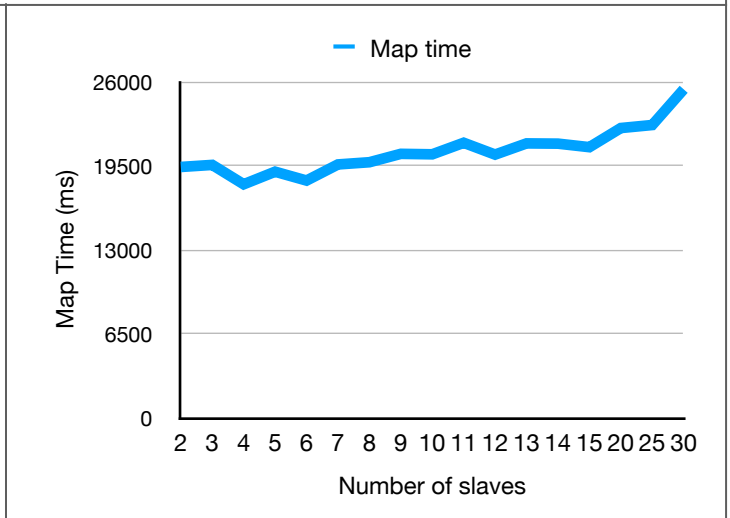
```
java -jar ./target/master-1-jar-with-dependencies.jar machines.txt data/CC-MAIN-20220116093137-20220116123137-00001.warc.wet
```

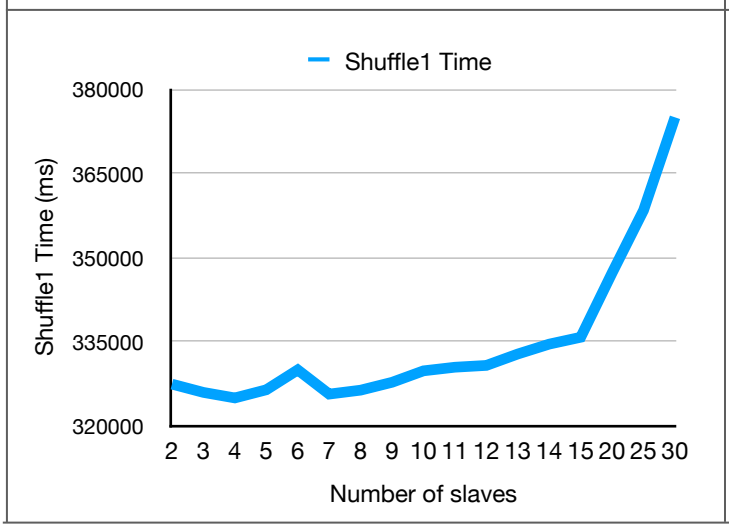
IV. Results analysis

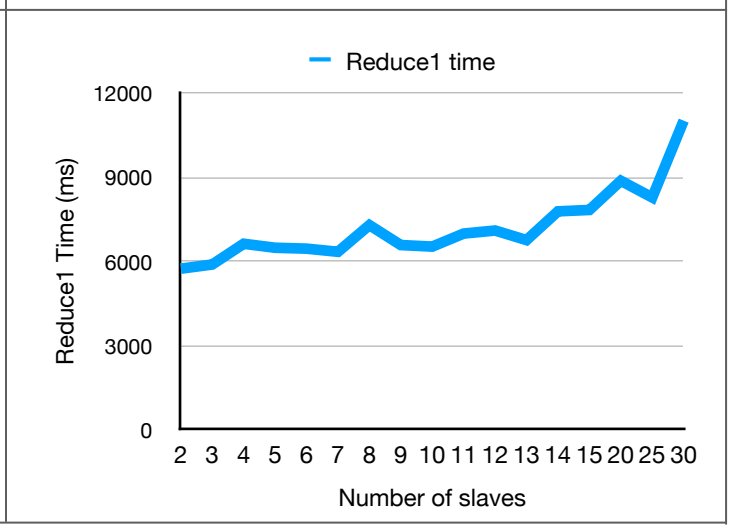
Detailed results can be found in the **`/report/results/results.md`** and **`/report/results/results.csv`** files.
The tests were carried out on the school network and with a single split of the common crawl, file of **370.6 MB**. For the speedup computation, the sequential execution gave a time of 19208ms.

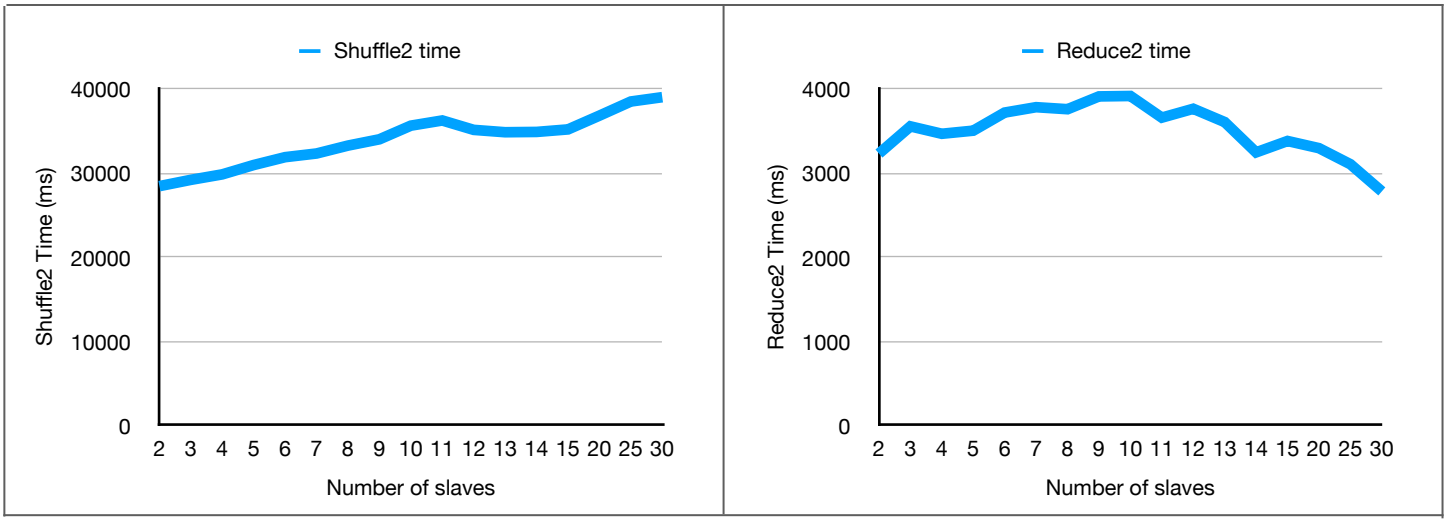












A quick analysis shows that :

- Map time doesn't vary significantly with the number of nodes. This suggests that the mapping phase is relatively consistent regardless of the node count.
- Both Shuffle1 and Shuffle2 times are highest for higher node counts and gradually decrease as the node count reduces. This indicates that shuffling data between nodes has a higher overhead when more nodes are involved.
- Reduce1 and Reduce2 times are also higher for more nodes and decrease as node count reduces, although the change is not as pronounced as in shuffle times. This suggests that reducing is somewhat more efficient with fewer nodes, possibly due to less communication overhead.
- The shuffle phases (Shuffle1 and Shuffle2) are consistently the longest stages.

However, one of the limiting factors is still the fact that the shuffle phase time is always very high, which is both due to the limitation of network file transfer speed via FTP as well as to poor parallelization of the shuffle process. Trying to shuffle files in different threads does not make the shuffle process any faster. The limitation therefore comes largely from the process of reading the map .txt files then writing the shuffle .txt files for the various other nodes. Writing/reading in Java is the most limiting, especially since the processes run on school machines which have a limitation in terms of computing capacity.

However, the speedup is not linear due to overheads such as communication, coordination, and data distribution.

Amdahl's Law states that the maximum speedup $S(n)$ of a task using n parallel processors is given by:

$$S(n) = \frac{1}{(1 - P) + \frac{P}{n}}$$

where P is the proportion of the task that can be parallelized, and $1 - P$ is the proportion that remains sequential.

Amdahl's Law highlights a fundamental limitation in parallel computing: as the number of processors increases, the impact of the sequential portion of the task becomes more significant, limiting the overall speedup. This means that even if a task is largely parallelizable, the non-parallelizable portion will ultimately constrain the speedup. For example, if 95% of a task can be parallelized ($P = 0.95$), the theoretical maximum speedup with infinite processors is only 20x.

In practical terms, our MapReduce implementation exhibits this behavior. While adding more nodes initially results in substantial speedup, the benefits diminish as the overheads related to data distribution, communication, and coordination grow. These overheads effectively act as the sequential portion in Amdahl's Law, capping the achievable speedup regardless of the number of nodes added. Thus, optimizing both the parallel and sequential components is crucial for maximizing performance.

V. Conclusion and possible improvements

To further improve the system, consider implementing a failure detector system to monitor the health of nodes. If a node fails, it can be excluded appropriately, ensuring the system continues to function smoothly without disruption. Additionally, optimizing the creation of groups for the Shuffle 2 phase by using word frequency density or other techniques can enhance the load balancing and efficiency of the MapReduce process. In addition to the mentioned improvements, enhancing the file reading/writing mechanism for the shuffle files could significantly boost computation speed. Since the shuffle phase typically consumes a considerable amount of time due to frequent disk I/O operations, introducing a faster file handling mechanism, such as utilizing memory-mapped files or employing parallel I/O techniques, could lead to substantial performance gains. By reducing the overhead associated with reading and writing shuffle files, the overall execution time of MapReduce jobs would decrease, resulting in faster data processing and improved system throughput.