# CSC_4SL07_TP: Distributed MapReduce from scratch

by: EL HELOU Bechara

GitHub repo: https://github.com/Bicho15H/Distributed-MapReduce

## 1. Overview:

This project implements a two-phase distributed MapReduce for word counting, written in Python. The system is designed to execute a word count task across a configurable number of machines, scaling from 1 up to N workers. It performs data splitting, shuffling, grouping, and reducing in a distributed fashion.

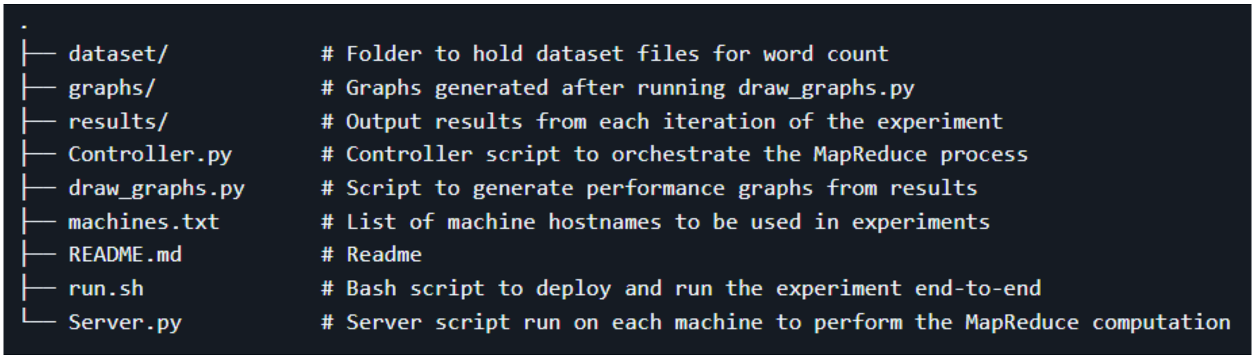
The project includes a controller coordinating the computation, a server script executed on each worker machine, and the result collection folder. After execution, it can generate performance graphs, highlighting phase timings and speedup behavior.

The MapReduce have two main phases:

1. First phase: Split → Shuffle → Synchronize → Group → Reduce
   At the end of the phase each server will have a max and min word count of its words.

2. Second phase: Shuffle2 → Synchronize2 → Group2
   At the end of the phase, each server will have a sorted word count within their assigned range.

## 2. Folder Structure:



```
.
├── dataset/            # Folder to hold dataset files for word count
├── graphs/             # Graphs generated after running draw_graphs.py
├── results/            # Output results from each iteration of the experiment
├── Controller.py       # Controller script to orchestrate the MapReduce process
├── draw_graphs.py      # Script to generate performance graphs from results
├── machines.txt        # List of machine hostnames to be used in experiments
├── README.md           # Readme
├── run.sh              # Bash script to deploy and run the experiment end-to-end
└── Server.py           # Server script run on each machine to perform the MapReduce computation
```

# 3. How it works:

### 1. Machines File

The *machines.txt* file should list, one per line, the hostnames of the machines that the controller will connect to (e.g., tp-1a226-02).

### 2. Running the Experiment

Execute the *run.sh* script from your local machine:

- *bash run.sh*

It will:

- Write the *machines.txt* file for each iteration (adding one machine at a time from 1 to N).
- Upload *Server.py* and *Controller.py* to the respective machines.
- Start the server scripts on the listed machines.
- Run the controller script on the controller machine.
- Collect output time results in the *results/* directory.

### 3. Generating Graphs

After all iterations are complete, run the graph generation script:

- *python3 draw_graphs.py*

This will process the time results in *results/* and save performance graphs in the *graphs/* directory.

### 4. Dataset

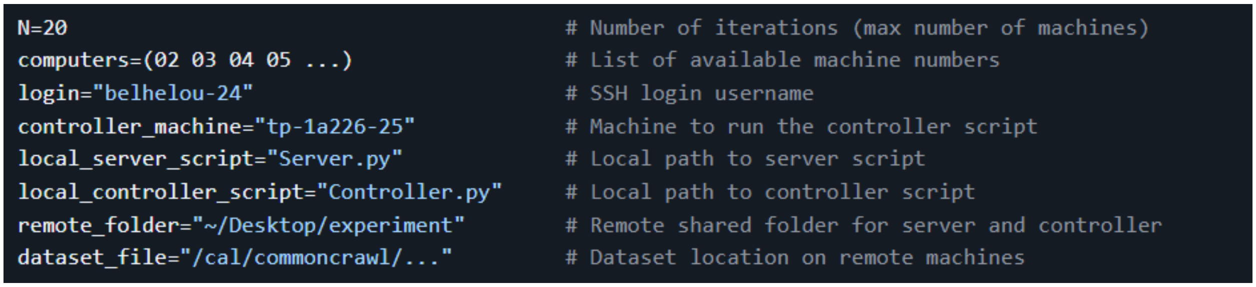
Place the dataset you want to analyze in the *dataset/* folder.

By default, *run.sh* will use the dataset from a shared location on the remote machines for efficiency:

- *dataset_file="/cal/commoncrawl/CC-MAIN-20230321002050-20230321032050-00486.warc.wet"*

### 5. Configuration

You can customize experiment parameters in *run.sh*:



```
N=20                                        # Number of iterations (max number of machines)
computers=(02 03 04 05 ...)                 # List of available machine numbers
login="belhelou-24"                         # SSH login username
controller_machine="tp-1a226-25"            # Machine to run the controller script
local_server_script="Server.py"             # Local path to server script
local_controller_script="Controller.py"     # Local path to controller script
remote_folder="~/Desktop/experiment"        # Remote shared folder for server and controller
dataset_file="/cal/commoncrawl/..."         # Dataset location on remote machines
```

# 4. Code overview:

## a) Server.py:

### i. Phase functions:

- **split():**
  - Handles the **SPLIT** phase.
  - Reads dataset files line by line, distributes words among servers using hash partitioning.
  - Prepares words_per_server mapping words to their destination servers.
- **shuffle():**
  - Handles the **SHUFFLE** phase.
  - Starts listening threads for incoming data from other servers.
  - Connects to peers and sends words to their designated servers.
  - Builds the local word count list for words that belong to this server.
- **synchronize():**
  - Waits for all SHUFFLE listener threads to finish.
  - Sends confirmation to the controller that the SHUFFLE phase is complete.
- **group():**
  - Aggregates words received from other servers.
  - Updates word_count_list by collecting counts for each word from the SHUFFLE listener threads.
- **reduce():**
  - Computes total counts for each word.
  - Determines global max and min word counts across this server's data.
  - Updates count_word_list mapping total counts to words.

- Exchanges range info with the controller to assign count ranges to servers.

- **shuffle2():**
  - Handles the **SHUFFLE2** phase.
  - Starts new listening threads for SHUFFLE2 communication.
  - Connects to peers and sends word lists grouped by their total counts to the server responsible for their count range.

- **synchronize2():**
  - Waits for all SHUFFLE2 listener threads to finish.
  - Confirms completion of the second shuffle phase to the controller.

- **group2():**
  - Aggregates final word groups received from SHUFFLE2 communication.
  - Updates final_count_word_list with words assigned to this server in the second phase.

- **quit():**
  - Finalizes the server's execution.
  - Optionally writes the final word counts to word_count.txt.
  - Cleans up resources: closes sockets, connections, and listener threads.
  - Notifies the controller of completion with QUIT_OK.

**ii. Support and utility functions:**

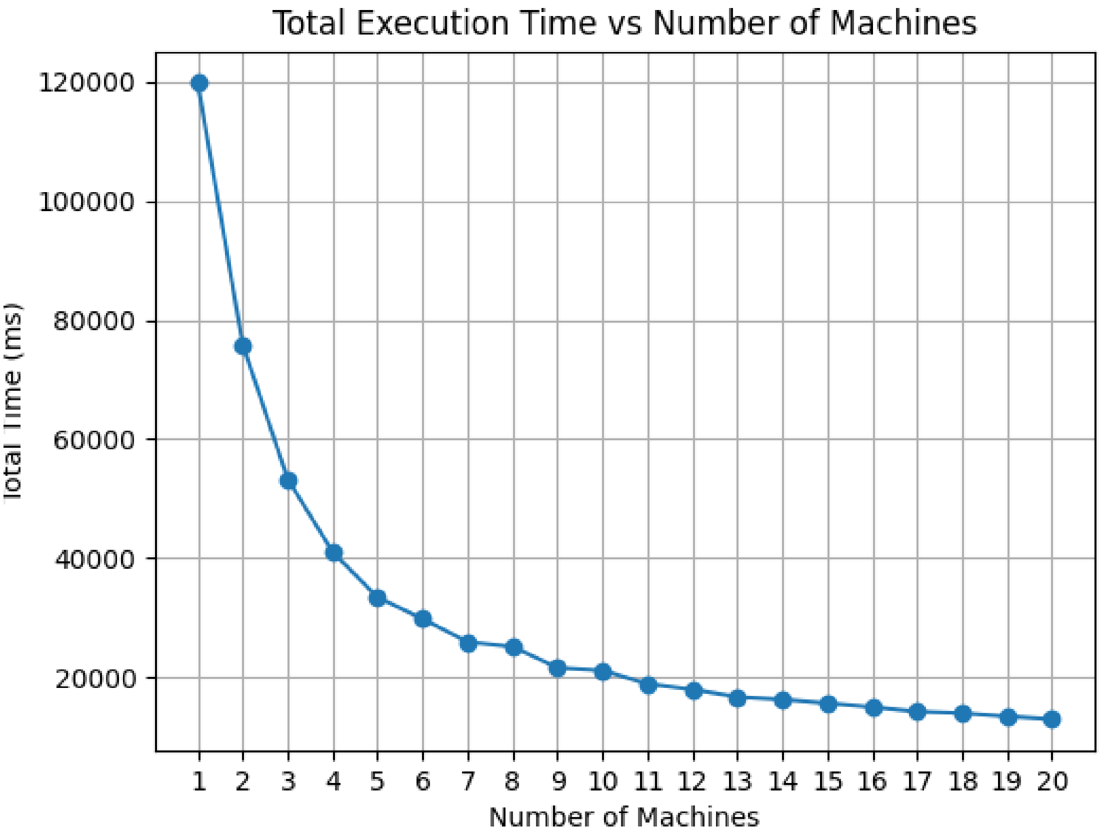
- **connect_to_peers():**
  - Connects to other servers in the cluster.
  - Returns a list of writable streams (peer_outputs) to send data directly to peers.
- **start_thread_listeners()**
  - Starts listener threads (Listener class) to receive incoming SHUFFLE or SHUFFLE2 messages.
  - Returns a list of running listener threads.
- **wait_threads()**
  - Waits for all listener threads to finish (join).
  - Closes listener sockets after they complete.

## b) Controller.py:

- Connects to each server listed in machines.txt on the specified port, and initializes communication streams.

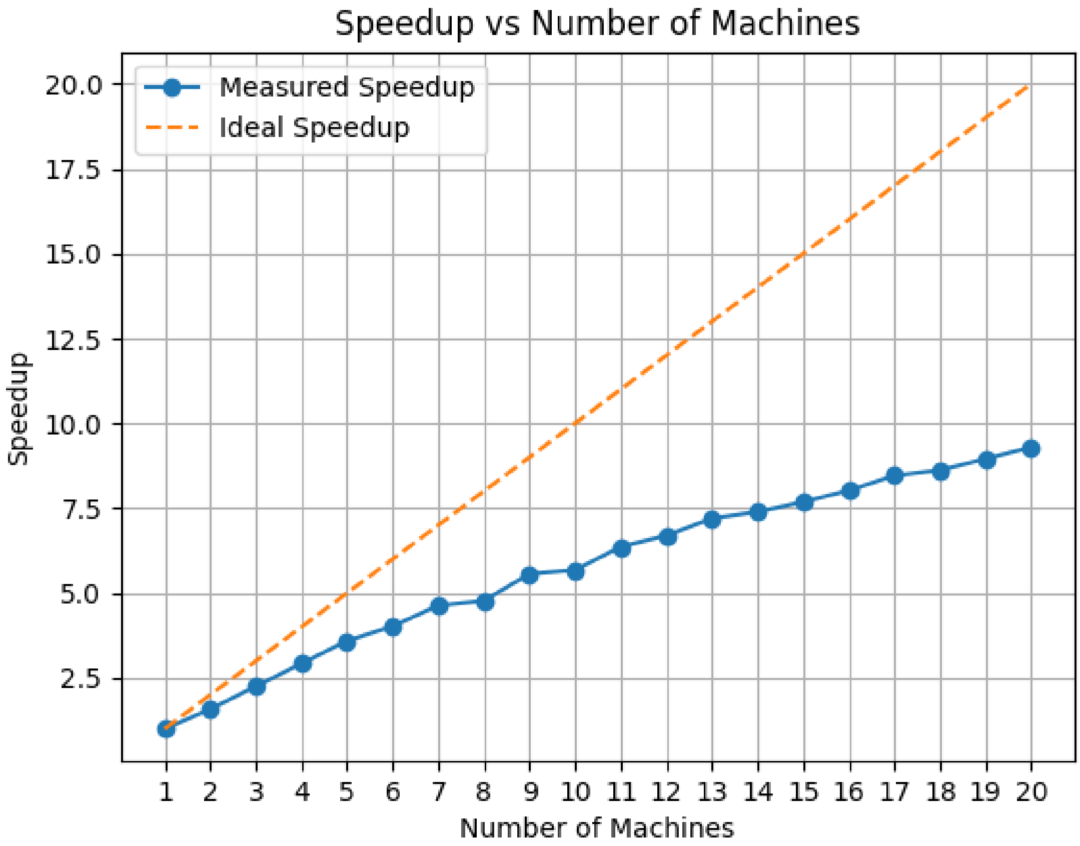
- Sends initial info to all servers, letting them know the list of participants and which one they are.
- Drives the MapReduce workflow by issuing a series of commands (SPLIT, SHUFFLE, GROUP, etc.), then:
  - Waits for servers' responses before advancing to the next phase.
- Calculates ranges after the REDUCE phase:
  - Gathers max/min word counts from all servers.
  - Computes the value ranges each server should handle.
  - Distributes these ranges back to servers.
- Tracks timing of computation, shuffle, and controller communication phases, and prints detailed timing summaries.
- Cleans up by sending QUIT and closing all sockets.

## 5. Results Analysis:



The total time decreases sharply as the number of machines increases.

As expected, adding machines provides diminishing returns after a certain point (~15 machines), indicating parallel overheads start to dominate.

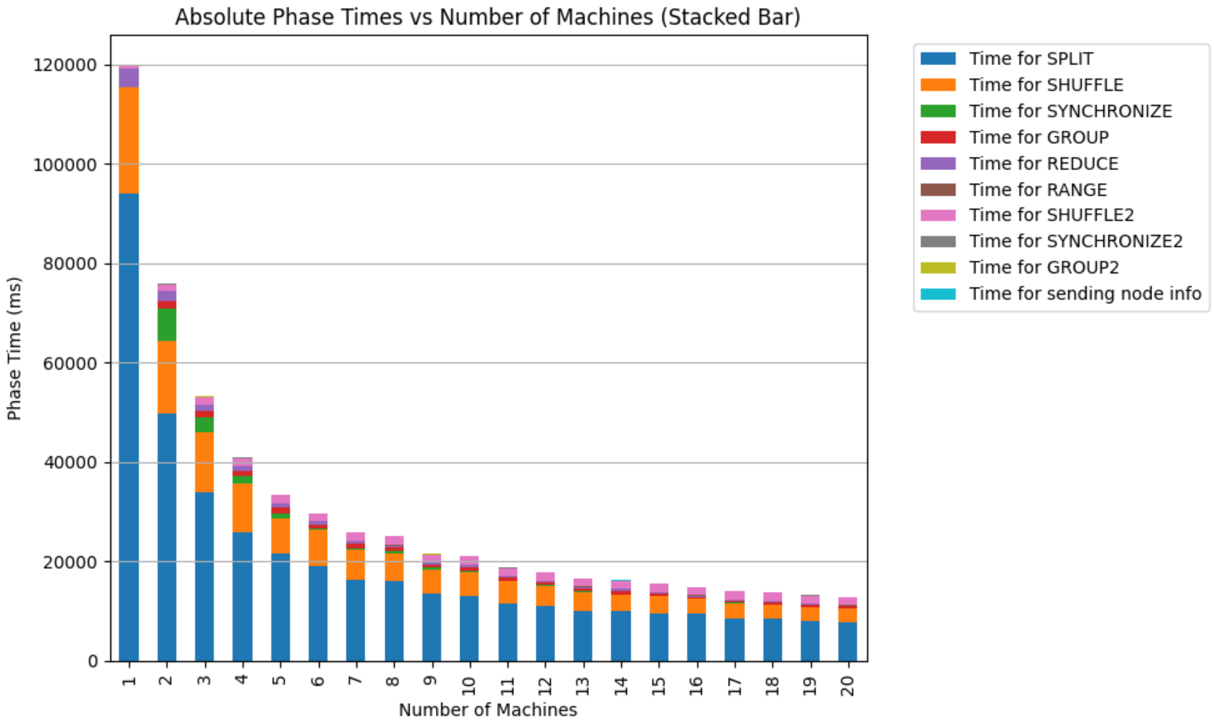


The blue line shows actual measured speedup, rising as more machines are added.

The Speedup is calculated by $T_1/T_n$ where $T_1$ is the execution time with one machine and $T_n$ is the execution time with n machines in parallel.

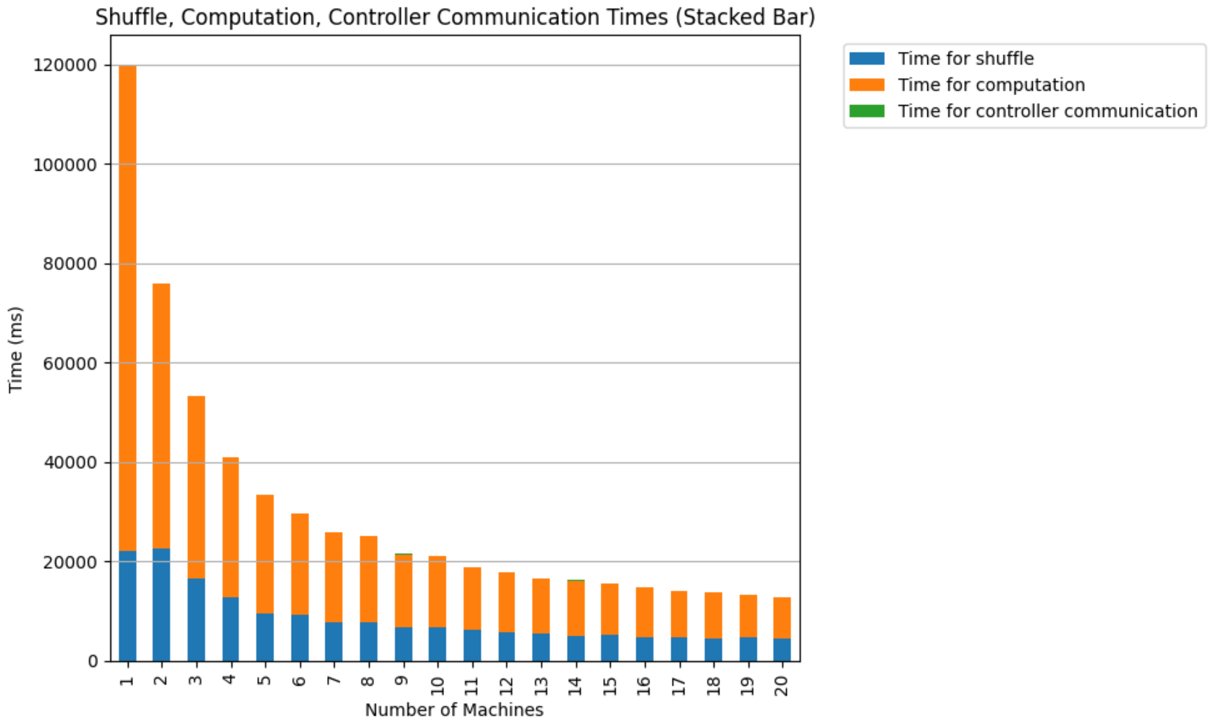
The orange dashed line is ideal linear speedup; the results shows near linear speedup at small scales, but saturates due to coordination and communication costs as the number of machines increases.

I also noticed that the graph is very similar to Amdahl's graph when P=0.95. So, I can conclude that my code is 95% parallelizable.



SPLIT and SHUFFLE phases dominate early execution times; their duration shrinks dramatically with more machines.

We also can notice that there is no synchronization with 1 machine since there is no one else to synchronize with.



Computation time decreases significantly as machines scale. Since the computation is now distributed on many machines so it will compute faster.

Shuffling between servers at first decreases but then stay consistent. This is due to the fact that the more machine, the more inter-communication between servers, so more time for synchronization between the servers.

Communication time between controller and servers is negligeable.