COL733 Assignment 1 Report

Kashish Goel(2021CS50602)

August 18, 2024

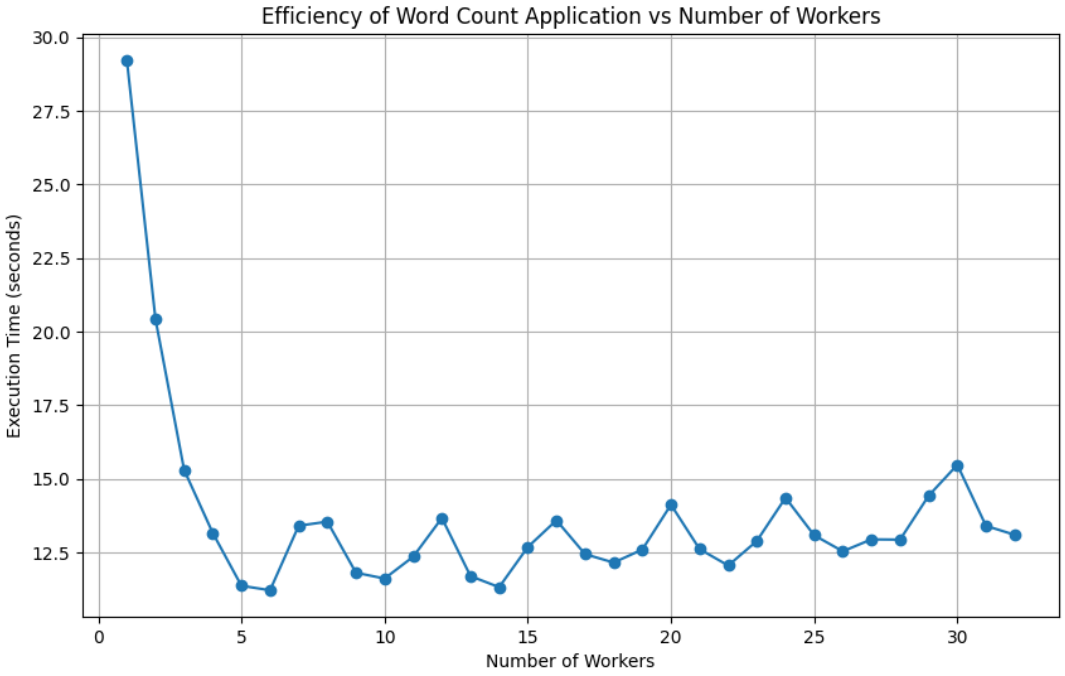
§1. Introduction

In this assignment, we aimed to build a distributed word count application where stateless python process act as distributed workers who co-ordinate with each other using **Redis**.
The application is worker as well as Redis server fault tolerant, i.e., if either of worker or Redis server crash, the application will still produce the correct results. We also utilise check-pointing by Redis to restore the data processed before crashing.

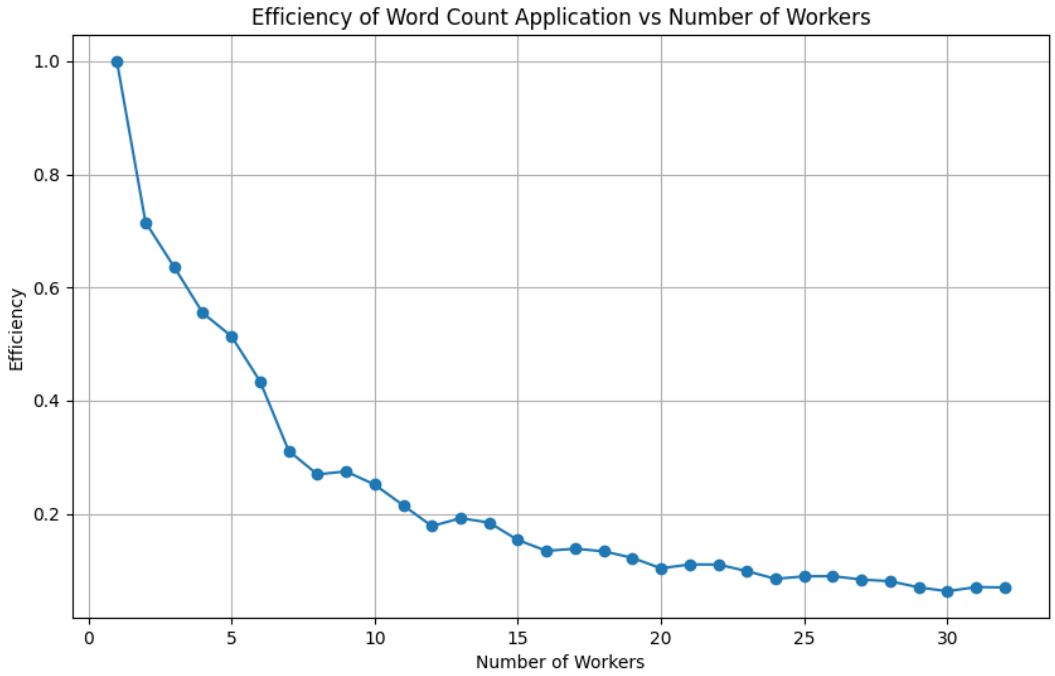
§2. Efficiency Analysis 1

Question: Given a fixed input size, measure how the efficiency of the word-count application varies with an increase in workers (in the range of [1, 32]) allocated to the application. Justify.

Analysis
I analysed the efficiency of the word-count application for the given dataset **twcs.csv** after creating 50 splits of the same using `split_csv.py`. The following graph reflects the time taken (in seconds) after varying the number of workers from 1 to 32 in fixed input size of around 175MB.



The following graph is a plot of the efficiency plotted as a function of the number of workers. To calculate efficiency, the reference time taken is that of run time of 1 worker.

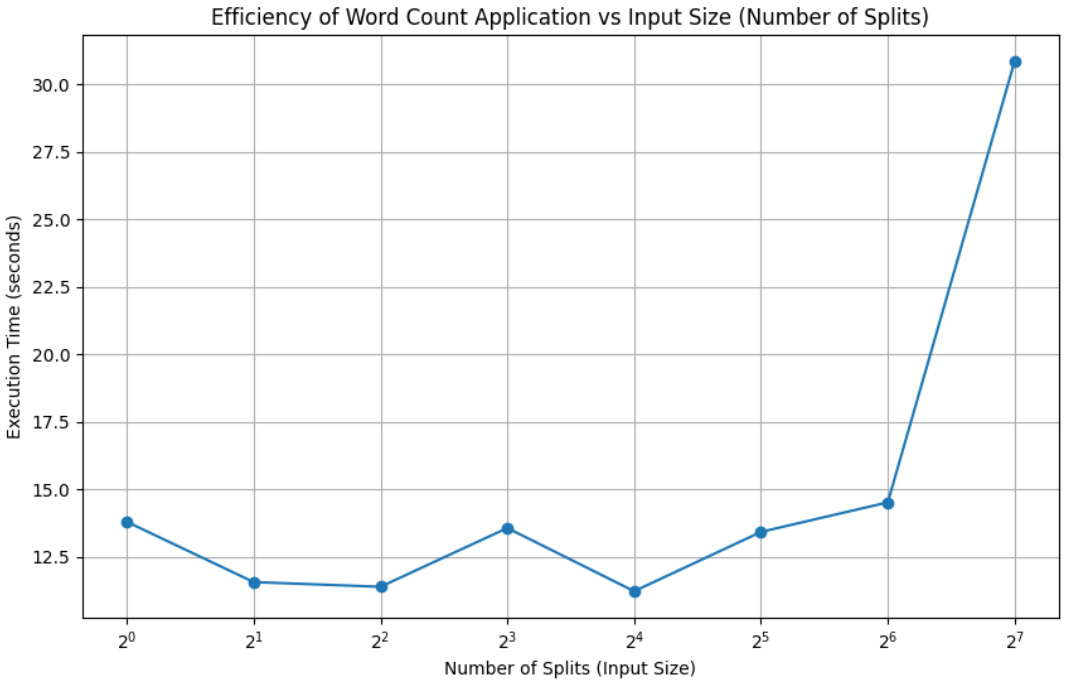


2.1. Justification

It can be seen that the time taken with multiple workers is less than the time taken by 1 worker as the work is distributed between multiple workers, it was expected that the application will be faster with respect to runtime.
The time taken reduces rapidly from 1 worker to 4 workers but it oscillates between 5 workers to 32 workers. This is because time has been measured on a 4 core machine. This prevents the overhead of creating different processes however with increasing number of workers, the overhead increases a lot.
The efficiency decreases with number of workers, as the speedup is not at par with the number of workers. This is due to the overhead created in running a new process.
The time taken by the function `atomicIncrementAndAck`, i.e., the lua script is a bottleneck as it makes update to sorted set in time **logN** and using one thread and hence slows down the process.

§3. Efficiency Analysis 2

Question: Given a fixed number of worker processes (= 8) allocated to the application, measure how the efficiency of the word-count application varies with input size. Justify.
Analysis
I analysed the efficiency of the word-count application for the given dataset **twcs.csv**. The following graph reflects the time taken (in seconds) after keeping the number of workers fixed to 8 and varying the number of splits from 1 to 128 in powers of 2 where each file measures around 3.5MB.



3.1. Justification

The time taken for splits from 1 to 4 decrease instantly. This is because we are working on a 4 core machine where only 4 workers can work at a time and hence if there are only 1 to 4 files, each worker would be assigned a separate file and hence the time is minimum. The initial drop in execution time happens because of better load balancing and parallel execution.
As the number of splits increases beyond 4, the efficiency begins to decrease due to a combination of process overhead and resource contention. When file splits exceed the number of cores (and workers), each worker has to handle multiple file splits. This leads to increased overhead in terms of scheduling, context switching, and communication with Redis.

Trend

- 1 File Split: The time is relatively high (13.79s) because there's no real parallelism. The entire input is processed by 1 worker (out of 8), leading to underutilization of resources.
- 2 to 4 Splits: Time decreases as parallelism improves. With more splits, the workers can share the load more evenly across cores, reducing the overall time. At 4 splits, the load is well balanced, and overhead is minimal.
- 8 Splits: Time rises slightly (13.56s) as overhead from managing more tasks and increased interaction with Redis begins to appear. More context switching and process management reduce efficiency.
- Beyond 8 Splits (16, 32, etc.): The rise in execution time continues (up to 30.86s for 128 splits). As the number of splits increases, the size of each task becomes smaller, increasing the task management and Redis interaction overhead. Redis becomes more of a bottleneck, and resource contention (CPU and I/O) leads to inefficient parallel processing.

§4. Worker Fault Tolerance

Question: Describe how your code is tolerant to worker failures. In other words, describe why your code is guaranteed to provide the same answer even if a worker crashes.
Answer: The code is tolerant to worker failures which means that even if a worker crashes the application provides the correct answer. A worker may crash at various following instances and the fault tolerance is ensured by the following:-

- **Failing just after reading the filename from the stream:** This can lead to problem as xreadgroup while assigning a file to a worker ensures that the same file has not been assigned to another worker. Hence, if a worker reads a file and crashes the same file can not be assigned to another worker using xreadgroup.
  It has been ensured that application is fault tolerant by autoclaiming a file by another worker if it has stayed with a worker for more than `min_idle_time`. The function `is_pending()` implemented in the Redis class tells whether there are pending tasks using the command `xpending`. If yes, and if a worker has not been able to get a new file from the stream, it checks if there are any files to claim using the `xautoclaim` command. If any worker has crashed after reading the filename, it will be later claimed by some other worker and hence will be processed. The application hence returns the correct answer.

- **Crashing after processing a file but before pushing the word count to Redis:** A situation similar to the one above will happen. This is because since the file has not been acked, it will remain in the pending stream and hence will be claimed by some worker later.

- **Crashing while incrementing the count:** In Redis the count increment happens one by one as in the words are put one by one in the queue and the processing is done by a single thread. If the worker crashes after putting the increment count for some words but not all then it will lead to inconsistent state.
  It has been ensured that the same does not happen by making the increment and acknowledgement of the processing of the file **atomic**. A function `atomicIncrementAndAck` was created and loaded into the redis server by means of a lua script. Within this function, we check if on acknowledging the given file, '1' is received. If yes, it means that the file was not yet processed by any other worker and we hence proceed with incrementing the count of the given words. If no, then the file has already been processed by some other worker and we hence don't increment the count.
  The function `atomicIncrementAndAck` is called using the redis command `fcall` which ensures that the function is run atomically which means that either the entire script function is run or it isn't. In such a case, it will not happen that the count of some words are increased and some are not. If the counts are not incremented at all, the file will not be acknowledged and will hence remain in the pending list of Redis following which it will be later claimed by some other worker. The application will hence remain worker fault tolerant.

- **Crashing after incrementing the count:** If the count has been incremented, it means that the file has been acknowledged (following atomic increment using `atomicIncrementAndAck` function and `fcall`, and hence it will no more in the pending list. The count for that file has been processed and it would not be handed over to any other worker and hence correctness is ensured.

- **Other cases:** In any other cases, such as crashing before reading a filename, crashing while processing the file, etc. it can be easily seen that the application remains consistent as it balls down to one of the above cases.

§5. Redis Fault Tolerance

Question: Describe how your code is tolerant to Redis failures. In other words, describe why your code is guaranteed to provide the same answer even if Redis crashes. **Answer:** The code is tolerant to Redis failures which means that even if Redis crashes the application provides the correct answer. Following is the tolerance procedure:-

- **Creating Checkpoints:** Checkpoints are created every `interval` seconds when the Redis server is on. Checkpoint is created using the command `bgsave` and done only when there are still un-processed files by using `is_pending()`. Using checkpoints, if Redis crashes, the server state is restored to the last checkpoint when it is started again as Redis automatically reads dump.rdb file (where checkpoint states are stored) and restore from it. Any file that may have been processed earlier but would not have been checkpointed would have been returned in the pending list and hence would be processed again. This ensures fault tolerance.
- **Worker Interaction with Redis while shutdown:** If a worker tries to run Redis commands while it has failed and hasn't started yet, it leads to connection error and the application crashes. To prevent the same, it has been ensured that redis commands are not run if the server is down and the worker waits for the server to be up again. If before reading the filename from stream, Redis crashes, the worker waits until Redis is up again and takes a new file input from it and processes the same. If Redis crashes after the worker has processed the file, it waits until Redis is up again and pushes the word count and later picks up a new file.
- **Ensuring server isn't accessed instantly:** If it is tried to access the Redis server immediately after it is re-run or it is tried to shutdown again, it may crash as running the server again has some overhead. Some sleep time has been put as a result.

§6. VM Configurations

Following are the configurations of the Virtual Machine, Baadal used:

- **HDD:** 80GB
- **RAM:** 8192 MB
- **Number of CPUs:** 4
- **Operating System:** Ubuntu 20.04 Server amd64 Docker

Requirements for running the application are as follows:

- mypy 1.4.1
- redis 7.4
- pandas 2.2
- python 2.10