A Python Implementation of MapReduce using Processes

George Boukouvalas
LIACS Leiden University
Leiden, Netherlands
s3645983@vuw.leidenuniv.nl

Srishankar Sundar
LIACS Leiden University
Leiden, Netherlands
s3151298@vuw.leidenuniv.nl

***Abstract*—This report covers the design and implementation of a minimal MapReduce system built using Python that can run both in stand-alone mode on a single node and in distributed mode across multiple nodes. Performance evaluation is conducted for both modes, using which we drive insights on both the implementation presented and the scalability of the same, and MapReduce as a whole. Limitations of the system are discussed and possible improvements are posited.**

I. Introduction

MapReduce, first proposed in 2004 by Dean and Ghemawat [1] is a programming paradigm that lends itself well to distributed processing on a cluster of nodes. It accomplishes this with the aforementioned paradigm - that of having the end-user break down their operations into two large categories - *map* and *reduce*. The rationale behind the paradigm is that every computation at a high level can be viewed as a set of map operations that yield key-value pairs obtained from the input data. These then act as the inputs in turn for the reduce operations that consolidate or reduce the values according to the intermediate keys provided. Since its introduction, MapReduce has seen various implementations, arguably the most prominent of which is *Hadoop*.

We take inspiration from MapReduce in two ways - first, the aspect of the paradigm itself, removed from the context of distributed computing. This led to the non-distributed implementation on Python. Secondly, since having it run across multiple nodes and scalability was an important criteria, we also present a distributed implementation of the same.

The goal behind the project is to gain familiarity with the paradigm and understand how very basic tasks we have frequently carried out as programmers (for example counting the number of words in the file, checking for palindromes etc.) translate or lend themselves to this type of structure. Secondly, we aim to derive inights on what could go into enabling the distributed aspect of the paradigm, and investigating aspects such as scalability of this simplified implementation.

The setup steps taken, the code used and any auxiliary files can be found in the repository linked - *Github*

II. MapReduce - A Brief Overview

A MapReduce program is composed of a map procedure, which performs filtering and sorting, and a reduce method, which performs an aggregation operation of all of the sorted outputs.

Besides the basic functional idea, MapReduce implementations that have seen usage do the following:

- Orchestrate the processing of the distributed servers
- Manage all communications and data transfers
- Provide redundancy and fault tolerance - dealing with node failures, operation failures etc.

The model is a special version of the divide and conquer algorithm with an addition of an aggregation step at the end. It is inspired by the map and reduce functions commonly used in functional programming. Despite being inspired by functional programming paradigms, its execution time is not faster in single threaded implementations and the gains of performance are only seen in multi-threaded implementations such as multi-processor hardware. Although it was created by Google it's popularized as a part of Apache Hadoop's open source distributed package.

III. Requirements for the Distributed System

We start off by defining certain requirements for the distributed system we are building.

A. Functional Requirements

- Must be based on MapReduce: The core functioning of the system enables computation using the mapreduce paradigm. This implies that beyond all functional details, at a high level the system must be able to execute the given map and reduce tasks
- Distributed and non-distributed modes: The system must support two implementations - that of executing the task on both a single system as well as making use of multiple systems
- Modular and usable implementation of the base system by a third-party: The system should be implemented such that the definition of tasks that can subsequently be run on the system can be done by any number of third parties. This is to be done through appropriate abstraction

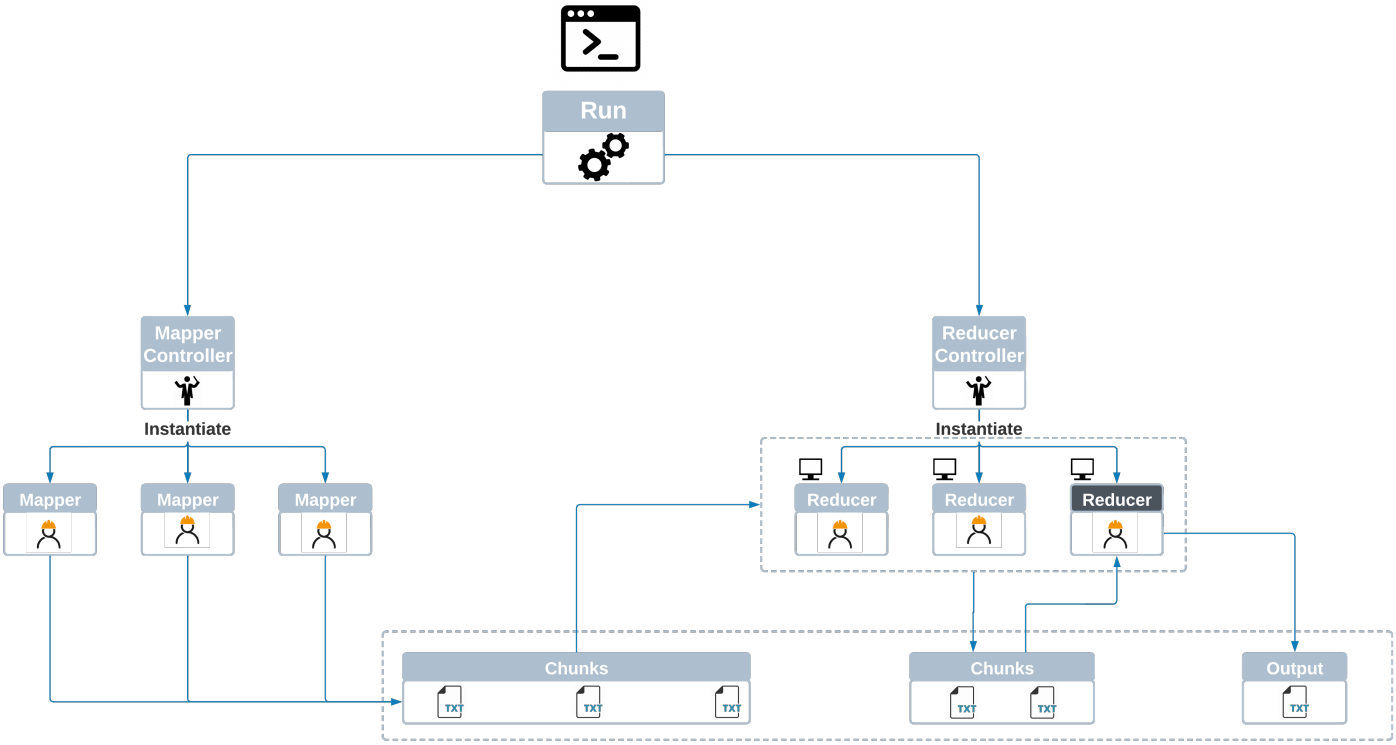

Fig. 1. High-level Solution Architecture

B. Non-functional Requirements

The non-functional requirements of the system are given below. These are decided based more on the higher-level goals we aim to achieve with this system.

- Scalability: The system should exhibit reasonably better performance when the number of nodes being used increases
- Performance: The system should show reasonable performance across both modes, and the same must be explainable

IV. Python Implementation

A. Solution Architecture

The architecture of the system is illustrated below. Conceptually, we can split it into the following segments:

1) Controller Script: This script triggers the system's Run method, which is discussed in the coming sub-section. The run method is triggered with an argument that specifies whether the run method should invoke the mapper controller, reducer controller or both. The scenarios for the three cases will be discussed in the coming sub-sections.

2) Run Method: The Run method is at the outermost level and takes in the input and certain parameters from the end-user's program. It is responsible for invoking the mapper and reducer controllers. Whether it invokes both or one of them is controlled by the controller script.

3) Mapper Controller: The Mapper Controller spins up the required number of mapper processes, each of which execute the map operation and yield a chunk containing key-value pairs.

4) Reducer Controller: The Reducer Controller spins up the required number of reducer processes which take in the chunks and yield a consolidated set of reduced chunks. The final Reducer process, illustrated in the figure in a darker shade is also additionally responsible for consolidating all the chunks outputted by the reducers into one final output file.

5) Storage: The storage runs off a shared filesystem among the nodes for the sake of simplicity, and is thus not distributed.

B. Flow

The flow and interactions between the aforementioned components is discussed below:

- The input data resides on a shared filesystem, which is an important underpinning of our implementation. The shared filesystem is used by all nodes

- The controller script starts the task script (for example, word count) such that the mapper controller is invoked internally. To this end, the task script is written such that it can take in certain arguments from the controller script and pass them on to the run method that it inherits. This is an obvious limitation of the system, as ideally this part should be abstracted from an end-user. This is discussed further in the Limitations section
- The mapper controller spins up mappers for the task all of which run on a single system as multiple threads within a process, regardless of whether we run the distributed or the non-distributed implementation. This is yet another limitation, discussed in the Limitations section
- The mappers yield output files (which have been previously referred to as intermediate files) that are stored back onto the shared storage
- The controller script then invokes the reducer controller *n-1* times, where *n* is the intended number of reducers. Each of the *n-1* reducers are run on separate machines. This is accomplished by the script using *ssh* onto the appropriate machine and instantiating the reducer there. In the case of the non-distributed implementation, the reducer controller instantiates n reducers within the same process on a single system
- The reduced outputs are written as chunks onto the shared filesystem
- The *n*th reducer is invoked by the controller script with the additional functionality of consolidating all the reducer output files and writing the final output onto the shared filesystem

C. Processes and Threads

The implementation presented uses the **Process** class from the *multiprocessing* Python package at its core. This allows us to envision each computational unit in one of two ways:

- **As separate threads in a single process** - This is used in the non-distributed implementation where the system spins up multiple threads which show up within the same process running on a single system, and each thread carries out work on a discrete chunk of data

```
[ddps2212@node123 ~]$ top | grep python2
  810 ddps2212  20   0 1127948 982524   2292 R 100.0  1.5   3:13.59 python2
  810 ddps2212  20   0 1127948 982524   2292 R 100.0  1.5   3:16.61 python2
  810 ddps2212  20   0 1127948 982524   2292 R 100.0  1.5   3:19.63 python2
  810 ddps2212  20   0 1127948 982524   2292 R  99.7  1.5   3:22.64 python2
  810 ddps2212  20   0 1127948 982524   2292 R 100.0  1.5   3:25.66 python2
  810 ddps2212  20   0 1127948 982524   2292 R 100.0  1.5   3:28.67 python2
  810 ddps2212  20   0 1127948 982524   2292 R 100.0  1.5   3:31.69 python2
```

Fig. 2. A single reduce process running on a single node, with information presented over a span of a few seconds - Note that the first column is the PID

- **As separate processes across multiple systems** - This is used in the distributed implementation, where similar processes are spun up across multiple nodes, each working on their discrete chunks of data

V. Experiment Setup

A. Cluster

- **CPU**: 32 cores
- **Memory**: 64GB RAM with a cache/buffer of 26.67GB out of the 64GB
- **Nodes**: 1, 5 or 9 nodes, depending on the experiment

B. Environment

The experiments are conducted on a specific Python version along with a few other packages, listed below:

- **OS**: CentOS Linux release 7.9.2009
- **Python Version**: Python 2.7
- **multiprocessing Package Version**: 0.70a1

C. Datasets and Task

We use two variants of the same enwik9 dataset, which is a dump from Wikipedia and is used for the Large Text Compression Benchmark competition. The original dataset is a text file of 1GB in size, out of which we also create a 500MB version. The task that we use to run the experiments is that of *word count*, where the number of occurrences of each word is to be recorded for a given piece of text.

D. Experiment Design

We broadly define three scenarios upon which we conduct experiments. These are described below:

- **Non-distributed implementation using a single process for reducers**: In this setup we run all reducers as threads within a single process which naturally executes on a single system. We test this implementation with multiple configurations for mapper and reducer instances and for both the 500MB and 1GB datasets
- **Distributed implementation**: In this setup we run each reducer as a separate process on separate systems. We test this with two configurations, one containing 4 reducers running on 4 nodes and the other containing 8 reducers on 8 nodes
- **Non-distributed implementation using multiple processes for reducers**: To bridge the gap between the aforementioned two experiments we essentially force the distributed version to run on a single system, thus yielding multiple reducer processes . We test this with one configuration, i.e 4 reducer processes running on the same system

VI. Results and Discussion

A. Non-distributed Implementation Using a Single Process

We first present the results from a word count task on a 1GB file in figure 3.

We observe that there is a decrease in execution time with increasing number of splits and reducers, although the

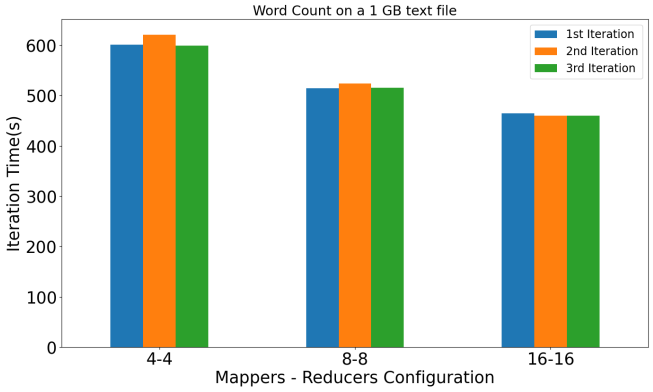

Fig. 3. Execution time for experiment with multiple reducers running as threads within the same process

relationship is definitely non-linear. We can postulate that the decrease in execution times is within expectation, as each reducer gets to work on smaller chunks of data, there are more reducers running at once. The bounding constraint is that there needs to be enough CPU resource to run the reducers in parallel.

We observe a similar trend with the word count task being run on a 500MB file, the results of which are also presented.

B. Distributed Implementation

The experiment for the distributed implementation is run on a 500MB dataset, with the same word count task.

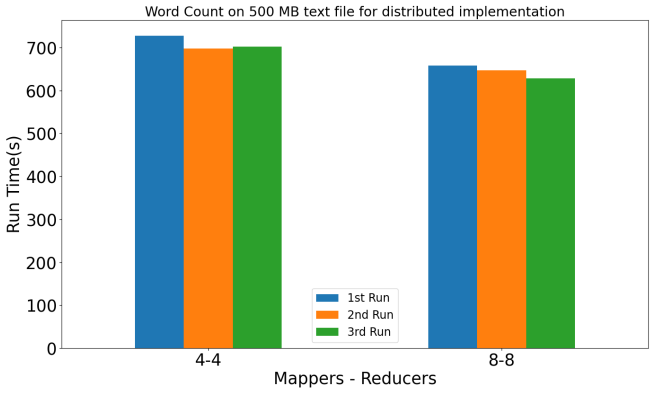

Fig. 4. Execution time using multiple reducers with distributed implementation

As seen in figure 5 we notice a decrease in execution times between the two configurations, that of 4 and 8 nodes. We note, importantly, that the quantum of decrease is less significant compared to the decrease seen in the non-distributed implementation with threads shown in the sub-section above. The basic execution time is also comparatively higher (12 minutes as opposed to 6 minutes for the 500MB dataset).

The reason for the discrepancy is discussed in the coming sub-section where we depict a comparison of the multiple experiment setups.

C. Non-distributed Using Multiple Processes

Finally, we compare the previous two setups with a third one, that of multiple reducers running on the same system as separate processes as opposed to threads within the same process (which is the case in the first non-distributed setup).

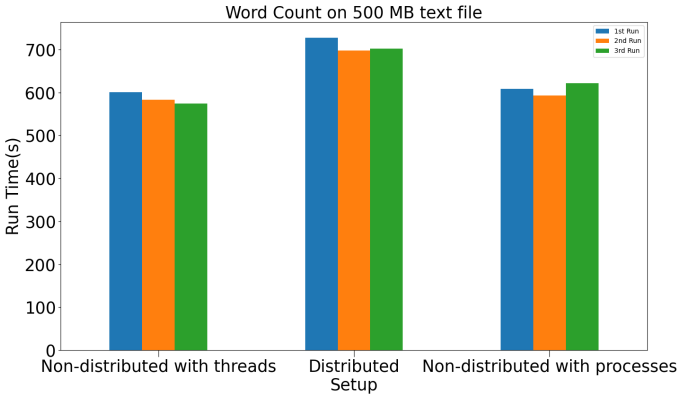

Fig. 5. Execution times with 4 reducers for non-distributed using threads, distributed and non-distributed using processes

We observe that running multiple reducers on the same system executes faster compared to running them in a distributed fashion. This is discussed further in the following sub-section.

D. Discussion - Discrepancy Between Distributed and Non-distributed Implementations

We intuitively expect execution times for the distributed implementation with 4 reducer nodes to be lesser than that of 4 reducers running on the same system as threads within a process. However, the results show the opposite. This is a result of the implementation.

To elaborate, the implementation of the non-distributed setup is as follows -

- We run *n* reducers in parallel, and once all reducers are finished we run the join operation. The join operation is still triggered from within the reducer controllers
- On the other hand, in the distributed implementation, we run *n-1* reducers in parallel but *wait* for them to complete before we execute the *n*th reducer, since the consolidation needs to happen from within a reducer

This is an obvious limitation, as the serialization of this final operation contributes to a very significant increase in execution time. This is also the reason the non-distributed implementation using multiple processes on the same machine (the third setup) also takes longer than the setup using threads within a single process.

It is important to note that **as the tasks scale up and the number of reducers increase, this discrepancy will narrow**, as the efficiency of the distributed implementation will start making up for the unwieldy serialization.

VII. Limitations and Further Improvements to the Design

Over the course of the report we have touched upon several limitations of the implemented system. These are consolidated and elaborated upon below:

- **User task scripts needing inputs from the controller script**: The task scripts need to be written in such a way that they can accept inputs from the controller script and pass them on to the run method. This has an impact on useability, as in an ideal system there is complete abstraction of these arguments from the user
- **The Mappers are not run in a distributed manner**: Although the mappers are concurrent, they are restricted to running on a single system. The ideal design might have been to have multiple systems for mappers or co-locate them to an extent
- **Serialization of the final reduce operation**: The biggest impedance to performance in the experiments conducted arises from this aspect of the implementation. The ideal design here would be to decouple the consolidation from the reducer and trigger it separately from the controller. This would allow all the reducers to run in parallel, thus greatly improving execution times for lower-scale tasks and datasets such as the ones used in the experiments

VIII. Conclusion

We presented an implementation of MapReduce in the form of a system that can process tasks written with this paradigm. To this end we created a class and defined the necessary functions in Python, and defined a controller script that invokes functions in specific ways. We evaluate the system with three setups, or variations in implementations. Through this, we see that the implementation of reducers as separate threads within the same process is the fastest performing, followed by an implementation of reducers as separate processes running on a singular node, and finally followed by the distributed implementation, where reducers run on separate machines.

We note that the system has multiple limitations, key among them being the serialized execution of the final reducer process and the following consolidation in the distributed implementation, which leads to the aforementioned hierarchy in performance.

Tasks	Person*	Time Taken**
System Design	Srishankar	4 days
Scripts for Experiments	Srishankar, George	1 day
Debugging, Code cleanup	George	1 day
Report Writing	Srishankar, George	1 day
Experiments and Visualizations	George	2 days

Appendix

A. Tasks and Roles

* Persons:
George - George Boukouvalas
Srishankar - Srishankar Sundar

** Time Taken: 1 day represents 1 working day. This is a rough estimate.

References

[1] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, jan 2008.