University of Reading | Department of Computer Science

MSc Data Science and Advanced Computing

Module Code: CSMBD21

Module Name: Big Data and Cloud Computing

Module Convenor: Professor Atta Badii

Student Number: 29805428
GitLab repository location: https://csgitlab.reading.ac.uk/iy805428/csmbd21-task-b
Document size: 3 pages (not including figures)

SECTION B: CLOUD COMPUTING

Abstract

The map-reduce paradigm offers a simple yet powerful approach for leveraging distributed and parallel processing capabilities to increase computational efficiency. In this project a software prototype is proposed for implementing a map-reduce-like programme for identifying passengers with the most flights from a provided list. The design successfully addressed the task through the use of a mapper-combiner setup and the deployment of multithreading. The code was hosted on a GitLab repository and was updated in line with version control best-practices. The rationale for the design, as well as the details of the implementation, are presented.

1) Task description

Given a list of passenger flights stored in a csv file (AComp_Passenger_data_no_error.csv) this report examines the task for determining the passenger(s) having had the highest number of flights.

This project involves solving the task of determining the passenger with the most flights from a list of 500 passenger flights.

2) Data exploration and preparation

As a first step, an examination of the csv file shows that it contains several exact duplicates, which must be removed before the map-reduce process can be performed. Removing duplicates brings the total list down from 500 to 389 unique flights. Another observation is that of the feature columns provided in the file, all but the first one (Passenger id) are not necessary the purposes of the current task. While they were retained for reference purposes, removing them has the potential to produce substantial efficiency gains when dealing with much larger datasets.

3) High-level description of the adopted solution

Different approaches were considered for designing and implementing this task. After experimenting with multiple options, a mapper-combiner design was deemed most appropriate for the present task. Figure 1 in Appendix A illustrates the design, which can be described as follows:

- A number of mappers is created based on the number of available cores.
- The data is partitioned into equal-sized chunks, also based on the number of cores.
- Each mapper is assigned to a partition/chunk, where for each PassengerID value it will return a (Passenger, 1) key-value pair.
- A combiner is used to group the outputs of the mapping phase according to passenger IDs, thus producing a list of PassengerIDs with their respective sequence of 1s.
- The combiner then reduces the list by summing up the total flights per passenger, returning the list in descending order.
- The mapping and combining tasks are distributed among the available cores via a multi-threading approach, whereby each mapper is assigned to a thread.

4) Design Rationale

The Mapper-Combiner design was adopted in great part because the task does not require a single final output but rather a list of calculated sums per passenger. As such the combiner is effectively replacing the reducer function by performing the aggregation step a reducer would typically perform in a standard MapReduce implementation. Using a combiner function instead of a reducer function can be useful when dealing with large datasets as it can help reduce the amount of data that needs

to be transferred between processes, since a combiner can be used to perform local aggregation of intermediate results within each chunk of data before the data is combined across all chunks. This can help to reduce the overall processing time and resource usage for the MapReduce pipeline.

5) Programme Implementation

In this section each component of the code is presented:

Importing libraries:

The following libraries are imported:

- *pandas* is used to read the input data from a CSV file and manipulate it as a DataFrame.
- *threading* is used to obtain the ID of the current thread running the mapper function.
- *os* is used to get information about the system, such as the number of available CPU cores.
- *functools.reduce* is used to apply the reduce function across a list of values.
- *concurrent.futures.ThreadPoolExecutor* is used to run the mapper and reducer functions in parallel across multiple CPU cores.

The code was divided into several main functions: pre-process, main, mapper, combiner, and run_map_reduce.

The main () function:

This function controls the main execution of the programme. It initially reads in a CSV file of passenger data into a pandas DataFrame, and calls the pre-process and run_map_reduce functions.

For this task, pandas dataframes are used to import the contents of the csv file. This is to allow maximum flexibility and stability during the running of the programme. However, a task involving much larger datasets could call for a different approach.

The clean () function

This function performs pre-processing task on the input data. It drops all columns except the first one and removes duplicates.

The mapper () function:

Mapper is the first function that is executed in the MapReduce algorithm. The function maps the input data to intermediate key-value pairs. More specifically each mapper takes a DataFrame chunk as input and then iterates through each passenger ID in the chunk, creating a key-value pair (passenger ID, 1), and appending it to a list. The function then returns this list of key-value pairs.

threading.get_ident() was used to obtain the current thread ID, which is then used to determine which CPU core the mapper function is running on. This was added principally for monitoring purposes, to examine the multi-processing execution of the task.

The mapped_data list is returned at the end of the function.

The combiner () function:

The combiner is the second function in the MapReduce algorithm. It takes the output of the mapper function, which is a list of key-value pairs, and groups them by passenger ID using a dictionary. The grouped_data dictionary stores the passenger ID as the key and a list of counts as the value.

The ThreadPoolExecutor is used to run the reduce function across all CPU cores in parallel. The reduce function calculates the total number of flights for each passenger by adding up the count values in the list of counts. This reduces the data to a list of tuples where each tuple contains a passenger ID and the total number of flights for that passenger.

The run_map_reduce () function:

run_map_reduce is the main function orchestrating the MapReduce algorithm. It is intended to allow the code to be easily adaptable for different map-reduce configurations. In the current implementation, the function first prints out the number of mappers, cores, combiners, and chunks being used. It then divides the input data into equal chunks, one per core. It uses a ThreadPoolExecutor to run the mapper and combiner across all processor cores. It then flattens the mapped data, using the combiner to group and reduce the mapped data, and converts the reduced data back into a pandas DataFrame. Finally, it prints out the top 10 passengers with the most flights.

Table 1 below illustrates the input and output of each of the map-reduce functions.

Function	Input	Output
mapper(data_chunk)	'data_chunk' is a pandas DataFrame containing a chunk of the input data.	Output: A list of key-value pairs, where each key is a passenger ID (int) and each value is 1 (int).
combiner(mapped_data)	'mapped_data' is a list of key-value pairs, where each key is a passenger ID (int) and each value is 1 (int).	A list of tuples, where each tuple contains a passenger ID (int) and the total number of flights for that passenger (int).
run_map_reduce(input_data)	'input_data' is a pandas DataFrame containing the input data.	This function does not return a value. Instead, it prints the top 10 passengers with the most flights as a pandas DataFrame with columns 'PassengerID' and 'TotalFlights'.

Table 1. Inputs and outputs of the map-reduce functions

6) Parallelisation and Multi-threading

For this task, the use of multi-threading was adopted to maximise parallel processing gains. The number of the threads was selected so that it matches that of mappers and cores. The ThreadPoolExecutor function from the concurrent.futures module is used to run the mapper function across all processor cores. The max_workers argument of ThreadPoolExecutor is set to os.cpu_count(), which is the number of available processor cores on the system. As such, the ThreadPoolExecutor creates a thread pool with the same number of threads as available cores.

The executor.map() method is used to apply the mapper function to each chunk of data in data_chunks in parallel across all threads. The mapped_data_chunks list contains the results of applying the mapper function to each chunk of data.

By using ThreadPoolExecutor and parallel processing, the mapper function can be applied to each chunk of data concurrently, which can significantly reduce the processing time compared to running the function on a single thread.

The combiner function is used to correctly sort and aggregate the output of the mappers for each passenger, even if the same passenger ID is present in multiple chunks (as long as the same passenger ID is represented with the same identifier).

7) Output format

The programme produces the output as a list of ten passengers with the most flights, in descending order, as show in Figure 2 in Appendix A.

The choice of ten, while relatively arbitrary, nevertheless allowed the observation that five passengers shared the top number of flights (17). Selecting to output only the single passenger with the most flights thus would have produced different results with each run, possibly causing confusion over the accuracy of the programme.

8) Object-Orientation Design Considerations

The project did not use a classic object-oriented programming (OOP) approach by creating objects and classes, largely due to the fact the task's complexity level did not require it. However, the programme design did adopt an approach based on designing key objects – such as the data chunks or the map-reduce calls – to be as self-contained as possible, so as to allow for a more intuitively object-oriented implementation.

9) Code Re-Usability and Design Optimality

For re-usability, the code was implemented in a modular approach, whereby each functional block was assigned to a single function that can be called separately from the main function, thus allowing for specific functional components – including different map-reduce implementations - to be updated, added or removed seamlessly.

10) Version Control

Throughout the design and implementation processes, different updates or iterations of the Python code - whether for the full protype or specific modules and components – were regularly updated on the GitLab repository, with the relevant comments documenting the key changes and rationales behind the update.

11) Conclusion

This project presented a MapReduce-like executable prototype developed in Python, which successfully provided the functional 'building-blocks' for carrying out the given task by emulating the MapReduce/Hadoop framework. The solution uses multi-threading to distribute the processing load among a single processor's cores but is readily adaptable to multi-processor contexts. This project has highlighted the importance of adopting the appropriate design, notably the choice of map-reduce architecture, to building successful distributed computing solutions.

Appendix A

GitLab instructions:

The implementation files are stored at the following GitLab repository location: https://csgitlab.reading.ac.uk/iy805428/csmbd21-task-b

The final coursework submission programme is the one provided in the FINAL folder and is comprised of two main files:

- FlightCounterMain.py [To be executed]
- FlightCounterCommon.py [Called by FlightCounterMain.py]

For reference and easier navigability, the files are provided in both .py and .ipynb versions.

High-Level Solution Design:

INPUT		PARTITIONING		MAPPER		COMBINER		OUTPUT
		SPLIT INTO CHUNKS		MAPPING		GROUPING BY ID	REDUCING (SUMS)	SORTING
PassengerID1	→	PassengerID1	→	(PassengerID1, 1)		(PassengerID1, (1,1,1,1,...))	(PassengerID1, 12)	(PassengerID3, 17)
PassengerID2	→	PassengerID2	→	(PassengerID2, 1)		(PassengerID2, (1,1,1,1,...))	(PassengerID2, (8)	(PassengerID1, 12)
PassengerID3	→	PassengerID3	→	(PassengerID3, 1)		(PassengerID3, (1,1,1,1,..))	(PassengerID3, 17)	(PassengerID2, (8)
PassengerID1	→	PassengerID1	→	(PassengerID1, 1)		...	...	...
PassengerID5	→	PassengerID5	→	(PassengerID5, 1)		...	...	...
PassengerID6	→	PassengerID6	→	...		...	...	...
PassengerID1	→	PassengerID1	→	...		...	...	...
PassengerID8	→	PassengerID8	→	...		...	...	...
PassengerID3	→	PassengerID3	→	...		...	...	...
PassengerID10	→	PassengerID10	→	...		...	...	...
PassengerID5	→	PassengerID5	→	...		...	...	...
PassengerID12	→	PassengerID12	→	...		...	...	...
...		...						

Figure 1. High-level Mapper-Combiner design

Programme Output Sample:

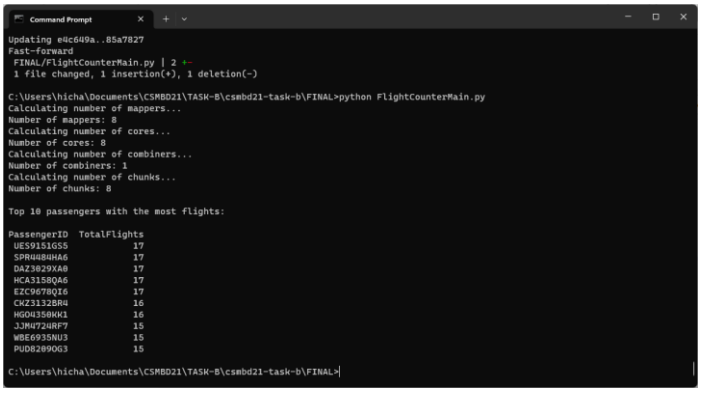

Figure 2. Output of the Flight Count programme