Implementing MapReduce Programming Framework using ZeroMQ sockets

CS403/534 - Distributed Systems
Programming Assignment (PA) #2, Fall 2023

November 17, 2022

Deadline: November 27, 23:55 (Turkish Time)

Abstract

In this assignment, you will implement a MapReduce framework using ZeroMQ socket comunication. Using this framework, you will develop an application that reads a text file containing paper citations, and computes the number of citations for each paper and checks whether there are cyclic citations.

1 Background

MapReduce is a programming framework that allows us to perform distributed and parallel processing on large data sets in a distributed environment. For this assignment, you will implement a toy MapReduce framework. For detailed information about map-reduce programming strategy, please see Lab 4 recording and materials.

For implementation, you are required to use ZeroMQ Pipeline Pattern with PYTHON programming language. For pipeline pattern, you might again visit the slides covered in Lab 3.

2 Description

The MapReduce framework contains a single abstract class, with the following functions:

```
abstract class MapReduce:
  def MapReduce(num_worker: int) # constructor
  abstract def Map(map_input):Partial Result
  abstract def Reduce(reduce_input):Result
  private def Producer(producer_input)
  private def Consumer(consumer_input)
  private def ResultCollector()
  def start(filename)
```

`MapReduce` is an abstract class meaning that it contains abstract methods (`Map` and `Reduce`), which are not implemented by this class but must be implemented by its subclasses.

The expected usage of a `MapReduce` object (which must be an instance of one of its subclasses since it is an abstract class) is as follows:

- The client generates an instance using the constructor by specifying the number of workers (mappers).
- Then, the client calls the `start` method. Rest of the computation is handled by `start`. Rest of the methods are not intended to be used by clients directly.
- The `start` method generates a `Producer` process, `Consumer` processes of which number is determined by the constructor's input and a `ResultCollector` process. All of these processes must be generated concurrently.
- The `start` method reads the file specified by its input and forms a list. Then, it passes this list to the `Producer` process while generating it. Then, it waits until all processes it generated finishes.
- The `Producer` process receives a list as its input and divides it into pieces with respect to the number of workers. Then, it pushes each piece into a ZMQ pipeline socket. In your implementation, you must ensure that pieces have almost equal sizes meaning that sizes of pieces can differ by at most one element.
- `Consumer` processes pull pieces from the ZMQ socket that the `Producer` pushes. In your implementation, you must guarantee that each `Consumer` gets exactly one piece. When the `Consumer` receives its piece, it processes this piece by calling the `Map` method and produces a partial result. When, the `Map` method returns, the `Consumer` must expect a dictionary as a return value. The returned dictionary (partial result) is pushed into another ZMQ pipeline socket to send it to the `ResultCollector` process.
- The `ResultCollector` process pulls partial results produced by `Consumer` processes, from the ZMQ socket. After collecting all the partial results, the `ResultCollector` calls `Reduce` method to obtain the final result. The **Reduce** method should expect a list of dictionaries as input and must return a single dictionary as the result. The output of `Reduce` method must be written to `results.txt` file before the `ResultCollector` returns.
- It should be noted that our implementation is a simplified version of the original MapReduce. The original MapReduce contains multiple *Reducers* that are responsible for a subspace of possible *Mapper* outputs. In our implementation, there is one *Reducer* which is the `ResultCollector` process.

While implementing the `MapReduce` abstract class, you should be careful about the following points:

- `Map` and `Reduce` are abstrac methods and must not be implemented in `MapReduce` class. However, each subclass of `MapReduce` must implement them.

- `Producer`, `Consumer` and `ResultCollector` are private methods implemented by `MapReduce` that are not accessible by clients but that are inherited by subclasses of `MapReduce`. They are similar to *protected* methods in Java.
- You need to utilize two ZMQ pipelines. For the first one, only the `Producer` pushes and `Consumers` pulls. For the second one, `Consumers` push and the `ResultCollector` pulls. Marshalling is done by transforming message objects into `json` format. You might use `send_json` and `recv_json` methods for this purpose.
- The `start` method must utilize Python's multiprocessing module for creating `Producer`, `Consumer` and `ResultCollector` processes[1].
- The `start` method takes a filename as its argument and forms the input data list for the `Producer` process. You might assume that the lines of the input file consists of tab separated integers. Each line of the file will be an element of the input list of the `Producer`.

3 Subclasses to Implement

In PA2, you are expected implement two subclasses of `MapReduce`: `FindCitations` and `FindCyclicReferences`. These classes must inherit `start`, `Produce`, `Consume` and `ResultCollector` methods of the `MapReduce` class and only implement `Map` and `Reduce` methods according to their descriptions presented below.

Both classes are expected to run on High-Energy Physics Citation Network data set[2] of Stanford Large Network Dataset Collection. Basically, this data set keeps a citation graph. Each line of the data file represents an edge. Each edge is represented by two integers separated by a tab. These integers are unique identifiers of papers and if the line is of the form $A\,B$, it means that the paper with ID A cites paper B.

The dataset contains 421578 edges which might be a little large for testing your implementations. Therefore, the PA3 package contains a Python script named `test_generator.py` that takes a subset of this dataset by uniformly sampling and the size of the sample set is determined by setting the variable *numLines*.

You need to implement `Map` and `Reduce` methods according to the following descriptions:

- `FindCitations` class calculates the number citations for each paper, i.e., it calculates the number of incoming edges for each node. When the `start` method terminates, `results.txt` should contain a dictionary of which keys are paper IDs and values are citation counts[3].

[1]See `https://docs.python.org/3/library/multiprocessing.html#multiprocessing.Process` for learning how to use Multiprocessing with ZMQ sockets. You can also check `zmq_with_processes.py` file provided in this package.

[2]See `http://snap.stanford.edu/data/cit-HepPh.html` for more information on this data set and to download the data file.

[3]see `result01.txt` in the package which is the output for `test01.txt`.

- `FindCyclicReferences` class extracts papers with cyclic references. If paper A gives reference to paper B and B gives reference to A, these references form a cycle. For this PA, we only consider cycles of length 2. For instance, if A cites B, B cites C and C cites A, this forms a cycle of length 3 and you do not need to find them. When the `start` method terminates, `results.txt` should contain a dictionary of which keys are pair of ordered node IDs (if key is (A, B) then $A \leq B$) and the value is always 1[4]. Note that while implementing your methods, you can assume that there are no duplicate lines i.e., the same edge is not repeated more than once.

4 Package Contents

The PA2 package contains the following files:

- `CS403-534_PA2.pdf`: This file.
- `zmq_with_processes.py`: Example Python script that shows how to use ZMQ sockets with Multiprocessing module.
- `Cit-HepPh.txt`: The original High-Energy Physics Citation Network dataset file.
- `test-generator.py`: Python script for generating data set of desired size from the High-Energy Pyhsics dataset.
- `FindMax.py`: An example subclass of `MapReduce` class that calculates the maximum among a sequence of integer IDs.
- `sample-01.txt`: A sample input for `FindMax` class to test it.
- `sample-01-flow.txt`: A sample run of the `FindMax`'s start method on `sample-01.txt`.
- `test01.txt` and `result01.txt`: A sample input and output for `FindCitations` class.
- `test02.txt` and `result02.txt`: A sample input and output for `FindCyclicReferences` class.

5 Program Flow

You should create a `main.py` with the following command line arguments[5]:

- The first parameter is either COUNT or CYCLE which switches between `FindCitations` and `FindCyclicReferences`.
- The second parameter is the number of workers. You should support as many as 10 of them.

[4] see `result02.txt` in the package which is the output for `test02.txt`.

[5] Please check `https://www.tutorialspoint.com/python/python_command_line_arguments.htm` for more information on Python command line arguments

- The third parameter is the name of the file that you will process.

In the following, we have two example invocations from the command line:

```
python main.py COUNT 4 test01.txt
python main.py CYCLE 3 test02.txt
```

6 Submission Guidelines

PA3 must be implemented in Python using ZMQ sockets. You need to submit following Python files:

- `main.py`: A Python script that can be called from command line with arguments as specified in Section 5.
- `MapReduce.py`, `FindCitations.py` and `FindCyclicReferences.py`: Python files that implement `MapReduce`, `FindCitations` and `FindCyclicReferences` classes respectively.

Do NOT submit the ".txt" files as they can be large in size.

Required files explained above should be put in a single zip file named as `CS_403-534_PA03_name_surname.zip` and submitted to PA3 under assignments in SUCourse+.

7 Grading Criteria

- Code Skeleton (10 pts): Correct class hierarchy, methods are implemented in right places and obey the visibility requirements.
- `start` method (15 pts): Input file is correctly processed and processes are created in the right way.
- `Producer` method (20 pts): Input list is (almost-)equally distributed to pieces and pushed smoothly to the ZMQ channels.
- `Consumer` method (10 pts): Each method instance gets a piece from the `Producer`, processes it and forwards to the `ResultCollector` using ZMQ sockets.
- `ResultCollector` method (5 pts): Collects all partial results and reflects the result to the output file.
- `FindCitations` (10 + 10 pts): `Map` and `Reduce` methods implement the specification described in Section 3.
- `FindCyclicReferences` (10 + 10 pts): `Map` and `Reduce` methods implement the specification described in Section 3.
- **BONUS** (10 pts): 5 pts bonus for each fastest subclass implementation.