**How it works**

1. Test script runs user process. User process get input file locations, mapper intermediate file folder, reducer file folder by calling udf.
2. User process creates master process.
3. Master process creates N mapper processes and pass the input file locations. Each mapper process will save the intermediate file on disk and send the file location to master process.
4. Master process check the state of mapper processes periodically until all mapper process finish.
5. Master process creates N reducer processes and pass the intermediate file locations. Each reducer process will save the reduced file on disk and send the file location to master process.
6. Master process check the state of reducer processes periodically until all reducer process finish.
7. Master process notify user process MapReduce finished.

**Design tradeoffs**:

1. I only use functions instead of classes because I think if a simple job can be done by functions, I don’t need to use classes which makes design more complicated. The only thing user need to do is write udf in specific format, then run the test script, and in the test script only need to call one API function. The simpler, the better.
2. I have to assume user already split the input files into chunks because if I just slice the file by bytes, the content could be broken. The paper said the user will call MapReduce API to split the input files into M chunks. Piazza has some posts talking about “slice the file by rows”. However, I want to point out that this statement has problem because it assumes the file content are rows and each row won’t exceed the memory limit. How do you know your file’s type, txt, pickle, json, csv? If the input file needs to load external modules to read? How do you know the input file content are rows? How about the input file has only one huge row that exceed your worker memory limit, a long array, a long string? Alternative approaches are user provide the split function in udf, or the MapReduce API only accept specific file types.
3. The reduce phase starts after all mappers finished. This will cause some performance drawdown because most mappers will have to wait for those minority fault mappers. I didn’t find a proper way to notify all other mappers if the master detects one mapper failed, so I designed in this way. Fortunately, the map phase could be finished very quick.

**How to run**

1. Put “udf.py”, “map\_reduce\_multi.py”, “test.py” in the same directory
2. In terminal, run “python3 test.py”

**map\_reduce\_multi.py**

There are 4 functions in this file.

**def run\_reducer(tuple reducer, func reduce\_function, event rd\_ready, list file\_location, str reduced\_folder, int ID)**:

1. reducer: a tuple of (IP, port), in this case (‘local host’, port)

reduce\_function: udf.

rd\_ready: a multiprocessing event.

file\_location: a list of intermediate file folders by mapper ID.

reduced\_folder: a directory that where the reduced file should write to.

ID: partition ID, each reducer works for a partition of keys.

1. This function will destroy itself with some probability to simulate fault.
2. Call reduce function, passes intermediate file folders and partition ID.
3. Get the reduced result, save it to disk.
4. Set the event to ready state. Listen on the given port, once master pings, notify master reduce finished.

**def run\_mapper(tuple mapper, func map\_function, event mp\_ready, str file\_location, str mapped\_folder, int N)**:

1. mapper: a tuple of (IP, port), in this case (‘local host’, port)

map\_funciont: udf.

mp\_ready: a multiprocessing event.

file\_location: input file location.

mapped\_folder: the directory where the mapper should save intermediate files.

N: number of partitions.

1. This function will destroy itself with some probability to simulate fault.
2. Call mapper function, passes input file location and number of partitions.
3. Get the mapped result where key is partition ID. Create N folders under mapped\_folder, and write the result to each folder by key.
4. Set the event to ready state. Listen on the given port, once master pings, notify master map finished.

**def run\_master(tuple master, list mapper, list reducer, tuple user, func run\_mapper, func run\_reducer, func map\_function, func reduce\_function, event ms\_ready, event us\_ready, int n\_worker)**:

1. master: a tuple of (IP, port), in this case (‘local host’, port)

mapper: a list of tuples, each tuple is (IP, port), in this case (‘local host’, port)

reducer: a list of tuples, each tuple is (IP, port), in this case (‘local host’, port)

user: a tuple of (IP, port), in this case (‘local host’, port)

run\_mapper: mapper process function.

run\_reducer: reducer process function.

map\_function: udf.

reduce\_function: udf.

ms\_ready: a multiprocess event.

us\_ready: a multiprocess event.

n\_worker: number of workers/partitions/input files.

1. Set the ms\_ready event to ready state. Listen on master port, get input file locations, mapper intermediate file folder, reducer file folder from user process.
2. Create N events for mappers and N events for reducers.
3. Create N mapper processes and pass the params.
4. Master process sleep for some time for mapping.
5. Periodically, for each mapper event, if the state is ready, ping the mapper and get mapped file locations. If the state is not ready, then the mapper will be marked as dead and respawn that mapper process.
6. Once all the mappers finished. Create N reducer processes and pass the params.
7. Master process sleep for some time for reducing.
8. Periodically, for each reducer event, if the state is ready, ping the reducer and get reduced file locations. If the state is not ready, then the reducer will be marked as dead and respawn that reducer process.
9. Once us\_ready event is set as ready, notify user process MapReduce finished.

**def map\_reduce\_multi(func file\_function, func map\_function, func reduce\_function)**:

1. This is the API function that test script calls. It takes the 3 udf as input.
2. Get the input file locations, mapper intermediate file folder, reducer file folder by calling file\_function.
3. Create master port, user port. Create N mapper ports, N reducer ports.
4. Create ms\_ready, us\_ready events.
5. Create master process and pass params.
6. Once ms\_ready event is set to ready, send the input file locations, mapper intermediate file folder, reducer file folder to master process.
7. Set the us\_ready event to ready state. Listen on user port, wait for finish information from master process.

**udf.py**

There are 3 functions user need to provide.

**def get\_inputs()**:

This function should return a tuple (**list** input file locations, **str** mapper output directory, **str** reducer output directory)

**def map\_function(str file, int N)**:

1. This function takes two params as input, the file location and number of partitions. Note the number of partitions is also equal to the number of workers.
2. Construct a dictionary and initialize with partition/worker IDs as keys and empty list as values.
3. User should load the file, perform custom map process. For each (k, v) pair mapped, user will store it in the dictionary by partition ID, i.e. a hash function that k % N.
4. Return the dictionary.

**def reduce\_function(list folders, int ID)**:

1. This function takes two params as input, a list of intermediate file locations, and partition ID. Note there are N folders (N workers) for intermediate files, and in each folder, there are also N folders (N partitions).
2. User should load the files in each folder by partition ID, combine them and perform custom reduce process. Then return the output.

**test.py**

1. import “udf.py” and “map\_reduce\_multi.py”
2. run “map\_reduce\_multi(get\_inputs, map\_funciton, reduce\_function)”