Project 1: MapReduce - Design Document

Introduction

This project implements a basic version of MapReduce in **C++** running on a single server. This library can be extended to implement user-defined map and reduce functions on similar data such that function definitions are as follows:

Map **(k1,v1)** → **list (k2,v2)**
Reduce **(k2, list(v2))** → **list(v2)**

User Defined Functions

1. **Inverted Index:** This application takes multiple document IDs and words in the document as an input. We want to build an inverted index out of this, i.e., we want to know the mapping of each word to all the documents it is present in. The initial input is a string with document IDs followed by its contents separated by '\t'. The map function will return a list of key-value pairs with word as key and document ID as value. The reduce function finally returns a list of key-value pairs with one unique word as key and all the document IDs where the word is seen as value.

2. **Word Count:** The goal of this application is to count the number of word occurences in the document. The input is a document ID and it's contents. The map function will return a list of key-value pairs, with each word as key and count "1" as value. The reduce function takes this list as input and returns another list of key-value pairs, with each unique word as key and their final count as value.

3. **K-mer Count:** In bioinformatics, k-mers are substrings of length k contained within a genome sequence containing nucleotides (A, C, T and G). In this application, we find all k-length substrings of a genome sequence and find the number of occurrences of each of these sequences. We have taken k=3 for this application.

Testing

The test script (*testfile.py*) written in **Python** compiles the code for MapReduce and user-defined functions, and compares the generated result files with previously-generated ground-truth values. Using this Python script, configuration files are generated to run each UDF. An iteration is done through all the UDFs that have been defined where they are compiled and run one after another. Each UDF is tested on the data stored in the corresponding input text file, e.g. UDF1 performs "inverted index" on the data in *inputfile1.txt*. The data present in these text files cover non-trivial test cases, e.g. multiple presence of the same word inside a document in word-count, across documents in inverted index, and words with leading and trailing punctuations in UDFs where white space, '**\t**' or '**\n**' is used as separators.

The test script also incorporates testing fault tolerance by killing worker processes based on an integer command-line argument. When this integer is 1, another python script (*kill_process.py*)

is executed concurrently which kills a worker process at random, once during the map phase and again during the reduce phase. Our MapReduce implementation then restarts the killed worker to handle the fault.

Description

1. **IO Arguments Handling:** The name of the input file, the file name with the code for a UDF class, number of worker processes (***N***), and the names of the output files are specified in a configuration text file. This configuration file is processed by the master. The master reads the corresponding input file, splits it into ***N*** partitions, invokes ***N*** workers for performing map and reduce. The final result is stored in ***N*** output files.

2. **Multiprocessing:** Once the master process is instantiated, it creates ***N*** child processes using *fork()* in C++ for each of the map and reduce phases. Each worker is instantiated with *inputfilename*, *outputfilename*, *UDF name* and the total number of partitions ***N***. The *mapper()* and *reducer()* functions inside the worker class in turn call the map and reduce functions of the specific UDF mentioned in the configuration file.

3. **Communication:**
   3.1. *Inter-Process Communication (IPC):* IPC has been implemented using pipes in C++. The master acts as a parent process that spawns ***N*** child processes. Each child process invokes a Worker object which runs as a mapper or a reducer.
   The mapper (child process) and the master (parent process) communicate over pipes where the mapper returns the names of the intermediate files it has created to the master. The parent process concatenates all the intermediate filenames to a string and passes it to every reducer. The reducer does a regular expression match to find the intermediate files it needs to read from.

   3.2. *Fault Tolerance:* Fault tolerance is achieved using the *waitpid()* function in C++. The master creates a list of *{Process ID, Mapper/Reducer Number}* pairs. It then checks to see if each of these mapper processes have been executed to completion using *waitpid(Process ID, &status, 0)*. If the process has exited normally, a value of 0 is stored in the status variable, otherwise a value greater than 0 is stored in status. *WIFEXITED(status)* returns to a non-zero value if status was returned for a child process that terminated normally.
   Once, we know which mapper or reducer has unexpectedly terminated, a call to *handle_dead_mapper()* or *handle_dead_reducer()* is made. This function creates a new worker process that restarts the specific mapper or reducer that got killed.
   For testing, the mapper/reducer process is being killed using *kill_process.py*. *kill_process.py* finds all the processes running with the name *mapreduce.exe*. The process with the lowest process ID in this list is the parent process, all others are child processes. We randomly select a child process and kill it. We parallely run two instances of *kill_process.py* from *testfile.py*. One instance, which starts at a delay of 5 seconds, kills a mapper process. Another, which starts at a delay of 17 seconds, kills a reducer process.

4. Worker Execution

4.1. *Single Mapper:* The initial input to the mapper is a string read from a partition of a text file. The mapper processes this string using delimiters like '**\t**', '**\n**', or white-space as specified in the UDF to extract the required key-value pairs. Each mapper writes the generated list of key-value pairs in ***N*** intermediate text files. A key-value pair is written in an intermediate file based on a hash function applied on the key. Each key-value pair is written in each line of the chosen intermediate file. The key and value in a line is separated by '**\t**'. Each of these ***N*** intermediate files is then read by a worker to execute the reduce function. The workers return the locations of the generated intermediate files to the master process.

4.2. *Single Reducer:* A worker executing the reduce function reads ***N*** intermediate files generated by the ***N*** mappers. The worker gets the locations of the ***N*** intermediate files as input from the master process. A list of key-value pairs is generated by processing the file contents using '**\n**' and '**\t**' as delimiters. The list is sorted according to the ascending order of the keys and values of the same keys are added/concatenated to generate the final list. The worker then writes this list in an output file. The desired name and location of the output file is received by the worker from the master.

5. Partitioning

Every input file is partitioned into ***N*** files such that every file has roughly ***(Number of lines / N)*** lines. Each mapper processes a partition of the input file and generates key-value pairs based on the specified User Defined Function. The key-value pairs are written to one of ***N*** intermediate files based on the hash value of the key. This ensures that the key-value pairs with the same keys are written to the same partition in the intermediate files. The reducer reads ***N*** intermediate files such that each file has keys with the same hash values. The below table depicts the ***NxN*** intermediate files generated during each mapping phase where every mapper writes to ***N*** files and every reducer reads from ***N*** files.

	Reducer 1	Reducer 2	...	...	...	Reducer N
Mapper 1	hash(k) = 1	hash(k) = 2	...	...	...	hash(k) = N
Mapper 2	hash(k) = 1	hash(k) = 2	...	...	...	hash(k) = N
...	...	...	...	...	...	...
...	...	...	...	...	...	...
Mapper N	hash(k) = 1	hash(k) = 2	...	...	...	hash(k) = N

6. Hash Function

The hash function is used to determine which intermediate file a key-value pair from the mapper is written to, based on the hash of the key. We have used the following calculation of hash for a string s :

$$\mathbf{hash(s) = (s[0] + s[1].p + s[2].p^2 ... + s[n\text{-}1].p^{n-1}) \bmod m}$$

where

$\mathbf{s[i]}$ denotes the $\mathbf{i}$th character of the string

$\mathbf{n}$ is the length of the string

$\mathbf{p}$ is a prime number roughly equal to the size of the alphabet (We have taken $\mathbf{p}$ = 31)

$\mathbf{m}$ is the number of partitions/buckets, here $\mathbf{m} = \boldsymbol{N}$

Design trade-offs

1. Delays have been added to the execution of mappers and reducers so that *kill_process.py* has enough time to find the child processes that are running and kill them. A delay has been added between the map and the reduce phase as well so that the two instances of *kill_process.py* do not kill two mapper or two reducer processes.
2. A process that has been unexpectedly terminated can only be restarted once all the other processes have been completed.
3. The map and reduce functions of a particular UDF are called using a switch statement. The condition on which the switch statement runs is the UDF value entered in the configuration file. A switch case had to be used in this scenario because, as per our research, there is no equivalent to Java's reflection in C++.
4. An integer value of 1 is being passed as the key to the mapper instead of null. This value is not used anywhere in our code.

Execution

To run the system for one application, please follow these steps:

1. Update *config.txt* with the correct input filename, output filename, number of mappers/reducers (***N***) and UDF number. For the input files provided in the repository, please use the following mapping:
    a. *inputFile1.txt* is for UDF1
    b. *inputFile2.txt* is for UDF2
    c. *inputFile2.txt* is for UDF3
2. Run the following commands:
    a. g++ -o mapreduce.exe src/master_fault_tolerance.cpp src/worker.cpp src/UDF1.cpp src/UDF2.cpp src/UDF3.cpp
    b. ./mapreduce.exe

For automated testing, run *testfile.py. testfile.py* takes a command line argument of 0 or 1 to test the system with or without fault tolerance, respectively. To run *testfile.py*, follow these steps:

1. *pip install psutil*
2. *python testfile.py 0* : To test the system without fault tolerance
3. *python testfile.py 1* : To test the system with fault tolerance