The Edward S. Rogers Sr. Department
of Electrical & Computer Engineering
UNIVERSITY OF TORONTO

COMPUTER SYSTEM PROGRAMMING
ECE454 – FALL 2023
ASSIGNMENT-5

Dues are defined at the end of this guideline.

Objectives

- Understanding/Implementing Map-Reduce Process on Big-Data resources
- Understanding/Implementing Parallel mechanism on Map-Reduce Process

5 - Implementing Map-Reduce Process on Database

In the context of Database Management Systems (DBMS), many problems can be processed using the Map-Reduce solution pattern, and in a bigger scale this process can simplify the Big-Data analysis greatly. The Map-Reduce process consists of three major tasks in general: [Total 30 Marks]

- Map: Extract something of interest from each record in database
- Shuffle: Group the intermediate results in a specific structure
- Reduce: Applying further process such as summarize /transform on each group of data to find the final result

More specifically, in a Map-reduced process:

- Each record of data should have the structure of a collection of `<key, value>` pairs
- The function Map( ), specified as below, transforms all initial <key, value> pairs to produce the intermediate <key, value> pairs

  map(K_1, V_1) → list (K_2, V_2)
- The function Shuffle( ), transforms the intermediate <key, value> pairs of data from the previous phase (Map), sorts, analyzes and classifies them to be transferred to the next phase (Reduce)
- The function Reduce( ), specified as below, transforms all intermediate values associated with the same key to produce a list of values (normally one for the key)

  reduce(K_2, list(V_2)) → list (V_3)

In this section, you will simulate a Map-Reduce process on a pre-defined database of your own. Your program to simulate the Map-Reduce process, should be implemented (using C++) based on the following requirements:

a) [9 Marks]
Expected input to the process: Create a **sample** of a huge data file with millions of lines of English Text (just have a **sample** of it). This text contains the English words relevant to your own database about the old and new (recent) movies of any type. This database should be created in advance of any phase in your program and can contain relevant data about the old and new movies of any type (max 5 movie names). Assuming that data file is distributed on different machines (max 4 machines) physically located at different points (4 points). A Map-Reduce process is used to collect the expected outputs for all the movies (#5) from this database (at 4 points).

Change the format of your input database (distributed at 4 points) to be stored in a hash-based-array (here you need 4 different arrays, and so it is recommended to divide your data file in to 4different parts from the beginning or having 4 different data files). Each element in the array can contain a header node pointing to a linked-list. Each node in that linked-list contains 3 fields: the key (movie name), some data about that key (movie info), and pointer to the next node. The number of the nodes in each linked-list determines the number of the key appeared in its data file. For example if the linked-list for the key “movie1” has 6 nodes, this means that “movie1” has appeared in its relevant data file 6 times (every time along with a different definition). The hash-function to address/find each element in the array is based on the input key and your own design.

So at this point, you should have 4 hash-based-array containing the data from 4 different data files (locations).

Expected output from the process: a list of (word, count) pairs, where the count is the number of occurrences of each word in the whole database (stored on different 4 machines), produced by applying a Map-Reduce process on the input database.

b) Implement the Map-Reduce process on the whole database above (4 data files which are now available in 4 hash-based array), with major functions implemented below. Display the results at each phase. [9 Marks]

```
map( word, data-file(i)) → list of <word, count>
```

Given the words in data-file(i), find the list of pairs `<word, count>` from data-file(i) (can be found from the hash-array already implemented in part a, relevant to data-file(i)) **Note -** word can be any word in data-file (i)

```
reduce (word, list of <word, count>) → (word, total-count)
```

Given the list of pairs `<word, count>`, find the output (w, s) where s is sum of counts for any word

Note-Each run of your program, is only about one input key to be processed in your database, and for this program only work on the word = "movie name".
Note-Feel free to use any number of other variables and functions in your implementation.

c) Measure the execution time for the whole-process (only one key-word) in your program. Find the average of the execution time in 5 different runs, and save all data in a table. [2 Mark]

d) After part (c), optimize your code by applying various techniques you have learned so far (no compilation optimization options, but just optimization in the code). Find the execution time for the whole program again like in (c), then compare the result with the previous result in (c). Discuss on this comparison. [5 Marks]

e) Using the first version of your program (code completed in b) and mostly with respect to the Map-Reduce process, which parts of your implemented process can be run in parallel? In case you find any parts, use the multi-threading technique to change your implementation in (b) to execute those parts in parallel, calculate the execution time for the whole program like in (c) and then compare the results properly.
What do you suggest as another parallelism technique to be applied in this part? [5 Marks]

Deliverables & Submission for Assignment-5 [Total 30 Marks = weight 5%]
-The teams should submit their final report through the submission folder on Quercus and then present their work in the labs (on the same day) to be evaluated by the TAs through Q/A by Thursday Nov30, at 8:59am [30 Marks]
-Your final report should include: all the source codes with proper comments (in plain text), all the results' tables with proper definition and discussion (in word or pdf), a screen-shot of each result.
-Make sure to add the names and student numbers of all Team members, and the Lab section your team attends at the top part of your final report. Only ONE final submission is required from each team.
-The teams should save all their answers in a final report and submit it under this naming rule: "PRAxx_Tyy_Az.txt" where "xx" is the Lab section (01 for 12-3pm, and 02 for 9am-12pm), "yy is the Team number and "z" is the Assignment number, e.g. "PRA01_T06_A3.txt" is the name for submission from Team06 for A3 working in the Lab on Thursday 12-3pm.

Good Luck.
Useful Resources:
https://www.javatpoint.com/mapreduce
https://www.talend.com/resources/what-is-mapreduce/
https://www.ibm.com/topics/mapreduce
https://thispointer.com/c11-multithreading-part-2-joining-and-detaching-threads/
https://www.geeksforgeeks.org/multithreading-in-cpp
https://stackoverflow.com/questions/

In order to simplify the implementation of the Map-Reduce algorithm in A5, you can follow the instructions below:

Input-phase:
-Get input text files from various locations (#4): Input1, Input2, Input3, Input4

Split-phase:
-Split Input1 to lines: line1, line2, line3, ..., lineN
-Split Input2 to lines: line1, line2, line3, ..., lineM
...
-Split Input4 to lines: line1, line2, line3, ..., lineY

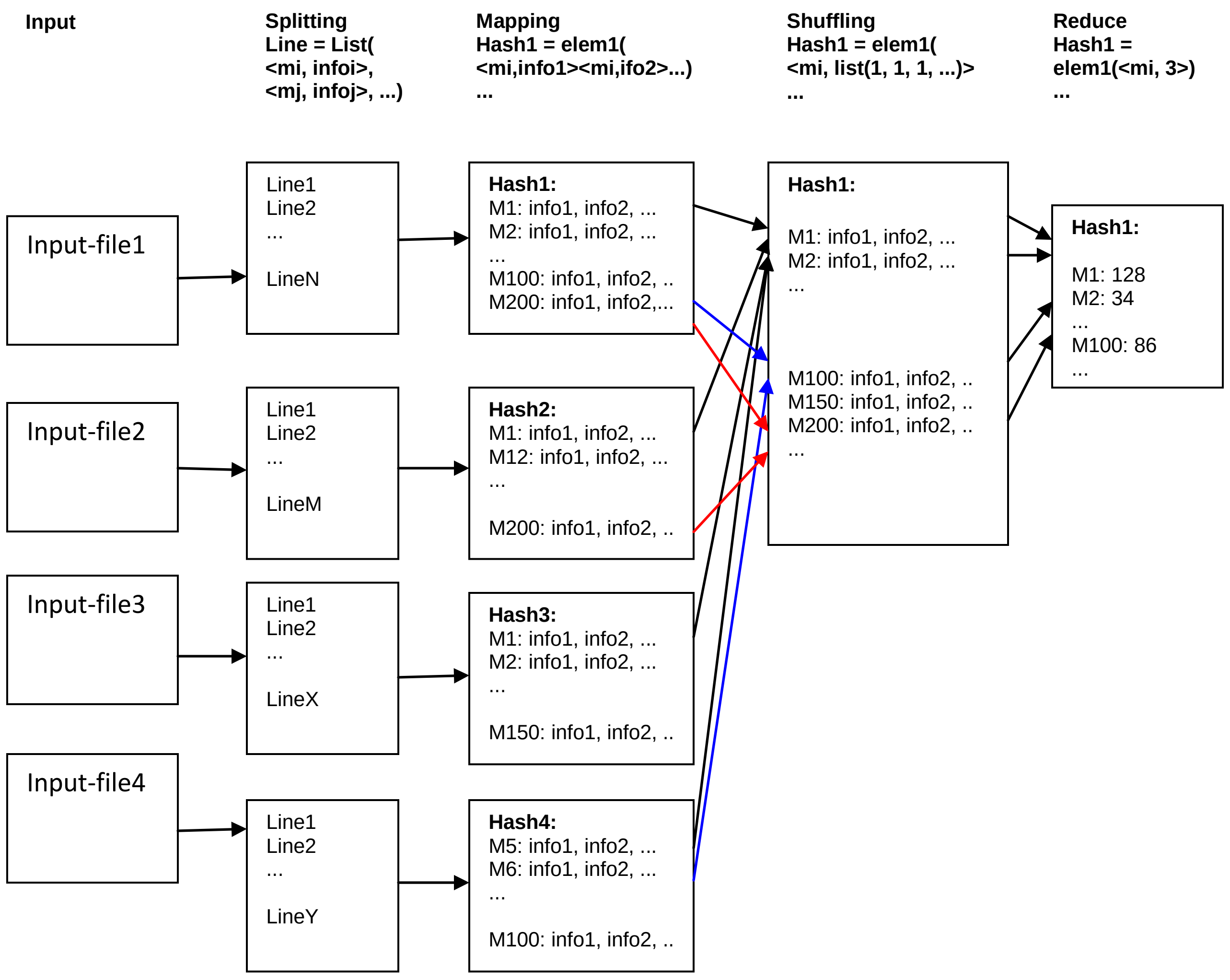
Mapping-phase:
a) Insert Input1-line1 into Hash1, based on <"movie-name", "movie-info">
Extract the data from Input1-Line1 and insert the movie info to the Hash1 table at different elements, for example: if "movie-name1" is found in the Line1, then its corresponding "movie-info" will be added to the Hash1 table and if there are more than one occurrence for "movie-name1" their "movie-info" will be added to that Hash1 element as a linked-list (at the key "movie-name1"), and this will be repeated for all other lines in the Input1

-Insert Input1-line2 into Hash1, based on <"movie-name", "movie-info">
...
-Insert Input1-lineN into Hash1, based on <"movie-name", "movie-info">

b) The same as step (a) for all other Inputs 2-4 to create hash tables Hash2 to Hash4
Note- At this phase you might add other information like an Index to each node to save in its linked-list, for example every time a new info is found in Input1 for "movie-name1", it can be added to its linked-list (key with "movie-name1") with two fields as a pair containing the <"movie-info", "movie-Index">, where Index can be its counter OR you might want to just have one single data "movie-info" (not a pair)

Shuffling phase:
At this phase all 4 Hash tables you have created so far should be merged together as one Hash table; Select one of them as the major Hash like Hash1, and merge the other tables with this major Hash1:
Go through all elements in the other Hash2 to Hash4 and add them to the corresponding elements in Hash1(for the same key or movie name) in their corresponding linked-list
Note-At this phase the nodes structure can still contain just one single data or a pair of data

Reducing phase:
a) At this phase for the major Hash1 table, go with each element, counts its nodes in its corresponding linked-list (or if you already had the Index as the counter, use the last Index in the list showing the last count of that key)
b) Create a new Hash table, where each element with the key "movie-name" contains the result of counting that key in the Hash1 as the pair of <"movie-name", "count">
   Note-Instead of creating a new Hash table, you might add this result node for each element either at the end or at the beginning of its relevant linked-list in the Hash1 table again


Input
Splitting
Line = List(
<mi, infoi>,
<mj, infoj>, ...)
Mapping
Hash1 = elem1(
<mi,info1><mi,ifo2>...)
...
Shuffling
Hash1 = elem1(
<mi, list(1, 1, 1, ...)>
...
Reduce
Hash1 =
elem1(<mi, 3>)
...
Input-file1
Input-file2
Input-file3
Input-file4
Line1
Line2
...
LineN
Line1
Line2
...
LineM
Line1
Line2
...
LineX
Line1
Line2
...
LineY
Hash1:
M1: info1, info2, ...
M2: info1, info2, ...
...
M100: info1, info2, ..
M200: info1, info2,...
Hash2:
M1: info1, info2, ...
M12: info1, info2, ...
...
M200: info1, info2, ..
Hash3:
M1: info1, info2, ...
M2: info1, info2, ...
...
M150: info1, info2, ..
Hash4:
M5: info1, info2, ...
M6: info1, info2, ...
...
M100: info1, info2, ..
Hash1:
M1: info1, info2, ...
M2: info1, info2, ...
...
M100: info1, info2, ..
M150: info1, info2, ..
M200: info1, info2, ..
...
Hash1:
M1: 128
M2: 34
...
M100: 86
...