Questions from Chapter 2 of textbook: Mining of Massive Datasets

**Aparna Pavithran**

# 2.3 Algorithms using MapReduce

Write pseudo-code for mapper and reducer for the following algorithms:  
\* No explanations needed, just write the pseudo-code \*

**a. Matrix-Vector multiplication**

**pseudo-code:**

map(String key, String value):

// key: contains i co-ordinate.

// value: contains mij element.

int val // this contains vj vector value. Assume it is there in memory

EmitIntermediate(key, AsString(ParseInt(value)\*val));

reduce(String key, Iterator values):

// key: a URL

// values: a list of multiplied values

int result = 0;

for each v in values:

result += v;

Emit(key,result);

**b. Relational Operators:**

**- Selection**

**pseudo-code:**

map(String key, String value):

// key: contains a tuple.

// value: contains a tuple.

//key and value are same

if(key satisfies condition C)

EmitIntermediate(key, value);

reduce(String key, String value):

// key: a tuple which passed the condition

// value: a tuple which passed the condition

Emit(key,value);

**- Projection**

**pseudo-code:**

map(String key, String value):

// key: contains a tuple.

// value: contains a tuple.

//key and value are same

String S; // contains columns needed in projection

String k = contents of key after eliminating columns in S.

EmitIntermediate(k, k);

reduce(String key, Iterator value):

// key: a tuple which passed the condition

// value: same tuples a number of times

Emit(key,key);

**- Union**

**pseudo-code:**

map(String key, String value):

// key: contains a tuple.

// value: contains a tuple.

//key and value are same

EmitIntermediate(key, key);

reduce(String key, Iterator values):

// key: a tuple which passed the condition

// values: same tuples a number of times

Emit(key,key);

**- Intersection**

**pseudo-code:**

map(String key, String value):

// key: contains a tuple.

// value: contains a tuple.

//key and value are same

EmitIntermediate(key, key);

reduce(String key, Iterator values):

// key: a tuple

// values: values from different relations

if (values contains multiple columns from different relations) then

Emit(key,key);

**- Difference**

**pseudo-code:**

map(String key, String value):

// key: contains a tuple.

// value: contains a name of relation.

EmitIntermediate(key, value);

reduce(String key, Iterator values):

// key: a tuple

// values: list of relation names

if values contain ‘R’ and does not contain ‘S’ then // name as relation R but as S

Emit(key,key);

**- Natural Join**

**pseudo-code:**

map(String key, String value):

// key: contains a tuple.

// value: contains a tuple.

//key and value are same

//B is the joining column of the relation

EmitIntermediate(value.B, value);

reduce(String key, Iterator values):

// key: a tuple which passed the condition

// values: a tuple which passed the condition

for each r in values

for each s in values

if r.relationname = ‘R’ and s.relationname = ‘S’: //or it can be any condition given

Emit(r.col1,r.col2,r.col3,s.col1,s.col2); //it will change depending on the columns in relation

**- Grouping and aggregation**

**pseudo-code:**

map(String key, String values):

// key: contains a tuple.

// value: contains a tuple.

//key and value are same

EmitIntermediate(key.col, key);

reduce(String key, Iterator value):

// key: a tuple which passed the condition

// value: a tuple which passed the condition

int sum = 0;

for each r in values:

sum += r.col

Emit(key,sum);

// it can be changed to any aggregation operators.

**c. Matrix Multiplication**

**pseudo-code:**

map(String key, String value):

// key: name of the matrix (Ie M or N) and row and column number

// value: element of the matrix.

String mtx = substring(key,0,1) //extract matrix name from key

Int col = ParseInt(substring(key,1,1)) //extract column number from key

Int row = ParseInt(substring(key,1,1)) //extract row number from key

EmitIntermediate(col, (mtx,row,value));

reduce(String key, Iterator values):

// key: column number

// values: list of (mtx,row,value)

for each v in values

String m = extract matrix name from v

If (m = ‘M’) then

int row = extract row value from v ie from (mtx,row,value) pair

int val1 = extract mij part ie value part from v

for each k in values

String n = extract matrix name from k

If (n = ‘N’) then

int col = extract row value from k ie from (mtx,row,value) pair

int val2 = extract nij part ie value part from k

Emit((row,col),val1\*val2);

// now perform another grouping and aggregation operation.

**pseudo-code:**

map(String key, String value):

// key: contains a (row,col) number pair

// value: contains a multiplied value

//key and value are same

EmitIntermediate(key, value);

reduce(String key, Iterator value):

// key: contains a (row,col) number pair

// value: contains a multiplied value

int sum = 0;

for each r in values:

sum += value

Emit(key,sum);

# 2.5 Communication Cost Model

**a. What is meant by the communication cost of a task and communication cost of an algorithm? Why is the communication cost more important than the execution cost of an algorithm in MR?**

\* If you want to understand more about this, you can read this paper:  
http://dl.acm.org/citation.cfm?id=2331053&dl=ACM&coll=DL&CFID=670610572&CFTOKEN=25724374 \*

**Answer:**

The communication cost of a task is the size of the input to the task. This size can be measured in bytes. The communication cost of an algorithm is the sum of the communication cost of all the tasks implementing that algorithm. In particular, we do not consider the amount of time it takes each task to execute when estimating the running time of an algorithm.

We can explain and justify the importance of communication cost as follows.

* The algorithm executed by each task tends to be very simple, often linear in the size of its input.
* The typical interconnect speed for a computing cluster is one gigabit per second. That may seem like a lot, but it is slow compared with the speed at which a processor executes instructions. Moreover, in many cluster architectures, there is competition for interconnect when several compute nodes need to communicate at the same time. As a result, the compute node can do a lot of work on a received input element in the time it takes to deliver that element.
* Even if a task executes at a compute node that has a copy of the chunk(s) on which the task operates, that chunk normally will be stored on disk, and the time taken to move the data into main memory may exceed the time needed to operate on the data once it is available in memory.

Assuming that communication cost is the dominant cost, we might still ask why we count only input size, and not output size. The answer to this question involves two points:

1. If the output of one task τ is input to another task, then the size of τ ’s output will be accounted for when measuring the input size for the receiving task. Thus, there is no reason to count the size of any output except for those tasks whose output forms the result of the entire algorithm.
2. But in practice, the algorithm output is rarely large compared with the input or the intermediate data produced by the algorithm. The reason is that massive outputs cannot be used unless they are summarized or aggregated in some way.

**b. What is meant by wall-clock time? If we assign all the work to just one task i.e. one node, there would be very little communication cost, so why is this approach not taken?**

**Answer:**

Wall-clock time is the time it takes a parallel algorithm to finish. Using careless reasoning, one could minimize total communication cost by assigning all the work to one task, and thereby minimize total communication. However, the wall-clock time of such an algorithm would be quite high. The algorithms we suggest, or have suggested so far, have the property that the work is divided fairly among the tasks. Therefore, the wall-clock time would be approximately as small as it could be, given the number of compute nodes available.

# **2.6 Complexity Theory for MapReduce**

**a. Define the following:**

**Answer:**

**- Reducer Size (q) :-** This parameter is the upper bound on the number of values that are allowed to appear in the list associated with a single key. Reducer size can be selected with at least two goals in mind.

1. By making the reducer size small, we can force there to be many reducers, i.e., many different keys according to which the problem input is divided by the Map tasks. If we also create many Reduce tasks – even one for each reducer – then there will be a high degree of parallelism, and we can look forward to a low wall-clock time.

2. We can choose a reducer size sufficiently small that we are certain the computation associated with a single reducer can be executed entirely in the main memory of the compute node where its Reduce task is located. Regardless of the computation done by the reducers, the running time will be greatly reduced if we can avoid having to move data repeatedly between main memory and disk.

**- Replication Rate (r) :-** Can define r to be the number of key-value pairs produced by all the Map tasks on all the inputs, divided by the number of inputs. That is, the replication rate is the average communication from Map tasks to Reduce tasks (measured by counting key-value pairs) per input.

**b. What are the advantages of a small value of q?**

**Answer:**

We can choose a reducer size sufficiently small that we are certain the computation associated with a single reducer can be executed entirely in the main memory of the compute node where its Reduce task is located. Regardless of the computation done by the reducers, the running time will be greatly reduced if we can avoid having to move data repeatedly between main memory and disk.

**c. How is the tradeoff between r and q expressed?**

**Answer:**

Consider the one-pass matrix-multiplication algorithm. Suppose that all the matrices involved are n X n matrices. Then the replication rate r is equal to n. That fact is easy to see, since for each element mij, there are n key-value pairs produced; these have all keys of the form (i, k), for 1 ≤ k ≤ n. Likewise, for each element of the other matrix, say njk, we produce n key-value pairs, each having one of the keys (i, k), for 1 ≤ i ≤ n. In this case, not only is n the average number of key-value pairs produced for an input element, but also each input produces exactly this number of pairs.

We also see that q, the required reducer size, is 2n. That is, for each key (i, k), there are n key-value pairs representing elements mij of the first matrix and another n key-value pairs derived from the elements njk of the second matrix. While this pair of values represents only one particular algorithm for one-pass matrix multiplication, we shall see that it is part of a spectrum of algorithms, and in fact represents an extreme point, where q is as small as can be, and r is at its maximum. More generally, there is a tradeoff between r and q, that can be expressed as qr≥2n^2.

**d. Read the other examples presented and try to understand them. No written answer required.**