CS61C Summer 2016 Discussion 12 – WSC, Map Reduce & Spark

1 Warehouse Scale Computing

1. **Amdahl's Law**
   You are going to train the image classifier with 50,000 images on a WSC having more than 50,000 servers. You notice that 99% of the execution can be parallelized. What is the speedup?

   1 / (0.01 + 0.99 / 50,000) $\approx$ 1 / 0.01 = 100

2. **Performance of a WSC**

	Local	Rack	Array
DRAM latency (μs)	0.1	100	300
Global hit rate	90%	9%	1%
DRAM bandwidth (MiB/sec) (μs)	20,000	100	10
Disk bandwidth (MiB/ sec)	200	100	10

   1) Calculate the AMAT of this WSC. What is vital for WSC performance?

      AMAT = 0.9 x 0.1 + 0.09 x 100 + 0.01 x 300 = 0.09 + 9 + 3 = 12.09 μs
      Locality of access within a server is vital for WSC performance

   2) How long does it take to transfer 1,000 MiB a) between disks within the server, and b) between DRAM within the rack? What can you conclude from this example?

      a) 1,000 / 200 = 5 sec
      b) 1,000 / 100 = 10 sec.
      Data transfer outside a single server is detrimental to WSC performance. Network switches are the bottlenecks

2 Map Reduce

Use pseudocode to write MapReduce functions necessary to solve the problems below. Also, make sure to fill out the correct data types. Some tips:

- The input to each MapReduce job is given by the signature of the **map()** function.
- The function **emit(key k, value v)** outputs the key-value pair **(k, v)**.
- The **for(var in list)** syntax can be used to iterate through **Iterables** or you can call the **hasNext()** and **next()** functions.
- Usable data types: **int, float, String**. You may also use lists and custom data types composed of the aforementioned types.
- The method **intersection(list1, list2)** returns a list that is the intersection of list1 and list2.

1. Given the student's name and the course taken, output each student's name and total GPA.
   Declare any custom data types here:

```
CourseData:
    int courseID
    float studentGrade  // a number from 0-4
```

map(String student, CourseData value):	reduce(String key, Iterable<float> values):

`emit(student, value.studentGrade)`	`totalPts = 0`
	`totalClasses = 0`
	`for (grade in values):`
	`  totalPts += grade`
	`  totalClasses++`
	`emit(key, totalPts / totalClasses)`

2. Given a person's unique int ID and a list of the IDs of their friends, compute the list of mutual friends between each pair of friends in a social network.
   Declare any custom data types here:

```
FriendPair:
    int friendOne
    int friendTwo
```

map(int personID, list<int> friendIDs):	reduce(FriendPair key, Iterable<list<int>> values):

`for (fID in friendIDs):`
`  if (personID < fID):`
`  friendPair = (personID, fID)`
`  else:`
`  friendPair = (fID, personID)`
`  emit(friendPair, friendIDs)`
	`mutualFriends =`
	`  intersection(values.next(), values.next())`
	`emit(key, mutualFriends)`

3. Given a set of coins and each coin's owner, compute the number of coins of each denomination that a person has.
   Declare any custom data types here:

```
CoinPair:
    String person
    String coinType
```

map(String person, String coinType):	reduce(CoinPair key, Iterable< int > values):

`key = (person, coinType)`
`emit(key, 1)`
	`total = 0`
	`for (count in values):`
	`  total += count`
	`  emit(key, total)`

4. Using the output of the first MapReduce, compute the amount of money each person has. The function `valueOfCoin(String coinType)` returns a float corresponding to the dollar value of the coin. `Declare any custom data types here:`

```
CoinPair:
    String person
    String coinType
```

map(CoinPair key, int amount):	reduce(String key, Iterable< float > values):

`emit(coinPair.person, valueOfCoin(coinPair.coinType)*amount)`	`total = 0`
	`for (amount in values):`
	`  total += amount`
	`  emit(key, total)`

3 Spark

Useful terminology and functions:

- Resilient Distributed Datasets (RDD): primary abstraction of a distributed collection of items
- Transforms: RDD → RDD

map(*func*)	Return a new RDD by passing each element of the source through a function *func*.
flatmap(*func*)	Similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item).
reduceByKey(*func*)	When called on a dataset of (K,V) pairs, returns a dataset of (K,V) pairs where the values for each key are aggregated using the given reduce function *func*, which must be of type (V,V) ⇒ V.
groupByKey(*func*)	When called on a dataset of (K,V) pairs, returns a dataset of (K,V) pairs where V is a list of values that were grouped together because they were mapped to the same key.

- Actions: RDD → Value

reduce(*func*)	Aggregate the elements of the dataset **regardless of keys** using a function *func*

1) Implement Problem 1 of MapReduce with Spark

```
# students: list((studentName, courseData))
studentsData = sc.parallelize(students)
out = studentsData.map(lambda (k, v): (k, (v.studentGrade, __1__)))


                    .reduceByKey(lambda v1, v2: (v1[0] + v2[0], v1[1] + v2[1]))
                    .map(lambda (k, v): (k, v[0] / v[1]))
```

2) Implement Problem 2 of MapReduce with Spark

```
def genFriendPairAndValue(pID, fIDs):
    return [((pID, fID), fIDs) if pID < fID else (fID, pID) for fID in fIDs]
def intersection(l1, l2):
    return [x for x in b1 if x in b2]
# persons: list((personID, list(friendID))
personsData = sc.parallelize(persons)


out = personsData.flatMap(lambda (k, v): genFriendPairAndValue(k, v))
                 .reduceByKey(lambda v1, v2: intersection(v1, v2))
```

3) Implement Problem 3 of MapReduce with Spark

```
# coinPairs: list((person, coinType))
coinData = sc.parallelize(coinPairs)


#(3.a) out: list(((person, coinType), count))
out1 = coinData.map(lambda (k1, k2): ((k1, k2), 1))
               .reduceByKey(lambda v1, v2: v1 + v2)
#(3.b)
out2 = out1.map(lambda (k, v): (k[0], v * valueOfCoin(k[1])))
           .reduceByKey(lambda v1, v2: v1 + v2)
```