**Warehouse Scale Computing**

1. **Amdahl’s Law:**
   
   1) You are going to train an image classifier on a training set of 50,000 images using a WSC of more than 50,000 servers. You notice that 99% of the execution can be parallelized. What is the speedup?

2. **Failure in a WSC**
   
   1) In this example, a WSC has 55,000 servers, and each server has four disks whose annual failure rate is 4%. How many disks will fail per hour?

   2) What is the availability of the system if it does not tolerate the failure? Assume that the time to repair a disk is 30 minutes.

3. **Power Usage Effectiveness (PUE) = (Total Building Power) / (IT Equipment Power)**

   Sources speculate Google has over 1 million servers. Assume each of the 1 million servers draw an average of 200W, the PUE is 1.5, and that Google pays an average of 6 cents per kilowatt-hour for datacenter electricity.

   1) Estimate Google’s annual power bill for its datacenters.

   2) Google reduced the PUE of a 50,000 machine datacenter from 1.5 to 1.25 without decreasing the power supplied to the servers. What’s the cost savings per year?
**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):**

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):**
3. a) 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:**

<table>
<thead>
<tr>
<th>CoinPair:</th>
</tr>
</thead>
<tbody>
<tr>
<td>String person</td>
</tr>
<tr>
<td>String coinType</td>
</tr>
</tbody>
</table>

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

b) 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.

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

• RDD: primary abstraction of a distributed collection of items

• Transforms: RDD → RDD

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>map(func)</td>
<td>Return a new distributed dataset formed by passing each element of the source through a function <code>func</code>.</td>
</tr>
<tr>
<td>flatMap(func)</td>
<td>Similar to map, but each input item can be mapped to 0 or more output items (so <code>func</code> should return a Seq rather than a single item).</td>
</tr>
<tr>
<td>reduceByKey(func)</td>
<td>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 <code>func</code>, which must be of type <code>(V,V) =&gt; V</code>.</td>
</tr>
</tbody>
</table>

• Actions: RDD → Value

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>reduce(func)</td>
<td>Aggregate the elements of the dataset <em>regardless of keys</em> using a function <code>func</code>.</td>
</tr>
</tbody>
</table>

1. Implement Problem 1 of MapReduce with Spark

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

2. Implement Problem 2 of MapReduce with Spark

```python
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)
```

3. Implement Problem 3 of MapReduce with Spark

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