# Jon 101 Dis 7

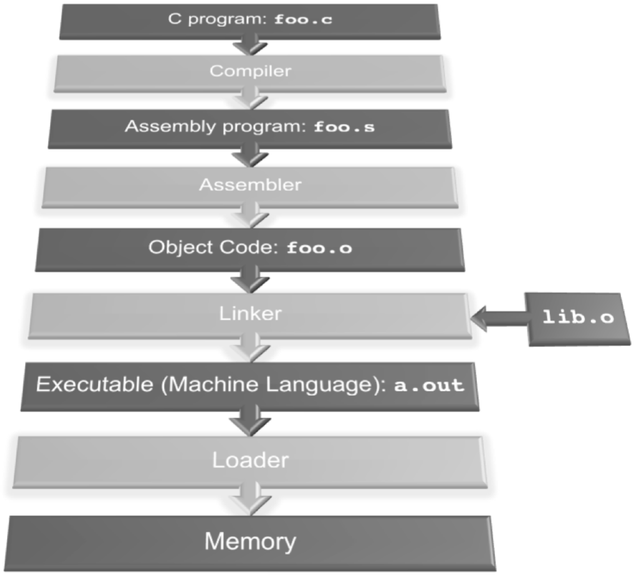
Wednesday, October 10, 2018 2:06 PM

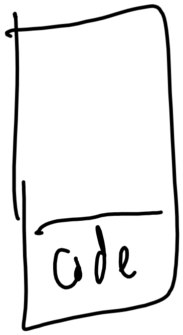
CS 61C Fall 2018

# CALL, WSC, MapReduce, Spark

Discussion 7: October 8, 2018

## 1 Compile, Assemble, Link, Load, and Go!





1.1 What is the Stored Program concept and what does it enable us to do?

Inst. are just data as well

⇒ manipulate program. code w/ other program

→ j foo:

1.2 How many passes through the code does the Assembler have to make? Why?

2 passes : 1 find label ~~for~~ forward refs

2 Convert to machine code, resolve forward refs

foo :

1.3 What are the different parts of the object files output by the Assembler?

Relocation Table

1.4 Which step in CALL resolves relative addressing? Absolute addressing?

Assembler Linker

1.5 What does RISC stand for? How is this related to pseudoinstructions?

input map → (k1, v1) (k2, v2) (k1, v3) ⋮ --> sort by key → (k1, [v1, v3]) (k2, [v2]) → reduce → (k1, v1+v3) (k2, v2)

## 2 MapReduce

For each problem below, write pseudocode to complete the implementations. Tips:

- The input to each MapReduce job is given by the signature of `map()`.
- `emit(key k, value v)` outputs the key-value pair `(k, v)`.
- `for var in list` 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.
- `intersection(list1, list2)` returns a list of the intersection of `list1`, `list2`.

2.1 Given a set of coins and each coin's owner, compute the number of coins of each denomination that a person has.

input: ((person, type), [1 1 1 1])

Declare any custom data types here:

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

```
map(String person, String type):
    key = (person, type)
    value = 1
    emit(key, value)
```

```
reduce(CoinPair key, Iterable<int> values):
    total = 0
    for val in values:
        total += val
    emit(key, total)
```

CoinPair, #of the

(person, [1 2 3])

2.2 Using the output of the first MapReduce, compute each person's amount of money. `valueOfCoin(String coinType)` returns a float corresponding to the dollar value of the coin.

```
map(CoinPair key, int total):
    outKey = key.person
    value = valueOfCoin(key.coinType)
            • total
    emit(outKey, value)
```

```
reduce(String per, Iterable<float> values):
    Same code
    for val in values:
        total += val
    emit(per, total)
```

( person, Σ values)

## 3 Spark

**Resilient Distributed Datasets (RDD)** are the primary abstraction of a distributed collection of items

**Transforms** *RDD → RDD*

`map(f)` Return a new dataset formed by calling *f* on each source element. returns item returns list ↓

words → ["a", "b", "c"]

1 input → multiple output

**flatMap**($f$) Similar to map, but each input item can be mapped to 0 or more output items (so $f$ should return a sequence rather than a single item).

→ join together after all is mapped

**reduceByKey**($f$) 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 $f$, which must be of type $(V, V) \to V$.

**Actions** *RDD → Value*

many → 1

**reduce**($f$) Aggregate the elements of the dataset *regardless of keys* using a function $f$.

Call `sc.parallelize(data)` to parallelize a Python collection, `data`.

(person, type)

↓

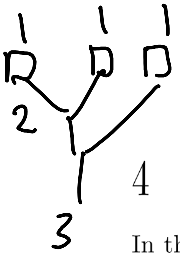
3.1 Given a set of coins and each coin's owner, compute the number of coins of each denomination that a person has. Then, using the output of the first result, compute each person's amount of money. Assume `valueOfCoin(coinType)` is defined and returns the dollar value of the coin.

The type of `coinPairs` is a list of (person, coinType) pairs.

```
coinData = sc.parallelize(coinPairs)
```

```
out1 = coinData.map(helper)
               .reduceByKey(help2)
out2 = out1.map(help3)
           .reduce ByKey (help2)
```

```
helper (tup):
    return (tup, 1)
help2 (v1, v2)
    return v1+v2
help3 ( tup):
    return (tup[0][0]
valueof(tup[0][1])·tup[1] )
tup = ((per, type), tot)
```



## 4 Amdahl's Law

In the programs we write, there are sections of code that are naturally able to be sped up. However, there are likely sections that just can't be optimized any further to maintain correctness. In the end, the overall program speedup is the number that matters, and we can determine this using Amdahl's Law:

$$\text{True Speedup} = \frac{1}{S + \frac{1-S}{P}}$$

where $S$ is the Non-sped-up part and $P$ is the speedup factor.

$$\frac{1}{S = \frac{1-S}{P}}$$

4.1 You write code that will search for the phrases "Hello Sean", "Hello Jon", "Hello Dan", "Hello Man", "Bora is the Best!" in text files. With some analysis, you determine you can speed up 40% of the execution by a factor of 2 when parallelizing your code. What is the true speedup?

4.2 You are going to run your project 1 feature analyzer on a set of 100,000 images using a WSC of more than 55,000 servers. You notice that 99% of the execution of your project code can be parallelized on these servers. What is the speedup?

# 5 Warehouse-Scale Computing

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.

5.1 Estimate Google's annual power bill for its datacenters.

5.2 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?

# 6 MapReduce/Spark Practice: Optimize Your GPA

6.1 Given the student's name and course taken, output their name and total GPA.

Declare any custom data types here:

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

```
map(________________, ________________):
```

```
reduce(________________, ________________):
```

6.2 Solve the problem above using Spark.

The type of `students` is a list of (studentName, courseData) pairs.

```
studentsData = sc.parallelize(students)
out = studentsData.map(lambda (k, v): (k, (v.studentGrade, _______)))
```

# 7 MapReduce/Spark Practice: Optimize the Friend Zone

7.1 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(________________, ________________):
```

```
reduce(________________, ________________):
```

7.2 Solve the problem above using Spark.

The type of `persons` is a list of (personID, list(friendID) pairs.

```
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]

personsData = sc.parallelize(persons)
```