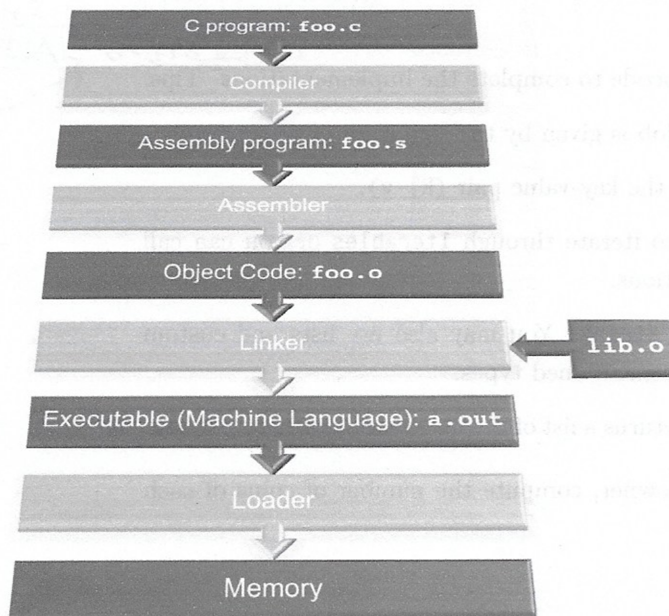
CS 61C Fall 2018

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?

instructions = Data

It is the idea that instructions are just the same as data, and we can treat them as such. This enables us to write programs that can manipulate other programs!

Programs can modify other programs.

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

Two, one to find all the label addresses and another to convert all instructions while resolving any forward references using the collected label addresses.

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

6 total Items.

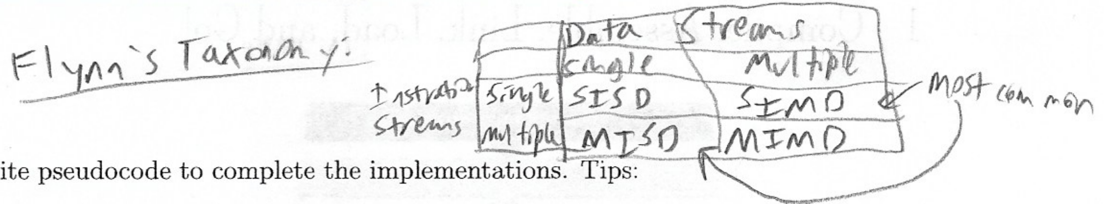
- Header: Size and position of other parts
- Text: The machine code
- Data: Binary representation of any data in the source file
- Relocation Table: Identifies lines of code that need to be "handled" by Linker
- Symbol Table: List of the files labels and data that can be referenced
- Debugging Information: Additional information for debuggers

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?

Reduced Instruction Set Computing. Minimal set of instructions leads to many lines of code. Pseudoinstructions are more complex instructions intended to make assembly programming easier for the coder. These are converted to TAL by the assembler.



2 MapReduce

MIMD

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.

Declare any custom data types here:

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

Collect all the pairs together + apply your fn.

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

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

We want to keep track of the number of what coin a person has

We put 1 to add them all up.

Take the emitted key pairs + combine (reduce) them.

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

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

adds up all the ones we set

since key = (person, coinType)

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.

```
1 map(_______________, _______________):
```

```
1 reduce(_______________, _______________):
```

```
map(CoinPair key, int amount):
    emit(coinPair.person,
        valueOfCoin(coinPair.coinType) * amount)
```

```
reduce(String key, Iterable<float> values):
    total = 0
    for amount in values:
        total += amount
    emit(key, total)
```

3 Spark

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

Transforms $RDD \rightarrow RDD$

- `map`(f) Return a new dataset formed by calling f on each source element.
- `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).
- `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 $(V, V) \rightarrow V$.

Actions $RDD \rightarrow Value$

- `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`.

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.

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

```
out1 = coinData.map(lambda (k1, k2): ((k1, k2), 1))
            .reduceByKey(lambda v1, v2: v1 + v2)

out2 = out1.map(lambda (k, v): (k[0], v * valueOfCoin(k[1])))
        .reduceByKey(lambda v1, v2: v1 + v2)
```

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.

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?

$$\frac{1}{0.6 + \frac{0.4}{2}} = \frac{1}{0.8} = 1.25$$

S = 60% = 0.6
P = 2

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?

$$\frac{1}{0.01 + \frac{0.99}{55000}} \approx \frac{1}{0.01} = 100$$

P = 55000
S = 0.01

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.

PUE · # of servers · KW/server · Electric cost.

$1.5 \cdot 10^6$ servers · 0.2kW/server · \$0.06/kW-hr · 8760 hrs/yr ≈ \$157.68 M/year

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?

$\text{PUE} = \frac{\text{Total building power}}{\text{IT equipment power}} \implies Savings \propto (PUE_{old} - PUE_{new}) * \text{IT equipment power}$

$(1.5 - 1.25) \cdot 50000$ servers · 0.2kW/server · \$0.06/kW-hr · 8760hrs/yr ≈ \$1.314 M/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
```

```
1 map(________________, ________________):

map(String student, CourseData value):
    emit(student, value.studentGrade)
```

we want to emit the student + grade for easy summing.

```
1 reduce(________________, ________________):

reduce(String key, Iterable<float> values):
    totalPts = 0
    totalClasses = 0
    for grade in values:
        totalPts += grade
        totalClasses++
    emit(key, totalPts / totalClasses)
```

gpa = total pts / totalclasses

so

6.2 Solve the problem above using Spark.

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

for total pts

for totalclasses ctr

```
1 studentsData = sc.parallelize(students)
2 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]))
```

add to total classes

adds gpa together.

final divide.

notice the different order. ORDER MATTERS!!!

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
```

```
1 map(________________, ________________):

map(int personID, list<int> friendIDs):
    for fID in friendIDs:
    if (personID < fID):
        friendPair = (personID, fID)
    else:
        friendPair = (fID, personID)
    emit(friendPair, friendIDs)
```

```
1 reduce(________________, ________________):

reduce(FriendPair key,Iterable<list<int>> values):
    mutualFriends = intersection(
        values.next(), values.next()
    )
    emit(key, mutualFriends)
```

7.2 Solve the problem above using Spark.

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

```
1 def genFriendPairAndValue(pID, fIDs):
2     return [((pID, fID), fIDs) if pID < fID else (fID, pID) for fID in fIDs]
3
4 def intersection(l1, l2):
5     return [x for x in b1 if x in b2]
6
7 personsData = sc.parallelize(persons)

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