CS4400: Database Systems
Parallel performance and algorithms

Brandon Myers
12/6/2016
Analyze speedup over sequential using BSP

```python
sc = SparkContext()

a = (sc.wholeTextFiles('hdfs:/user/bdmyers/sci.space'))

print (a.flatMapValues(lambda contents: contents.split(' ')).
    .filter(lambda x: x[1] in interesting_words).
    .distinct().
    .map(lambda x: (x[1], x[0])).
    .countByKey())
```

Assume:
- \( P \) processors
- \( F \) files in sci.space/
- files range from \( w \) words to \( W \) words
- selectivity of the filter is 0.01
- \( |\text{interesting\_words}| = C \)

Time for read+filter per word = \( T_{sw} \)
Time to distinct per tuple = \( T_{sd} \)
Time to swap key/value per tuple = \( T_{sk} \)
Time to count by key per tuple = \( T_{sc} \)
If we take a random walk through the graph, PageRank(B) is how often we are at node B.
Iterative PageRank algorithm

\[ PR(p_i; 0) = \frac{1}{N}. \] initially start equal PRs

\[ PR(p_i; t + 1) = \frac{1 - d}{N} + d \sum_{p_j \in M(p_i)} \frac{PR(p_j; t)}{L(p_j)} \]

N=number of pages
PR(p_i, t)= pagerank of p_i at iteration t
d = damping factor (see below)
M(p_i) = neighbors of p_i
L(p_j) = number of neighbors of p_j

\( PR(p_i; t+1) \) Needs two terms because during the random walk

- (1-d) probability that we decide to randomly jump anywhere in the graph
- d probability that we uniformly randomly follow one of the outgoing links

\[ \text{so, first term means probability that the walker jumps to } p_i \]

\[ \text{the second term means the probability that the walker was at one of } p_i \text{'s neighbors and then followed the edge to } p_i \]
PageRank in Spark

```scala
// SPARK
val links = spark.textFile(..).map(..).persist()
var ranks = // RDD of (URL, 1/n) pairs
for (k <- 1 to ITERATIONS) {
  // Build RDD of (targetURL, float) pairs
  // with contributions sent by each page
  val contribs = links.join(ranks).flatMap {
    (url, (links, rank)) =>
      links.map(dest => (dest, rank/links.size))
  }
  // Sum contributions by URL and get new ranks
  ranks = contribs.reduceByKey((x,y) => x+y)
  .mapValues(sum => a/n + (1-a)*sum)
}
```

basically, do this in a loop
a fixed number of times

join every page with its links

for every page, aggregate over its links’ contributions
Parallel hash join

pages
(a,1) (b,2) (c,3)
(d,4) (e,5)
(f,6) (g,7)

hyperlinks
(a,b) (e,a)
(e,d) (c,e)
(a,d) (b,g)
Administrivia

• ACE – online course reviews
Hash Aggregation, in BSP

1. for each tuple t, send t to worker with id = hash(t.grouping_key)
2. locally aggregate

for the analysis, assume the values of t.grouping_key is distributed as a power law
Hash Aggregation with local combining, in BSP

1. locally aggregate
2. for each tuple $t$, send $t$ to worker with $id = \text{hash}(t\text{.grouping_key})$
3. locally aggregate