# EE 451 Introduction to Parallel and Distributed Computation

Discussion 02/26/2021

University of Southern California

# Map-Reduce Background

- Large set of data needs to be processed in a fast and efficient way
- In order to process large set of data in a reasonable amount time, this needs to be distributed across thousands of machines
- Programmers need to focus on solving problems without worrying about the implementation

# MapReduce

- Programming Abstraction
- Two operations
  - Map
  - Reduce

# MapReduce

- Map
  - Input : key-value pairs
  - Output: intermediate key-value pairs

- MapReduce framework groups all pairs with same key

- Reduce
  - Input: key, iterator values (list of values)
  - Output: list with results

# MapReduce Example

- Counting each word in a large set of documents

```
map (String key, String value)
    // key: document name
    // value: document contents
    for each word w in value:
        EmitIntermediate (w, "1")
```

```
reduce ( String key, Iterator values)
    // key: word
    // value: a list of counts
    for each v in values:
        result + = ParseInt(v);
    Emit(AsString(result));
```

# MapReduce Example

- Counting each word in a large set of documents

| Document_1 | Document_2 |
|---|---|
| foo | test |
| bar | foo |
| baz | baz |
| foo | bar |
| bar | foo |
| test | |

Expected results:

<foo, 4>,<bar, 3>,<baz,2>,<test,2>

# MapReduce Example

- Counting each word in a large set of documents

```
map(String key, String value):
    // key: document name
    // value: document contents
    for each word w in value:
        EmitIntermediate(w, "1");
```

```
Map(document_1,contents(document_1))
<foo, "1">
<bar,"1">
<baz, "1" >
<foo, "1">
<bar, "1">
<test, "1">
```

```
Map(document_2,contents(document_2))
<test, "1">
<foo, "1">
<baz, "1">
<bar, "1">
<foo, "1">
```

# MapReduce Example

- Counting each word in a large set of documents

```
reduce(String key, Iterator values):
    // key: a word
    // values: a list of counts
    int result = 0;
    for each v in values:
        result += ParseInt(v);
    Emit(AsString(result));
```

```
Reduce(word, values)
<foo, “2”>
<bar,”2”>
<baz, “1” >
<test,”1”>
```

```
Reduce(word, values)
<test, “1”>
<foo, “2”>
<baz, ”1”>
<bar, ”1”>
```

# MapReduce Example

- Counting each word in a large set of documents

<foo, “2”>

<bar, ”2”>

<baz, “1”>

<test,”1”>

<test, “1”>

<foo, “2”>

<baz, ”1”>

<bar, ”1”>

Reduce(word, values)

<foo, “4”>

<bar, ”3”>

<baz, “2”>

<test,”2”>

Expected results:

<foo, 4>,<bar, 3>,<baz,2>,<test,2>

# Benefit of MapReduce

- Easy to use for programmers that do not need to worry about the details of distributed computing

- Flexible and scalable in large clusters of machines

# Apache Spark (1)

- Open source cluster computing framework
- Provides an interface for programming entire cluster with implicit data parallelism and fault tolerance

# Apache Spark (2)

- Spark Components
  - Core: distributed task dispatching, scheduling and basic I/O functionalities
  - Spark SQL: provides support for structured and semi-structured data
  - Spark streaming: provides support for streaming analytics
  - Mlib: Distributed machine learning framework
  - GraphX: Distributed graph processing framework

# Resilient Distributed Datasets (RDD) (1)

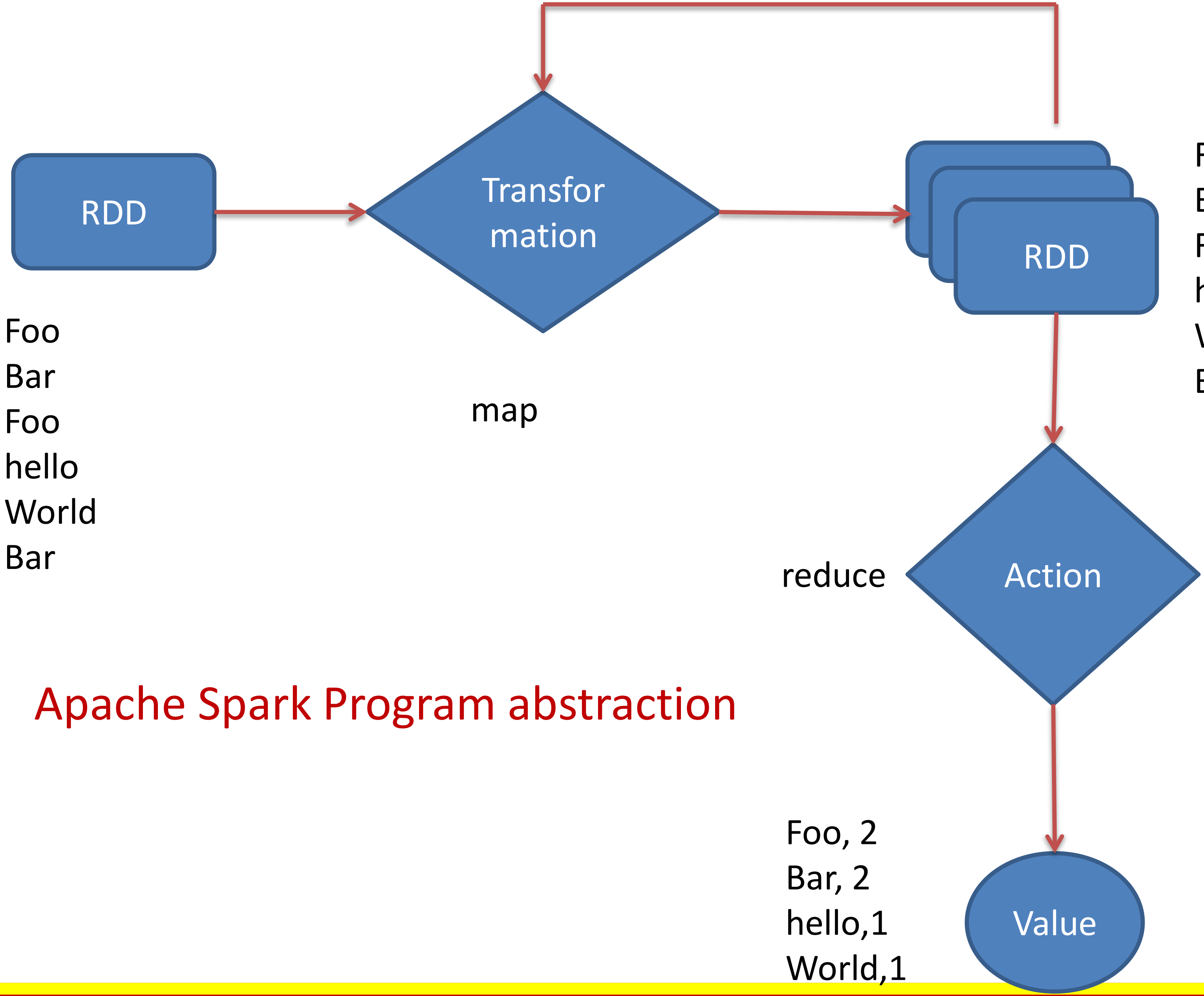
- Fault tolerant read-only collection of datasets that can be operated in parallel
- Creating RDDs
  - Transformation from an existing RDD
  - Referencing an external dataset (filesystem, HDFS)
- Operating on RDD
  - Transformation: creates new dataset from existing ones
  - Action: returns a value after running a computation on a dataset

# Resilient Distributed Datasets (RDD) (2)

- Transformations:
  - Map: each element passed through a function.
  - Union: union of elements in source RDD
  - Intersection: intersection of elements in source RDD
  - Filter: elements which match a criteria in source RDD
- Actions
  - Reduce: aggregate elements of RDD using a function
  - Collect: create an array out of RDD
  - Count : count the number of elements in RDD

# Resilient Distributed Datasets (RDD) (3)

- Transformations are lazy. Applied only when an action requires its results
- Transformation is recomputed each time an action is run on it. The results can be persisted in memory using functions such as: **cache() or persist()**




# Apache Spark Program abstraction

# Running a job

Submit to spark cluster

Cluster is the local machine

- ../bin/spark-submit  --master local[*] kmeans.py data.txt centroid.txt

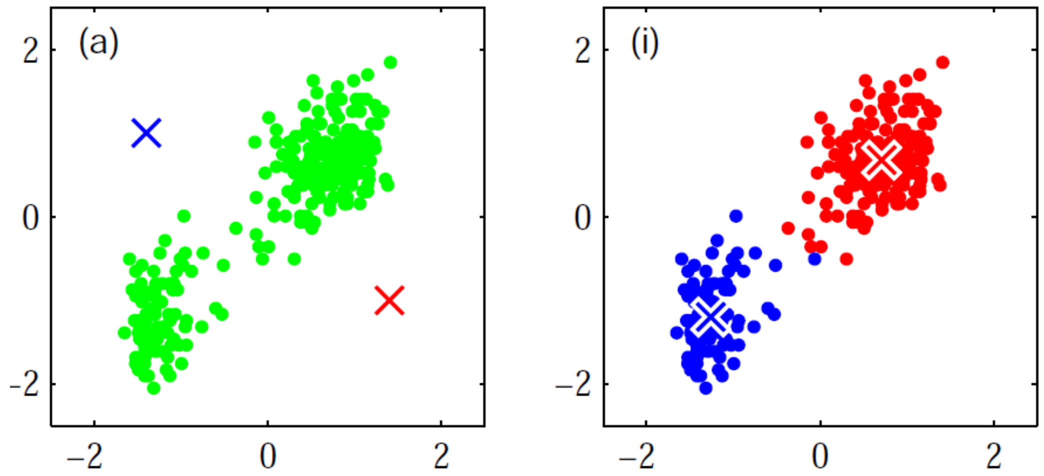
Python file

arguments

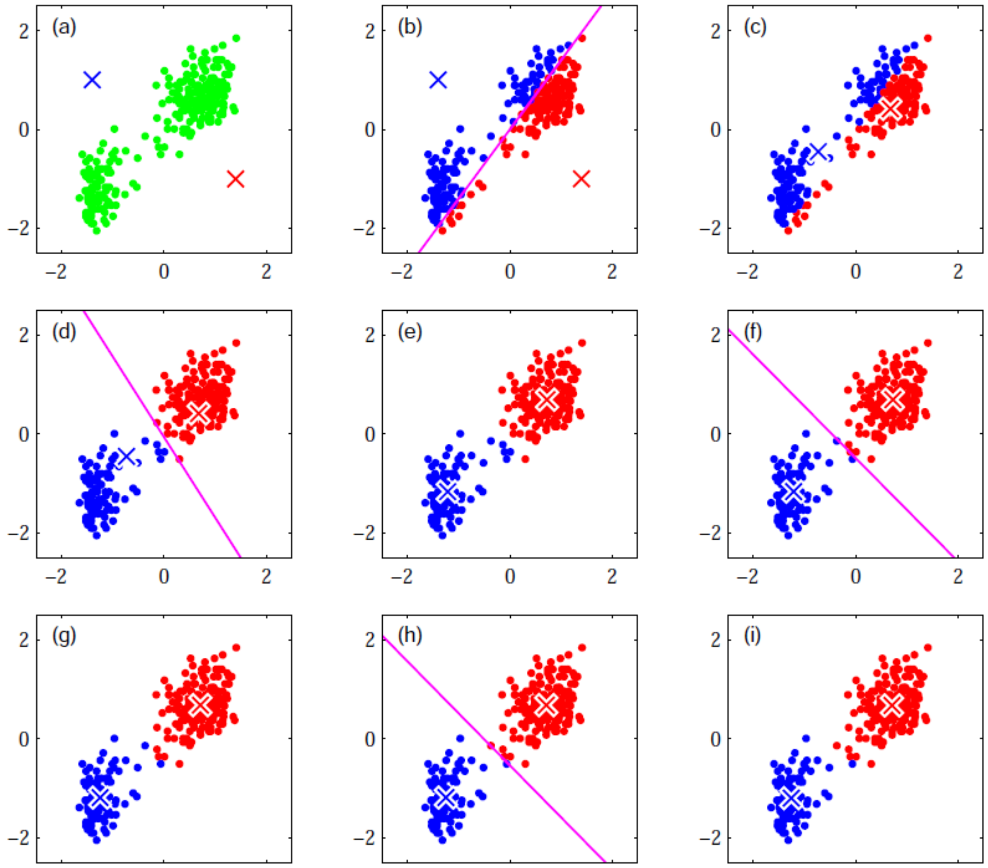
# PHW #5

# K-means Clustering (1)

- Input: a set of observations (values) $X$={$x_0, \ldots, x_{N-1}$}
- Objective: partition observations into $K$ clusters
  - Each cluster has a mean value, $\mu_i$ $(0 < i < K)$
  - Each observation belongs to the cluster with the closest mean



# K-means Clustering (2)



# K-means Clustering (3)

- **Map** $(x, \mu_0, \ldots \mu_{K-1})$

  Distance = ∞

  For $j = 0$ to $K - 1$ do

  If Distance < $|x \quad - \mu_j|$

  Distance = $|x \quad - \mu_j|$

  Cluster ID $key \quad$ = $j$

  End if

  End for

  Key

  Output key-value pair ($key \quad , x \quad$ )

  End for

  Value

# K-means Clustering (4)

- **Shuffle:** $L_i \leftarrow x \mid (i, x)$
- **Reduce** $(L_0, \dots L_{K-1})$

  For $i = 0$ to $K - 1$ do

  $\mu_i$ = average of elements in $L_i$

  End for

- **ReduceByKey**(L)

  return average of element in L

# K-means Clustering (5)

- Example
  - $X$ ={12, 10, 20, 34, 38, 40}
  - $K = 2$ ; Initially $\mu_0 = 10$, $\mu_1 = 20$

- Map($X$, $\mu_0$, $\mu_1$) $\rightarrow$ (0, 12), (0, 10),(1,20),(1,34),(1,38),(1,40)

- Shuffle( ) $\rightarrow$ $L_0$ =(0; 12, 10), $L_1$ =(1; 20, 34, 38, 40) (automatically done by the spark framework)

- Reduce($L_0, L_1$) $\rightarrow$ $\mu_0 = 11$, $\mu_1 = 32$ (Using ReduceByKey operation)

- Map($X$, $\mu_0$, $\mu_1$) $\rightarrow$ (0, 12), (0, 10),(0,20),(1,34),(1,38),(1,40)

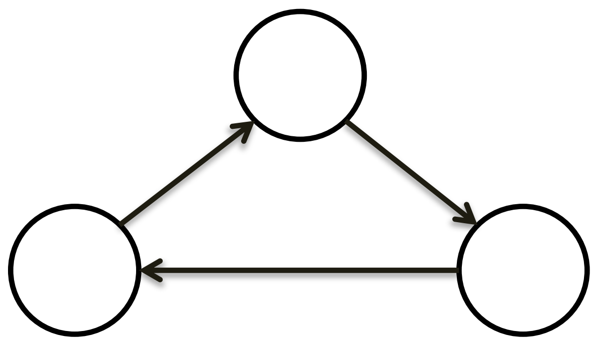
- …

# Template for k-means program

```python
import sys
from pyspark import SparkContext

def mapToCluster(data, means):
    #data -> a single integer value.
    #means -> list of the mean values.
    #return the mean value to which this data point belongs to
    return 0.0

def updatemeans(data1, data2):
    #data1,data2 -> tuple of format (meanvalue, count)
    #give (avg1, n1), (avg2, n2), new average will be (n1*avg1 + n2*avg2)/(n1+n2)
    return (newavg,newcount)

if __name__ == "__main__":
    if len(sys.argv) != 3:
        print(str(len(sys.argv))+"Usage: kmeans <datafile> <initialmeanfile>")
        exit(-1)

    #Create a sparkcontext
    sc = SparkContext(appName="kmeans")
    #load data from the text file
    data = sc.textFile(sys.argv[1]).cache()
    #load initial mean values from the text file
    means = sc.textFile(sys.argv[2])
    #We cannot directory use RDD. It should first be converted into a list to be iterated upon.
    meansList = means.collect()

    #we will run 50 iterations for calculating k means.
    numiter = 50

    for i in range(numiter):
        #For each data point create a tuple of the format (meanvalue,(datapoint, 1))
        clustermap = data.map(lambda p: (mapToCluster(p,meansList),(p,1)))
        #Use reduce operation to calculate new mean value for all the datapoint belonging to the same key
        newmeans = clustermap.reduceByKey(updatemeans)
        #Create a list from the RDD
        meansTupleList = newmeans.collect()
        meansList = []
        for mi in meansTupleList:
            meansList.append(mi[1][0])

    finalclustermap = data.map(lambda p: (mapToCluster(p,meansList),p)).sortByKey()
    finalclustermap.saveAsTextFile("output");
```

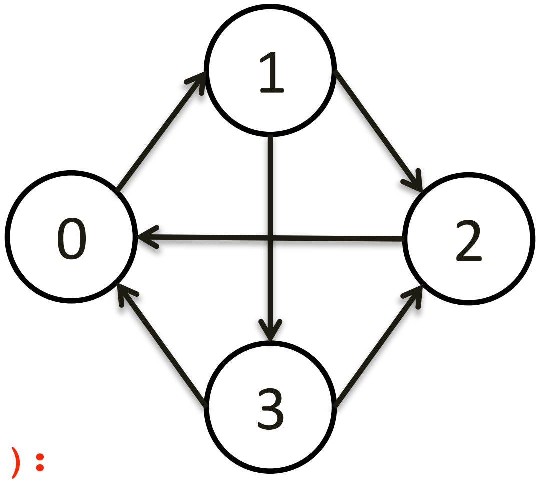
# Triangle Counting (1)

- Input: directed graph
- Output: how many graph triangles each vertex belongs to



# Triangle Counting (2)

- Map
  - $X$: $\{x_1, x_2, \dots, x_N\}$ be a list relating vertex $X$ with vertices $x_1, x_2 \dots x_n$
  - Produce key-value pairs $(k, v)$
    - $k$: neighbour of X
    - $v$: $x_i$
    - E.g. 3:3 -> map -> (0,3), (2,3)
    - Number of key-value pairs produced for X: $|d^+(X)| \times N$, where $d^+$ is the out-degree of $X$

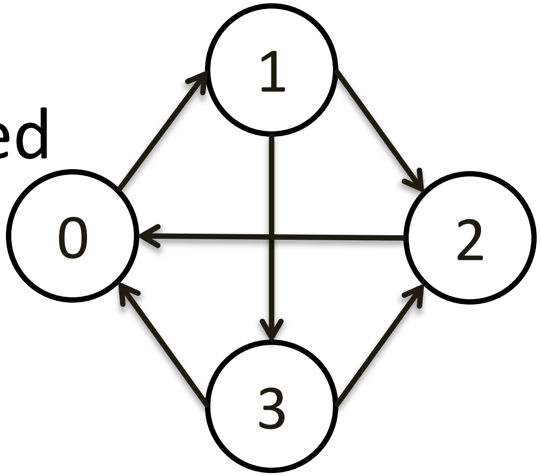



```
Produced
pairs(*,3):
(0,3) (2,3)
```

# Triangle Counting (3)

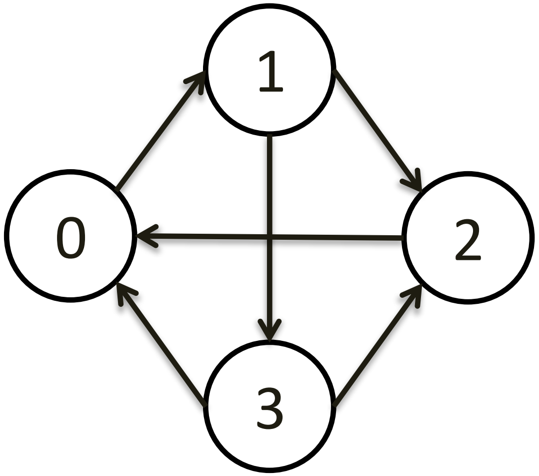
- Reduce
  - Collect all the key–value pairs produced in the previous Map step and produce $L_k = (k; v_1, v_2, \ldots)$
  - Eg.
    - $L_0 = (0; 2,3)$
    - $L_1 = (1; 0)$
    - $L_2 = (2; 1,3)$

    L3=….



# Triangle Counting (4)

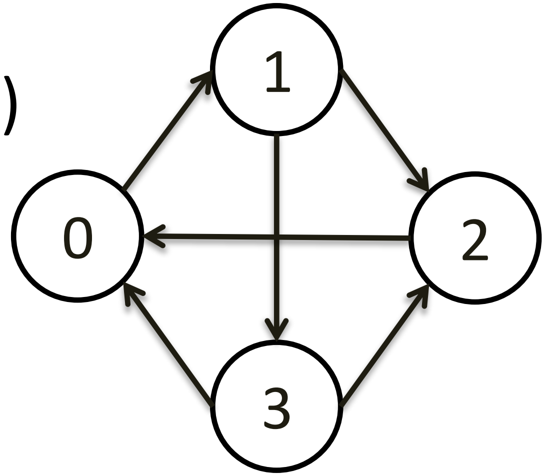
- The algorithm has 3 rounds
- Each round runs a map reduce step as described previously
- Interpretation of the rounds:
  - Input to Round $i$: $(X; \{x_1, x_2, \ldots, x_N\})$ : $\{x_1, x_2, \ldots, x_N\}$ denote all the vertices from which we can reach $X$ in $i - 1$ steps.
  - Map: $(d, s)$, all pairs of vertices such that we can reach $d$ from $s$ in $i$ steps.
  - Reduce:$(X; \{x_1, x_2, \ldots, x_N\})$: $\{x_1, x_2, \ldots, x_N\}$ denote all the vertices from which we can reach $X$ in $i$ steps. Used as input for map in the next round.



# Triangle Counting (5)

- Input: 0-1, 1-2, 1-3, 2-0, 3-0, 3-2
- Round 1:
  - $X_n$ : (0;0), (1;1), (2;2), (3;3)
  - Map: (1,0), (2,1), (3,1), (0,2), (0,3), (2,3)
  - Reduce: (1;0), (2;(1,3)), (0;(3,2)), (3;1)
- Round 2:
  - $X_n$: (1;0), (2;(1,3)), (0;(3,2)), (3;1)
  - Map: (2,0), (3,0), (0,1), (0,3), (0,1), (2,1), (1,2), (1,3)
  - Reduce: (2;(0,1)), (3;0),(0;(1,1,3)),(1;(2,3))

Map: Given (a;b) produce (destn(a),b)




(U,v): U is the 2-hop Neighbor of v.

# Triangle Counting (6)

- Round 3:
  - $X_n$ : (2;(0,1)), (3;0),(0;(1,1,3)),(1;(2,3))
  - Map: (0,0), (0,1), (0,0), (2,0), (1,1), (1,3), (1,1), (2,2), (2,3), (3,2), (3,3)
  - Reduce: (0;(0,1,0)), (2;(0,2,3)), (1;(1,3,1)), (3;(2,3))
- Output:
  - 0 -> 2
  - 2->1
  - 1-> 2
  - 3 -> 1

# Questions?

# Thank you

## Reference:

https://www.cs.rutgers.edu/~pxk/417/notes/content/mapreduce.html

http://www.slideshare.net/mcorrea11/map-reduce-5584234

http://static.googleusercontent.com/media/research.google.com/en/us/archive/mapreduce-osdi04.pdf

https://github.com/himank/K-Means