Part A:
Massive Parallelism with MapReduce

- Introduction
- Model
- Implementation issues
Acknowledgements

The material is largely based on

• material from the Stanford courses CS246, CS345A and CS347 (http://infolab.stanford.edu)

• the freely available textbook “Data-Intensive Text Processing with MapReduce”

• the “Mining of Massive Datasets” book.

Of course, all errors are mine!
MapReduce in a nutshell

- It is a programming model
- that is suitable for processing very large volumes of data on top of distributed infrastructures.
- It is based on ideas, principles and notions that are known since decades!
  - map and reduced are *adapted* from functional programming
- Initially, it was developed by Google;
  - nowadays most people use the Hadoop open source implementation.
Motivation

• Huge Data:
  – eBay (2009): $170 \cdot 10^{12}$ records, $150 \cdot 10^9$ new records daily, 2-6.5 PB user data.
  – LHC: 15PB/year.

• Note that many algorithms based on training (e.g., machine learning) perform better if they are trained with more data.
Building a text index - I

Web page stream

Buffers

Loading

Tokenizing

Sorting

Intermediate runs

Disk
Building a text index - II

Intermediate runs

Merge

Final Result

\[
\begin{align*}
(\text{cat}, 2) \\
(\text{dog}, 1) \\
(\text{dog}, 2) \\
(\text{dog}, 3) \\
(\text{rat}, 1) \\
(\text{rat}, 3)
\end{align*}
\]

\[
\begin{align*}
(\text{ant}, 5) \\
(\text{cat}, 2) \\
(\text{cat}, 4) \\
(\text{dog}, 1) \\
(\text{dog}, 2) \\
(\text{dog}, 3) \\
(\text{dog}, 4) \\
(\text{dog}, 5) \\
(\text{eel}, 6) \\
(\text{rat}, 1) \\
(\text{rat}, 3)
\end{align*}
\]

\[
\begin{align*}
(\text{ant}, 2) \\
(\text{cat}, 2, 4) \\
(\text{dog}, 1, 2, 3, 4, 5) \\
(\text{eel}, 6) \\
(\text{rat}, 1, 3)
\end{align*}
\]
Generic Processing Model: Map

- **Document stream**
- **Loading**
  - rat
  - dog
- **Tokenizing**
  - (rat, 1)
  - (dog, 1)
  - (dog, 2)
  - (cat, 2)
  - (rat, 3)
  - (dog, 3)
- **Map**
- **Sorting**
  - (cat, 2)
  - (dog, 1)
  - (dog, 2)
  - (dog, 3)
  - (rat, 1)
  - (rat, 3)
- **Intermediate runs**
- **Disk**

- **Buffers**

**MapReduce**
Map-Reduce

Generic Processing Model: Reduce

Intermediate runs

Merge

Reduce

Final Results
MapReduce: The Map Step

Input key-value pairs

Intermediate key-value pairs

Map

...
MapReduce: The Reduce Step

Intermediate key-value pairs

Key-value groups

Output key-value pairs

reduce

reduce

reduce
Short Interface Description

- Map: (key1, value) → (key2, value2) list
- Reduce: (key2, list of values2) → list of final_values

Alternatively:

- Map: (key1, value1) → [(key2, value2)]
- Reduce: (key2, [value2]) → [value3]  // in Hadoop, we may
  // change key2 to key3

- Key should not be understood in the strict database sense; more
  than one values can share the same key.
The famous Wordcount example

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

Example:
```
map(doc, "cat dog cat bat dog") emits
    [cat 1], [dog 1], [cat 1], [bat 1], [dog 1]
```
The famous Wordcount example – cont’d
reduce(String key, Iterator values);
    // key: word
    // values: counter list
    int result = 0;
    for each v in values:
        result += ParseInt(v)
    Emit(AsString(result));

Example:
reduce(“dog”, “1 1 1 1”) emits “4”
Summary of Parallel Execution

User Program

fork

fork

fork

Master

assign map

assign reduce

Worker

Worker

Worker

Worker

Worker

Worker

Input Data

Split 0

read

Split 1

local write

Split 2

remote read, sort

write

Output File 0

write

Output File 1
Coordination – master node

- The master mode
  - Checks if a task is
    - idle,
    - in-progress,
    - or completed.
  - Tasks are scheduled as soon as workers become available.
  - When a map task completes, the master is informed about the location and the size of intermediate results.
  - The master notifies the reducers.

- The master periodically pings all the workers
  - To detect failures.
Implementation issues

- Combine functions
- DFS
- Input/key partitioning
- Failures
- Backup Tasks
- Result sorting
Methods and Classes

```java
1: class MAPPER
2:     method MAP(docid a, doc d)
3:         for all term t ∈ doc d do
4:             EMIT(term t, count 1)

1: class REDUCER
2:     method REDUCE(term t, counts [c₁, c₂, ...])
3:         sum ← 0
4:         for all count c ∈ counts [c₁, c₂, ...] do
5:             sum ← sum + c
6:         EMIT(term t, count sum)
```

- Each MR job is split into tasks that comprise sequences of key-value pairs.
- For each task, we create a mapper object, which calls the map method for each key-value pair.
Map-Reduce

Combine functions

[cat 1], [cat 1], [cat 1]... worker

[dog 1], [dog 1]... worker

Same as if a *local* reduce is executed.

[cat 3]... worker

[dog 2]... worker
Map-Reduce

Distributed File System

All data transfers are done via the DFS.

Each worker accesses part of the global input (in splits)

Each reducer had access to the mappers' local disk space.

Each reduce worker can write in the same file.

Figure 1: Execution overview
Input partitioning

• Number of workers
  – We prefer to have many input splits per worker for load balancing and failure recovery purposes.

• How many splits?
  – More than the number of map workers,
  – Which may be already high.
  – Typically, an input split has roughly the same size as a DFS chunk, which is 64MB in most of Google’s applications.

• Try to benefit from data locality when assigning map tasks to workers
  – When this is possible.
Failures

- The master node, upon failure detection, re-allocates the tasks to another node.
- If failures are due to erroneous data, then the relevant split is removed.
Backup Tasks

- Some machines are relevantly slow, e.g., due to a broken disk. These machines are called stragglers.
- Stragglers may slow down the entire execution.
  - In parallel execution, the total running time depends on the slowest machine.
- Solution: near the end of execution, the master schedules redundant tasks.
  - At least one such task should complete normally.
  - Such an approach requires mechanisms to handle duplicate results.
Result sorting

- Sorting is performed automatically (i.e., as a built-in feature).

![Diagram of Map-Reduce process]

Figure 1: Execution overview
Pros

• All technical details w.r.t parallelism and fault tolerance are hidden.
• Although the system is simple, it is flexible enough to support many problems.
• MR applications can scale to thousands of machines.
  – Note that other parallel models, including some forms of shared nothing databases, can exhibit similar scalability.
Cons

- Not arbitrary scenarios can be supported, at least in an elegant/straightforward manner.
  - Due to the single input, two-stage processing model.
- Need to write code even for the simplest tasks.
  - Task declaration is at a much lower level than in SQL.
- The fact that the code inside map and reduce functions is treated as a black box, prohibits optimizations.
Part B:
Algorithm design and basic data management operators in MR

- Design Techniques
- Relational operators
- Matrix Multiplication
Reminder: Methods and Classes

1: class MAPPER
2:  method MAP(docid a, doc d)
3:  for all term t ∈ doc d do
4:    EMIT(term t, count 1)

1: class REDUCER
2:  method REDUCE(term t, counts [c₁, c₂, ...])
3:    sum ← 0
4:  for all count c ∈ counts [c₁, c₂, ...] do
5:    sum ← sum + c
6:  EMIT(term t, count sum)

• Each MR job is split into tasks that comprise sequences of key-value pairs.
• For each task, we create a mapper object, which calls the map method for each key-value pair.
Improvement of Mapper

1: class MAPPER
2: method MAP(docid a, doc d)
3: \[ H \leftarrow \text{new ASSOCIATIVE ARRAY} \]
4: \[ \text{for all term } t \in \text{doc } d \text{ do} \]
5: \[ H\{t\} \leftarrow H\{t\} + 1 \]
6: \[ \text{for all term } t \in H \text{ do} \]
7: \[ \text{EMIT(term } t, \text{count } H\{t\}) \]

• Instead of emitting a key-value pair for each term in \( d \),
  this version emits a key-value pair for each unique term in \( d \).
Further Improvement

```java
1: class Mapper
2:   method INITIALIZE
3:     H ← new AssociativeArray
4:   method Map(docid a, doc d)
5:     for all term t ∈ doc d do
6:       H{t} ← H{t} + 1
7:   method Close
8:     for all term t ∈ H do
9:       Emit(term t, count H{t})
```

- In-mapper combining:
  - Full control of local aggregations;
  - More efficient because it emits less pairs;
  - But breaks the functional programming model and requires more memory\(^{29}\)
Another combiner example

```java
class Mapper
method MAP(string t, integer r)
    EMIT(string t, pair (r, 1))

class Combiner
method COMBINE(string t, pairs [(s1, c1), (s2, c2) ...])
    sum ← 0
    cnt ← 0
    for all pair (s, c) ∈ pairs [(s1, c1), (s2, c2) ...] do
        sum ← sum + s
        cnt ← cnt + c
    EMIT(string t, pair (sum, cnt))

class Reducer
method REDUCE(string t, pairs [(s1, c1), (s2, c2) ...])
    sum ← 0
    cnt ← 0
    for all pair (s, c) ∈ pairs [(s1, c1), (s2, c2) ...] do
        sum ← sum + s
        cnt ← cnt + c
    r_avg ← sum/cnt
    EMIT(string t, integer r_avg)
```

Computes the average of values for each key
Following in-mapper combining design

1: class MAPPER
2:     method INITIALIZE
3:         S ← new ASSOCIATIVEARRAY
4:         C ← new ASSOCIATIVEARRAY
5:     method MAP(string t, integer r)
6:         S\{t\} ← S\{t\} + r
7:         C\{t\} ← C\{t\} + 1
8:     method CLOSE
9:         for all term t ∈ S do
10:             Emit(term t, pair (S\{t\}, C\{t\}))
Relational operators

- Selections
- Projections
- Union, Intersection, and Difference
- Joins
- Grouping and Aggregation
Selections

- $\sigma_C(R)$
- Map: For each tuple $t$ in $R$, test $C$.
  - If it satisfies the predicate emit $(t,t)$ ;
  - or $(t,\text{NULL})$, so that all the info is in the key
- Reduce: nothing to be done
Map-Reduce

Projections

- $\pi_C(R)$
- Map: For each tuple $t$ in $R$, produce $t'$ and emit $(t', t')$
- Reduce: receive $(t', [t', t', t'..., t'])$ and emit $(t', t')$
  - We perform duplicate elimination

Alternatively:

- Map: For each tuple $t$ in $R$, produce $t'$ and emit $(t', 1)$
- Reduce: receive $(t', [1, 1, 1..., 1])$ and emit $(t', NULL)$
  - We perform duplicate elimination
Map-Reduce

Unions

- $R(X,Y) \cup S(Y,Z)$
- Map: For each tuple $t$ either in $R$ or in $S$, and emit $(t,t)$
- Reduce: either receive $(t, [t,t])$ or $(t,[t])$
  - Always emit $(t,t)$
  - We perform duplicate elimination

Alternatively:

- Map: For each tuple $t$ in $R$, produce $t'$ and emit $(t',1)$
- Reduce: receive $(t',[1,1])$ or $(t',[1])$ and emit $(t',NULL)$
  - We perform duplicate elimination
Intersections

- $R(X,Y) \cap S(Y,Z)$
- Map: For each tuple $t$ either in $R$ or in $S$, emit $(t,t)$
- Reduce: either receive $(t,[t,t])$ or $(t,[t])$
  - Emit $(t,t)$ in the former case and nothing in the latter.
Differences

- $R(X,Y) - S(Y,Z)$
- Map: For each tuple $t$ either in $R$ or in $S$, emit $(t, R \text{ or } S)$
- Reduce: receive $(t, [R])$ or $(t, [S])$ or $(t, [R, S])$
  - Emit $(t, t)$ only when received $(t, [R])$
Simple group-by queries

- Employees(id, dno, salary).
  
  Select dno, SUM(salary)
  
  from employees
  
  where salary>1000
  
  group by dno

- **Map:** for each tuple s.t. salary>1000, emit a pair (dno, salary)

- **Reduce:** for each value of dno, compute the sum of the associated list with the multiple values of salary.
Reduce-Side Joins

- \( R(X,Y) \bowtie S(Y,Z) \).
  - **Map:**
    - Input: (relation name \( R \) or \( S \), tuples \( t \))
    - Output: list (\( Y \) value, list (relation name, remainder of tuples \( X \) or \( Z \))).
  - **Reduce:** for each \( Y \), create all pairs \( XYZ \).

- **Secondary sort:**
  - Extends the key with part of the value.
  - Allows the reducer to receive input in specific order.
  - Good for 1-to-many joins.
  - Requires a partitioner.
Map-Side Joins

- $R(X,Y) \bowtie S(Y,Z)$.
  - Less generic
  - Assume that both relations are sorted by the join key.
  - Assume that both relations have identically split inputs.
  - Similar to merge-join.
  - Map: local merge join, receive as input a split from one relation and read the corresponding partition from the other relation within map.
  - Reduce: nothing.
Comparison

• **Map-side**
  (+) can reduce intermediate data significantly if highly selective
  (-) requires identically keyed/split inputs

• **Reduce-side**
  (+) works with any input
  (+) may be easier to use if number of inputs is expected to increase
  (-) intermediate data size ~ input size
Matrix – Vector Multiplication

• It is essential in algorithms such as PageRank
• Suppose we have an \( n \times n \) matrix \((M)\) and a vector \(v\) of length \(n\)
• \(Mv = x\), where \(x_i = \sum m_{ij} v_j\)
• Map:
  – read a chunk of \(M\) and all \(v\).
  – For each element of \(M, m_{ij}\), produce \((i, m_{ij}v_j)\)
• Reduce: sum all values with a given key \(i\).
Matrix –Vector Multiplication

- If \( v \) does not fit in main memory:
Matrix –Matrix Multiplication in 2 steps

• Relational Representation
• Suppose we have an \((i \times j)\) matrix \(M\) and a \((j \times k)\) matrix \(N\)
• \(M \rightarrow (i,j,m_{ij}), N \rightarrow (j,k,n_{jk})\)
• Map1: for each element \(m_{ij}\) emit \((j, (M,i,m_{ij}))\). Similarly, do the same for \(n_{jk}\) values.
• Reduce 1: For each key \(j\), produce \((j, (i,k,m_{ij},n_{jk}))\) –similarities with join.
• Map2: for each \((i,k,m_{ij},n_{jk})\) value, emit \(( (i,k), m_{ij}n_{jk})\).
• Reduce2: for each \((i,k)\) key, sum the values.
Matrix – Matrix Multiplication in 1 step

- Relational Representation
- Suppose we have an \((i \times j)\) matrix \(M\) and a \((j \times k)\) matrix \(N\)
- \(M \rightarrow (i,j,m_{ij}), N \rightarrow (j,k,n_{jk})\)
- Map1: for each element \(m_{ij}\) emit \(k\) pairs: \(((i,k'),(M,j,m_{ij}))\)
  - \(k': 1..k\)
- Similarly, for each \(n_{jk}\) emit \(i\) pairs: \(((i',k),(N,j,n_{ik}))\)
  - \(i'=1..i\)
- Reduce 1: For each key:
  - Multiply \(m_{ij}\) with \(n_{ik}\) if they have the same \(j\) value;
  - Sum the products.
Part C:
Data Mining Examples

- URL access frequency
- Friend Recommendation
- Frequent itemsets
Map-Reduce

URL access frequency

- We assume log files that contain the URL visited.

- Map: process log files and emit \( <\text{URL}, 1> \) for each log entry.

- Reduce: sum all values for a given URL key, thus producing \( <\text{URL}, \text{total count}> \).
Frequent Itemsets

- **Map 1**: find all frequent itemsets using any main-memory algorithm in a split with transaction records; output a key-value pair \((L, 1)\) for each itemset \(L\) found frequent in the split. The exact support value does not play any role.

- **Reduce 1**: do nothing with the value \(\Rightarrow\) eliminate duplicate keys, i.e., locally frequent itemsets. The result contains a list of \(C\) candidate frequent itemsets.

- **Map 2**: read a) all output \(C\) of Reduce1, and b) an input split. For each candidate itemset \(c\) in \(C\), emit its support in that split in the form \((c, \text{support})\)

- **Reduce2**: sum the supports for each key \(c\) of Map2; if the sum is not smaller than the support threshold, emit \((c, \text{sum})\)
Map-Reduce

Friend Recommendation

• Suppose that friend connections in a social network are stored as an adjacency list (graph representation)
  – E.g., p245: p125, p246, p347, p893, p899

• Recommend friend links if both persons have mutual friends; rate the recommendations.

• Map:
  – Process records of the form p_id: n1, n2, n3, ..., n_M
  – Emit all pairs (p_id, (ni, 0)), (ni, (nk, 1)), i<>k and i, k: 1..M

• Reduce:
  – For each pair (a, (b, count)) sum the counts for each a-b combination if there is no zero value (why?) and sort by the sum.