MapReduce for Graph Algorithms
Modeling & Approach

Ankur Sharma
ankur@stud.uni-saarland.de

May 8, 2015
Agenda

Map-Reduce Framework
  Big Data
  MapReduce

Graph Algorithms
  Graphs
  Graph Algorithms

Design pattern

Summary & Conclusion
Big Data: Definition

- Sources? : Web pages, social network, tweets, messages and many more
- How Big? : Google’s Index is larger than 100 Peta Bytes and consumed over 1M computing hours to build \(^a\)
- Number of webpages indexed: 48B (Google) & 5B (Bing) \(^b\)
- Big Data Market will account $50 Billion by 2017 \(^c\)

\(^a\)https://www.google.com/search/about/insidesearch/howsearchworks/crawling-indexing.html
\(^b\)www.worldwidewebsize.com
\(^c\)www.wikibon.org
Ankur Sharma | MapReduce for Graph Algorithms

Figure: Comic from Socmedsean¹

¹Source: www.socmedsean.com
Quick and efficient analysis of Big Data requires extreme parallelism.

A lot of systems were developed in late 1990s for parallel data processing such as SRB\(^a\).

These systems were efficient but lacked nice wrapper for average developer.

In 2004, Jeffrey Dean and Sanjay Ghemawat from Google introduced MapReduce\(^b\) which is a part of company’s proprietary infrastructure.

Similar open source infrastructure Hadoop\(^c\) was developed in late 2005 by Doug Cutting & Mike Cafarella.

\(^a\)Wan et. al. The SDSC storage resource broker, CASCON ’98  
\(^b\)Ghemawat et. al. MapReduce: Simplified data processing on large clusters, OSDI’14  
\(^c\)www.hadoop.apache.org
### MapReduce: Framework

#### Introduction

- MapReduce motivates from LISP’s map and reduce operations
  - \((\text{map square} \ (1 2 3 4 5)) \rightarrow (1 4 9 16 25)\)
  - \((\text{reduce } + \ (1 2 3 4 5)) \rightarrow (15)\)
- Framework provides *automatic parallelism, fault-tolerance, I/O scheduling and monitoring* for data processing
MapReduce: Framework

Introduction

- MapReduce motivates from LISP’s map and reduce operations
  
  - \((map \ square \ '(1 \ 2 \ 3 \ 4 \ 5)) \rightarrow (1 \ 4 \ 9 \ 16 \ 25)\)
  
  - \((reduce \ + \ '(1 \ 2 \ 3 \ 4 \ 5)) \rightarrow (15)\)

- Framework provides *automatic parallelism*, *fault-tolerance*, *I/O scheduling and monitoring* for data processing

MapReduce: Programming Model

- Input / Output are a set of key value pairs
- User only needs to provide simple functions map(), reduce() and combine()\(^a\)
  
  - \(map\(in\_k, \ in\_v) \rightarrow list(out\_k, \ imt\_v)\)
  
  - \(combine(in\_k, list(in\_v)) \rightarrow list(out\_k, \ imt\_v)\)
  
  - \(reduce(out\_k, list(imt\_v)) \rightarrow list(out\_v)\)

\(^a\)optional
MapReduce: Execution Model

Figure: Overview: Map-Reduce execution model

Source: Ghemawat et. al. *MapReduce: Simplified data processing on large clusters*, OSDI’14

Ankur Sharma | MapReduce for Graph Algorithms

\(^2\) Source: Ghemawat et. al. *MapReduce: Simplified data processing on large clusters*, OSDI’14
MapReduce: Example
Word Count

Map function $M(in_k, in_v)$
for each word in in_v:
emit(word, 1)
end function

Combine/Reduce function $C_{-R}(out_k, list{imt_v})$
emit(word, $\sum_{imt_v}$)
end function
### MapReduce: Example

**Word Count**

<table>
<thead>
<tr>
<th><strong>Map</strong></th>
<th><strong>Combine/Reduce</strong></th>
</tr>
</thead>
</table>
| **function**  
\[
M(\text{in}_k, \text{in}_v) \\
\text{for each word in in}_v: \\
\quad \text{emit(word, 1)} \\
\text{end function}
\] | **function**  
\[
C_R(\text{out}_k, \text{list}\{\text{imt}_v\}) \\
\quad \text{emit(word, } \sum \text{imt}_v) \\
\text{end function}
\] |
MapReduce: Example

**Word Count**

**Map**

```plaintext
function M(in_k, in_v)
    for each word in in_v:
        emit(word, 1)
    end function
```

**Combine/Reduce**

```plaintext
function C_R(out_k, list{imt_v})
    emit(word, \( \sum \) imt_v)
end function
```

Diagram:

- **Input:** Deer Bear River, Car, Car, River, Deer, Car, Bear
- **Splitting:** Deer Bear River → Car, Car, River
- **Mapping:** Deer, 1 → Bear, 1, Bear, 1, River, 1
- **Shuffling:** Car, 1, Car, 1, River, 1 → Car, 1, Car, 1, Car, 1
- **Reducing:** Bear, 1, Bear, 1 → Bear, 2, Bear, 2, Car, 3, Deer, 2, River, 2
- **Final result:** Bear, 2, Car, 3, Deer, 2, River, 2
Graphs

- Graphs are everywhere: social graphs representing connections or citation graphs representing hierarchy in scientific research.
- Due to massive scale, it is impractical to use conventional techniques for graph storage and in-memory analysis.
- These constraints had driven the development of scalable systems such as distributed file systems like Google File System\(^a\) and Hadoop File System\(^b\).
- MapReduce provides a good way to partition and analyze graphs as suggested by Cohen et. al.\(^c\).

---

\(^a\)Ghemawat et. al. *The Google File System*, SOSP ’03
\(^b\)www.hadoop.apache.org
\(^c\)Jonathan Cohen, Graph twiddling in a MapReduce world, Computing in Science & Engineering 2009
Augmenting edge with vertex degree

**Input:** All edges as *Key Value Pair* → \(<-, e_{ij} > : edge connecting v_i & v_j\)

**MapReduce Job I**
- **map**\(\left<-, e_{ij} \right> \rightarrow \text{list}\[\left<v_i, e_{ij} \right>, \left<v_j, e_{ij} \right>\]\
- **reduce**(\(v_k, \text{list}\left[e_{pk}, e_{qk} \ldots \right]\)) → \([\left<e_{pk}, \text{deg}(v_k) \right>, \left<e_{qk}, \text{deg}(v_k) \right>, \ldots]\)
  - where \(\text{deg}(v_k) = \text{size of the list in the parameter}\)

**Input:** Output of Job I

**MapReduce Job II**
- **map**() is an *identity* function: does no modification to input
- **reduce**\(\left<e_{ij}, \text{list}\left[\text{deg}(v_i), \text{deg}(v_j) \right]\right> \rightarrow \left<e_{ij}, \text{deg}(v_i), \text{deg}(v_j) \right>\)
Augmenting edge ... contd

Example

Map 1

<table>
<thead>
<tr>
<th>key</th>
<th>(FRED, ETHEL)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(FRED, ETHEL)</td>
</tr>
<tr>
<td>key</td>
<td>(FRED, RICKY)</td>
</tr>
<tr>
<td></td>
<td>(FRED, RICKY)</td>
</tr>
</tbody>
</table>

Reduce 1

<table>
<thead>
<tr>
<th>FRED</th>
<th>(FRED, ETHEL)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FRED</td>
<td>(FRED, RICKY)</td>
</tr>
<tr>
<td>FRED</td>
<td>(LUCY, FRED)</td>
</tr>
<tr>
<td>FRED</td>
<td>(FRED, TIM)</td>
</tr>
</tbody>
</table>

Map 2 (Identity)

<table>
<thead>
<tr>
<th>(FRED, ETHEL)</th>
<th>(FRED, ETHEL)</th>
<th>d(FRED) = 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>(FRED, ETHEL)</td>
<td>(FRED, ETHEL)</td>
<td>d(FRED) = 4</td>
</tr>
</tbody>
</table>

Reduce 2

<table>
<thead>
<tr>
<th>(FRED, ETHEL)</th>
<th>(FRED, ETHEL)</th>
<th>d(FRED) = 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>(FRED, ETHEL)</td>
<td>(FRED, ETHEL)</td>
<td>d(ETHEL) = 2</td>
</tr>
<tr>
<td>(FRED, ETHEL)</td>
<td>(FRED, ETHEL)</td>
<td>d(FRED) = 4</td>
</tr>
</tbody>
</table>
The main idea behind enumerating the triangles is to find triad and an edge joining the open ends together.

**Input**: Augmented edges with vertex valency

### MapReduce Job 1

**Map**

- `map(<e_{ij}, deg(v_i), deg(v_j)>)` →
  - output: `deg(v_i) < deg(v_j) ? <v_i, e_{ij}> : <v_j, e_{ij}>`
  - If `deg(v_i) == deg(v_j)`, we can break the tie by any consistent method such as using vertex names

**Reduce**

- `reduce(<v_i, list[edges connecting v_i]>)` →
  - For each pair of edge connected by `v_i`, emit `<e_1, e_2>` with key as `<v_p, v_q>`, where `v_p` and `v_q` represent the open end of triad
  - The order of `v_p` and `v_q` is decided using some consistent technique like vertex names
Enumerating Triangles ...contd

**Input**: Augmented edge set and output of Job I

**MapReduce Job 2**

**map(<R>) →**

- if $R$ is a record from Augmented Edge Set, change key to $<v_i, v_j>$ such that $\text{deg}(v_i) < \text{deg}(v_j)$
- if $R$ is a record from MR Job I and hence a **triad**, change key to $<v_i, v_j>$ such that $\text{deg}(v_i) < \text{deg}(v_j)$, where $v_i$ & $v_j$ are vertices of open ends

**reduce([<v_i, v_j>, list[containing a triad, edge]a]) →**

- If there is a record corresponding to an edge and a triad with same key, emit a triangle $<e_{ij}, e_{jk}, e_{ki}>$

---

*May or may not be there*
**Idea:** Two triads joining with a common edge can be merged to form a rectangle.

- $4! = 24$ different orderings. Since symmetry group ($S_4$) on 4 elements has $|S_4| = 8$, we have $\frac{24}{8} = 3$ distinct orderings.
Enumerating Rectangles

**Idea:** Two triads joining with a common edge can be merged to form a rectangle

1. **MapReduce Job I**
   - Map: $<e_{ij}, \text{deg}(v_i), \text{deg}(v_j)> \rightarrow [v_i, H, e_{ij}, v_j, L, e_{ij}]$
   - Reduce: $<v_k, \text{list}[\text{edges incident on } v_k]> \rightarrow$ emit a triad for L-L & H-L record
   - If we express $\text{deg}(v) = \text{deg}_L(v) + \text{deg}_H(v)$, each vertex appears in $O(\text{deg}_L^2(v))$ low triads and $O(\text{deg}_L(v)\text{deg}_H(v))$ mixed triads

2. **MapReduce Job II**
   - Map: $<v_k, \text{list}[\text{triads}]> \rightarrow \text{list}[<\text{key: edge joining open ends}, \text{value: triad}>]$
   - Reduce: $<e_{ij}, \text{list}[\text{triads open at same end}]> \rightarrow$ emit rectangle: $[e_{ij}, e_{jk}, e_{kl}, & e_{li}]$
**Idea:** Two triads joining with a common edge can be merged to form a rectangle

- $4! = 24$ different orderings. Since symmetry group($S_4$) on 4 elements has $|S_4| = 8$, we have $\frac{24}{8} = 3$ distinct orderings

**MapReduce Job I**

- **map**($<e_{ij}, \text{deg}(v_i), \text{deg}(v_j)>)$ →
  - [$<v_i, H, e_{ij}>, <v_j, L, e_{ij}>$]
- **reduce**($<v_k, \text{list[edges incident on } v_k]>)$ →
  - emit a triad for L-L & H-L record
  - if we express $\text{deg}(v) = \text{deg}_L(v) + \text{deg}_H(v)$, each vertex appears in $O(\text{deg}_L^2(v))$ low triads and $O(\text{deg}_L(v)\text{deg}_H(v))$ mixed triads

**MapReduce Job II**

- **map**($<v_k, \text{list[triads]>}) →
  - list[<key: edge joining open ends, value: triad>]
- **reduce**($<e_{ij}, \text{list[triads open at same end]}>)$ →
  - emit rectangle: [$e_{ij}, e_{jk}, e_{kl}, \& e_{li}$]
**Finding k-truss**

**Definition:** A k-truss is a sub-graph in which each edge is a part of at least k-2 triangles.

**MapReduce Job**

*Input:* Augmented edge set & enumerated triangles of the graph

- **map:** `<e_{ij}, triangle_i>` →
  - if `(e_{ij} \in triangle_i)`: `<e_{ij}, 1>`

- **reduce:** `<e_{ij}, list[1,1,......,1]>` →
  - if `(\sum list_i) \geq k`: emit `e_{ij}`

All edges emitted by map-reduce job belongs to the k-truss.
Clusters are sub-graphs that partitions graph without loosing the co-relevant information. Barycentric Clustering is a highly scalable approach of finding tightly connected sub-graphs.

- \( w_{ij} \) : weight of edge \( e_{ij} \) and \( d_i = \sum w_{ij} \) : weighted degree of vertex \( v_i \)
- If \( x = (x_1, x_2, \ldots, x_n)^T \) denotes the position of \( n \) vertices, then \( x' = Mx \) modifies the vertex position by the average of its old position and position of its neighborhood.
- The process is repeated and literature\(^3\) suggests that 5 repetitions are adequate.
- If the length of edge > threshold, it can be cut

\[
M = \begin{pmatrix}
\frac{1}{d_1 + 1} & \frac{w_{12}}{d_1 + 1} & \cdots & \frac{w_{1n}}{d_1 + 1} \\
\frac{w_{21}}{d_2 + 1} & \frac{1}{d_2 + 1} & \cdots & \frac{w_{2n}}{d_2 + 1} \\
\vdots & \vdots & \ddots & \vdots \\
\frac{w_{n1}}{d_n + 1} & \frac{w_{n2}}{d_n + 1} & \cdots & \frac{1}{d_n + 1}
\end{pmatrix}
\]

\(^3\)Cohen et. at. Barycentric Graph Clustering, 2008
MapReduce Job I

- map($<e_{ij}>$) → [$<v_i, e_{ij}>$, $<v_j, e_{ij}>$]
- reduce($<v_i$, list[$e_{ij}$, $e_{ik}$,...]$) → [$<e_{ij}, P(v_i)>$, $<e_{ik}, P(v_i)>$, ...], where $P(v)$ denotes the position vector of vertex $v$. 

Now we execute 5 iterations of next two map reduce jobs

MapReduce Job II
- map : (identity)
- reduce($<e_{ij}$, list[$P(v_i)$, $P(v_j)$]$) → <$e_{ij}$, $P(v_i)$, $P(v_j)$>

MapReduce Job III
- map($<e_{ij}$, $P(v_i)$, $P(v_j)$)$) → [$<v_i$, $e_{ij}$, $P(v_i)$, $P(v_j)>$, $<v_j$, $e_{ij}$, $P(v_i)$, $P(v_j)>$]
- reduce($<v_i$, list[$e_{ij}$, $P(v_i)$, $P(v_j)$]$,...$) → calculates modified $P'(v_i) = M * P'(v_i)$ and emits [$<e_{ij}$, $P'(v_i)$, $P(v_j)>$, $<e_{ik}$, $P'(v_i)$, $P(v_k)>$, ...]
MapReduce Job I

- **map**($<e_{ij}>$) → $[<v_i, e_{ij}>, <v_j, e_{ij}>]$
- **reduce**($<v_i, list[e_{ij}, e_{ik}, ...]>$) → $[<e_{ij}, P(v_i)>, <e_{ik}, P(v_i)>, ...]$, where $P(v)$ denotes the position vector of vertex $v$.

MapReduce Job II

- **map** : (identity)
- **reduce**($<e_{ij}, list[P(v_i), P(v_j)]>$) → $<e_{ij}, P(v_i), P(v_j)>$
Barycentric Clustering ... contd

MapReduce Job I

- **map**($e_{ij}$) → [$v_i$, $e_{ij}$, $v_j$, $e_{ij}$]
- **reduce**($v_i$, list[$e_{ij}$, $e_{ik}$,...]) → [$e_{ij}$, $P(v_i)$, $e_{ik}$, $P(v_i)$,...], where $P(v)$ denotes the position vector of vertex $v$.

MapReduce Job II

- **map** : (identity)
- **reduce**($e_{ij}$, list[$P(v_i)$, $P(v_j)$]) → $e_{ij}$, $P(v_i)$, $P(v_j)$

Now we execute 5 iterations of next two map reduce jobs

MapReduce Job III

- **map**($e_{ij}$, $P(v_i)$, $P(v_j)$) → [$v_i$, $e_{ij}$, $P(v_i)$, $P(v_j)$, $v_j$, $e_{ij}$, $P(v_i)$, $P(v_j)$]
- **reduce**($v_i$, list[$e_{ij}$, $P(v_i)$, $P(v_j)$, $e_{ik}$, $P(v_i)$, $P(v_k)$,...]) → calculates modified $P'(v_i) = M*P'(v_i)$ and emits [$e_{ij}$, $P'(v_i)$, $P(v_j)$, $e_{ik}$, $P'(v_i)$, $P(v_k)$, ...]
MapReduce Job IV

- map: (identity)
- reduce($<e_{ij}, \text{list}[<e_{ij}, P'(v_i), P(v_j)>], <e_{ij}, P(v_i), P'(v_j)>]+$) $\rightarrow <e_{ij}, P'(v_i), P'(v_j)>$
MapReduce Job IV

- map: (identity)
- reduce\(\langle e_{ij}, \text{list[<e_{ij}, P'(v_i), P(v_j)>]}, <e_{ij}, P(v_i), P'(v_j)>\rangle \rightarrow <e_{ij}, P'(v_i), P'(v_j)>\)

MapReduce Job V

- map\(e_{ij}, \text{list}[P_1, P_2, ..., P_r]\) \rightarrow \[<v_i, \text{AVG}(P_i), \text{AVG}(P_j)>, <v_j, \text{AVG}(P_j)>\]
- reduce\(\langle v_i, \text{list[<e_{ij}, P_j>, <e_{ik}, P_k>..]>}\rangle \rightarrow \[<e_{ij}, >, <e_{ik}, \sum P_j, P_k, ...>...], \sum P_j, P_k, ..., a_{ij} = \sum P_j, P_k, ... \text{ represents the weight of edge } e_{ij} \)

Ankur Sharma | MapReduce for Graph Algorithms
### MapReduce Job IV

- **map**: (identity)
- **reduce**($\langle e_{ij}, \text{list}[e_{ij}, P'(v_i), P(v_j)], e_{ij}, P(v_i), P'(v_j)\rangle \rightarrow \langle e_{ij}, P'(v_i), P'(v_j)\rangle$)

### MapReduce Job V

- **map**($e_{ij}, \text{list}[P_1, P_2, ..., P_r]$) → $\langle v_i, \text{AVG}(P_i), \text{AVG}(P_j), v_j, \text{AVG}(P_j)\rangle$
- **reduce**($\langle v_i, \text{list}[e_{ij}, P_j, e_{ik}, P_k, ...], e_{ij}, \sum P_j, P_k, ..., a_{ij} = \sum P_j, P_k, ... \rangle \rightarrow [e_{ij}, >, e_{ik}, \sum P_j, P_k, ...], \sum P_j, P_k, ..., a_{ij} = \sum P_j, P_k, ...$ represents the weight of edge $e_{ij}$.

### MapReduce Job VI

- **map**: (identity)
- **reduce**($\langle e_{ij}, \text{list}[\text{weights emitted by previous map reduce job}]\rangle$) : computes $\bar{a}_{ij}$, if $\bar{a}_{ij} > a_{ij}$, emit the edge

$$
\bar{a}_{ij} = \frac{\sum_{k \in N_i} a_{jk} + \sum_{k \in N_j} a_{jk} - a_{ij}}{|N_i| + |N_j| - 1}
$$

- where $N_k$ represents the set of vertices in the neighborhood of $v_k$.
Finding Components

Algorithm using MapReduce

- **Step 1** → consists of two MapReduce Jobs focussing on translating the vertex-vertex information into zone zone information
  - **MapReduce 1** assigns edges with zones, producing 1 record for each vertex in edge and then translating this information to assign 1 zone to both the vertices
  - **MapReduce 2** merges the result of Job 1. If distinct zones are connected by an edge, then the zone with lower zone_id captures the other

- **Step 2** → consists of one MapReduce Job
  - **MapReduce 1** It record any updates to zone information for each vertex
  - Step 1 and 2 alternate until step 1 produces no new record, which marks the completion of zone assignment

Figure: Graph with three components
Lin et. al\textsuperscript{a} suggested following design patterns for efficiently implementing graph algorithms using MapReduce.

- **Local aggregation** done using combiners reduce the amount intermediate data. It is effective only in case of high duplication factor of intermediate key-value pair.

- **In-Mapper Combining** performs better than than regular combiner. Combiner often have to create and destroy objects and perform object serialization and de-serialization which is often costly.

- **Schimmy Pattern** divides the graph into \( n \) partitions each sorted by *vertex id*. Number of reducers are then set to \( n \) and it is guaranteed that reducer \( R_i \) processes same *vertex ids* as in partition \( P_i \) in sorted manner. Hence reducing the cost of shuffle and sort.

- **Range Partitioning** proposes to maximize intra-block connectivity and minimize inter-block connectivity. Dividing a graph into blocks with such a property helps reducing the amount of communication over network.

\textsuperscript{a}Jimmy Lin, Michael Schatz, Design patterns for efficient graph algorithms in MapReduce, MLG’10
Design Patterns ... contd
Performance Comparison

Figure: Performance comparison for PageRank using suggested strategies.\(^4\)

\(^4\) Jimmy Lin, Michael Schatz, Design patterns for efficient graph algorithms in MapReduce, MLG’10
In general, it is a good contribution to attract MapReduce community to develop distributed systems for graph processing.

Author focused on breaking down smaller problems in map-reduce jobs and merging them together to solve bigger problem.

The contribution provides no information about the performance of the implementation. No comparison to standard approach.

Most of the algorithms need complete re-transformation for implementation using MR framework. Some may become easy and some like finding component becomes complex.

Lin et. al. introduced some really good strategies for efficient MapReduce implementation which can be used even for non-graph algorithms.
Thank You
Questions?