Asynchronous Algorithms in MapReduce

Karthik Kambatla
Naresh Rapolu
Prof. Suresh Jagannathan
Prof. Ananth Grama
Agenda

• Background
  – Asynchronous Algorithms, MapReduce
• Motivation
• Relaxed semantics for MapReduce
• Evaluation
• Conclusions
Asynchronous Algorithms

• Improve performance in parallel environments.
  – Infrequent synchronization reduces communication
  – Can increase data locality
  – Examples
    • Graph algorithms, Numerical methods, Classification, etc.

• More pronounced gains in distributed environments
  – Higher communication and data-movement costs
MapReduce

• Distributed execution engine: easy-to-use, highly scalable, fault-tolerant.
  – Processes big data on commodity hardware
  – Great for embarrassingly data-parallel applications

• Usage
  – Data analytics and other operations on web-scale data
  – Does it lend itself to other kinds of parallelism and optimizations?
MapReduce Model

INPUT <key, value>: list

Map1
- Input: <K11, V11>
- Output: <K’11, V’11>, …, <K’1n, V’1n>

Map2
- Input: <K21, V21>
- Output: <K’21, V’21>, …, <K’2n, V’2n>

Map3
- Input: <K31, V31>
- Output: <K’31, V’31>, …, <K’3n, V’3n>

INTERMEDIATE <key, value:list>: list

Reduce1
- Input: <K’11, V’11:list>
- Output: <K”11, V”11>, …, <K”1n, V”1n>

Reduce2
- Input: <K’21, V’21:list>
- Output: <K”21, V”21>, …, <K”2n, V”2n>

OUTPUT value:list
MapReduce: Data Flow

Master

Mapper

Reducer

Reducer

Mapper

Distributed File System
Observations

• I/O takes longer than Computation
  – More pronounced in iterative algorithms
    • Especially when the input doesn’t change much across iterations

• Strict synchronization barrier
  – Between map and reduce
  – Between iterations in iterative MapReduce

• Limited applicability of known optimizations
  – Asynchronous algorithms, speculative parallelism etc.
Relaxed Synchronization

• Hierarchical MapReduce: local and global
  – Each iteration in a MapReduce job has multiple iterations of local MapReduce
  – **Partial synchronization**: after every local iteration, synchronize on subset of the data
  – **Global synchronization**: at the end of local iterations, synchronize on the whole data across all machines
  – Input data **partitioning**
    • Fewer dependencies across partitions
Illustrative Example: PageRank

• Weighted in-links determine the rank of a node.

\[ PR_d = (1 - \chi) + \chi \times \sum_{(s,d) \in E} s.pagerank / s.outlinks \]

• Implementation: Power method
  – Each iteration executes a MapReduce
    • map: each node emits (dest, pagerank/#outlinks)
    • reduce: sums up all the in-link weights and computes pagerank
  – Input: power-law
    • Many strongly-connected components with few edges across components
General PageRank

Synchronization Barrier

- Maps
- Reduces
Relaxed PageRank

Synchronization Barrier

Maps
Reduces
Semantics: Iterative MapReduce

\[
\begin{align*}
l, \sigma &\rightarrow \sigma(l) \quad \text{(LOCAL-LOOKUP)} \\
\text{Apply}(I, < e, f_m, f_r, l>) &\rightarrow_g I e f_m f_r l \\
\text{while}(\text{cond}_g) G \text{ cond } f_m f_r l_g, \lambda &\rightarrow_g l'_g, \lambda' \\
I \text{ cond } f_m f_r l_g, \lambda &\rightarrow_g l'_g, \lambda' \\
\text{while cond map } (L f_m f_r) \bar{l}_l, \sigma &\rightarrow_{\eta_\bar{l}_l'} \bar{l}_l', \sigma' \text{ agg } \bar{l}_l', \sigma, \lambda \equiv l'_g, \sigma, \lambda' \text{ fold } f_r l_g', \lambda &\rightarrow_g l''_g, \lambda' \\
G \text{ cond } f_m f_r l_g, \lambda, \sigma &\rightarrow_g l''_g, \lambda', \sigma' \\
\text{map } f_m l_l, \sigma &\rightarrow_{\eta_{l''_l}} \sigma'' \text{ fold } f_r l''_l, \sigma &\rightarrow_{\eta_{l'_l}} \sigma' \\
L f_m f_r l_l, \sigma &\rightarrow_{\eta_{l'_l}} \sigma' \quad \text{(MAPRED-LOCAL)} \\
\text{ch } l_g &\equiv \bar{l}_l \equiv \{l_i | l_i \subset l_g \& \cap_{l_i \in \bar{l}_l} l_i = \phi\}; \forall l_i, \sigma_i[l_i \mapsto \lambda(l)] \\
\text{agg } \bar{l}_l &\equiv l_g \equiv \bigcup_{l_i \in \bar{l}_l} l_i; \forall l_i, \lambda[l \mapsto l_i] \quad \text{(AGGREGATE)}
\end{align*}
\]
Realizing the semantics

• Code *gmap, greduce, lmap, lreduce*
  – *lmap, lreduce* use EmitLocalIntermediate() and Emit Local() 
  – Synchronized hashtables for local storage

```java
gmap(xs : X list) {
    while (no-local-convergence-intimated) {
        for each element x in xs {
            lmap(x); // emits lkey, lval
        }
        lreduce(); // operates on the output of lmap functions
    }
    for each value in lreduce-output{
        EmitIntermediate(key, value);
    }
}
```
Evaluation

• 3 applications
  – PageRank (mat-vec: eigen value and linear system solvers)
  – Single Source Shortest Path (MST, transitive closure, etc.)
  – K-Means (clustering)

• Test bed
  – 8 Amazon EC2 Large Instances
    • 64-bit compute units with 15 GB RAM, 4x 420 GB storage
    • Hadoop 0.20.1; 4 GB heap space per slave
PageRank

- Input: Partitioned using METIS ( < 10 seconds)

<table>
<thead>
<tr>
<th></th>
<th>Graph A</th>
<th>Graph B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes</td>
<td>280,000</td>
<td>100,000</td>
</tr>
<tr>
<td>Edges</td>
<td>3 million</td>
<td>3 million</td>
</tr>
</tbody>
</table>

- Damping factor = 0.85
Performance: Graph A

- **# Iterations**
  - General PageRank
  - Relaxed PageRank

- **Time (seconds)**
  - General PageRank
  - Relaxed PageRank

IEEE CLUSTER, Heraklion, Greece  23rd September 2010
Performance: Graph B

- # Iterations
  - General PageRank
  - Relaxed PageRank

- Time (seconds)
  - General PageRank
  - Relaxed PageRank

IEEE CLUSTER, Heraklion, Greece   23rd September 2010
Single Source Shortest Path

- General SSSP
- Relaxed SSSP

# Iterations vs # Partitions

Time (seconds) vs # Partitions
**K-Means**

- On US Census data
- General K-Means: 18 iterations
- Threshold: 0.01 for convergence
K-Means: Threshold

- **# Iterations**
  - **0.1**
  - **0.01**
  - **0.001**
  - **0.0001**

- **Time (seconds)**
  - **0.1**
  - **0.01**
  - **0.001**
  - **0.0001**

**Threshold - Euclidean Distance**

- **Relaxed K-Means**
- **General K-Means**
Ongoing and Future Work

• Continuation
  – Implement truly hierarchical relaxed synchronization
  – Automatically generate local map and reduce functions from the original MapReduce program

• Other optimizations
  – Speculative Parallelism
Conclusions

• MapReduce has limited applicability outside data-parallel applications
• We propose relaxed synchronization semantics for iterative MapReduce to allow asynchrony
• Semantics evaluation
  – API implementation using hashtables for local storage
• Upto 8x speed-ups achieved
Thank You!