HaLoop: Efficient Iterative Data Processing On Large Clusters

Yingyi Bu, UC Irvine
Bill Howe, UW
Magda Balazinska, UW
Michael D. Ernst, UW
Outline

- Motivation
- Caching & scheduling
- Fault-tolerance
- Programming model
- Related work
- Conclusion
- Cloud Computing Projects in UCI
Motivation

- MapReduce can’t express recursion/iteration
- Lots of interesting programs need loops
  - graph algorithms
  - clustering
  - machine learning
  - recursive queries (CTEs, datalog, WITH clause)
- Dominant solution: Use a driver program outside of MapReduce
- Hypothesis: making MapReduce loop-aware affords optimization
  - lays a foundation for scalable implementations of recursive languages
Example 1: PageRank

Rank Table $R_0$

<table>
<thead>
<tr>
<th>url</th>
<th>rank</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://www.a.com">www.a.com</a></td>
<td>1.0</td>
</tr>
<tr>
<td><a href="http://www.b.com">www.b.com</a></td>
<td>1.0</td>
</tr>
<tr>
<td><a href="http://www.c.com">www.c.com</a></td>
<td>1.0</td>
</tr>
<tr>
<td><a href="http://www.d.com">www.d.com</a></td>
<td>1.0</td>
</tr>
<tr>
<td><a href="http://www.e.com">www.e.com</a></td>
<td>1.0</td>
</tr>
</tbody>
</table>

Linkage Table $L$

<table>
<thead>
<tr>
<th>url_src</th>
<th>url_dest</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://www.a.com">www.a.com</a></td>
<td><a href="http://www.b.com">www.b.com</a></td>
</tr>
<tr>
<td><a href="http://www.a.com">www.a.com</a></td>
<td><a href="http://www.c.com">www.c.com</a></td>
</tr>
<tr>
<td><a href="http://www.c.com">www.c.com</a></td>
<td><a href="http://www.a.com">www.a.com</a></td>
</tr>
<tr>
<td><a href="http://www.e.com">www.e.com</a></td>
<td><a href="http://www.c.com">www.c.com</a></td>
</tr>
<tr>
<td><a href="http://www.d.com">www.d.com</a></td>
<td><a href="http://www.b.com">www.b.com</a></td>
</tr>
<tr>
<td><a href="http://www.c.com">www.c.com</a></td>
<td><a href="http://www.e.com">www.e.com</a></td>
</tr>
<tr>
<td><a href="http://www.e.com">www.e.com</a></td>
<td><a href="http://www.c.com">www.c.com</a></td>
</tr>
<tr>
<td><a href="http://www.a.com">www.a.com</a></td>
<td><a href="http://www.d.com">www.d.com</a></td>
</tr>
</tbody>
</table>

Rank Table $R_3$

<table>
<thead>
<tr>
<th>url</th>
<th>rank</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://www.a.com">www.a.com</a></td>
<td>2.13</td>
</tr>
<tr>
<td><a href="http://www.b.com">www.b.com</a></td>
<td>3.89</td>
</tr>
<tr>
<td><a href="http://www.c.com">www.c.com</a></td>
<td>2.60</td>
</tr>
<tr>
<td><a href="http://www.d.com">www.d.com</a></td>
<td>2.60</td>
</tr>
<tr>
<td><a href="http://www.e.com">www.e.com</a></td>
<td>2.13</td>
</tr>
</tbody>
</table>

\[
R_{i+1}(url) = \pi(url, \gamma \sum(rank)) \\
R_i.rank = R_i.rank / \gamma \cdot COUNT(url) \\
R_i.url = L.url.src
\]
PageRank Implementation on MapReduce

Join & compute rank

\[ R_i \] -> \[ M \]

Count -> \[ M \]

L-split0 -> \[ M \]

L-split1 -> \[ M \]

i = i + 1

Converged?

Aggregate

Fixpoint evaluation

\[ M \] -> \[ r \]
\[ M \] -> \[ r \]
\[ M \] -> \[ r \]

Client

done
What’s the problem?

L and Count are loop invariants, but
1. They are loaded on each iteration
2. They are shuffled on each iteration
3. Also, fixpoint evaluated as a separate MapReduce job per iteration
Example 2: Transitive Closure

*Find all transitive friends of Eric*

\[\begin{align*}
R_0 & \Rightarrow \{\text{Eric, Eric}\} \\
R_1 & \Rightarrow \{\text{Eric, Elisa}\} \\
R_2 & \Rightarrow \{\text{Eric, Tom} \quad \text{Eric, Harry}\} \\
R_3 & \Rightarrow \{\}\end{align*}\]
Transitive Closure on MapReduce

(join next generation of friends)

Friend0
Friend1

(removing the ones we’ve already seen)

Client

Anything new?

\( i = i + 1 \)
What’s the problem?

Friend is loop invariant, but
1. Friend is loaded on each iteration
2. Friend is shuffled on each iteration
Example 3: k-means

\[ k_i = k \text{ centroids at iteration } i \]

\[ k_i - k_{i+1} < \text{threshold?} \]

\[ i = i+1 \]

\[ \text{done} \]
What’s the problem?

\[ k_i = \text{k centroids at iteration i} \]

1. \text{P is loaded on each iteration}
Push loops into MapReduce!

- Architecture
- Cache loop-invariant data
- Scheduling
- Fault-tolerance
- Programming Model
HaLoop Architecture
Inter-iteration caching

Mapper input cache (MI)

Mapper output cache (MO)

Reducer input cache (RI)

Reducer output cache (RO)

Mapper output cache (MO)

Reducer input cache (RI)

Reducer output cache (RO)
RI: Reducer Input Cache

- **Provides:**
  - Access to loop invariant data without map/shuffle
- **Data:**
  - Reducer function
- **Assumes:**
  1. Static partitioning (implies: no new nodes)
  2. Deterministic mapper implementation

- **PageRank**
  - Avoid loading and shuffling the web graph at every iteration

- **Transitive Closure**
  - Avoid loading and shuffling the friends graph at every iteration

- **K-means**
  - No help
Reducer Input Cache Benefit

Friends-of-friends query

Billion Triples Dataset (120GB)

90 small instances on EC2

Overall run time
Reducer Input Cache Benefit

Friends-of-Friends query
Billion Triples Dataset (120GB)
90 small instances on EC2

Join step only

Livejournal, 12GB
Reducer Input Cache Benefit

Friends-of-friends query

Billion Triples Dataset (120GB)

90 small instances on EC2

Reduce and Shuffle of Join Step

Livejournal, 12GB
RO: Reducer Output Cache

- **Provides:**
  - Distributed access to output of previous iterations

- **Used By:**
  - Fixpoint evaluation

- **Assumes:**
  1. Partitioning constant across iterations
  2. Reducer output key functionally determines Reducer input key

- **PageRank**
  - Allows distributed fixpoint evaluation
  - Obviates extra MapReduce job

- **Transitive Closure**
  - No help

- **K-means**
  - No help
Reducer Output Cache Benefit

Fixpoint evaluation (s)

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Instances</th>
</tr>
</thead>
<tbody>
<tr>
<td>Livejournal</td>
<td>50 EC2 small instances</td>
</tr>
<tr>
<td>Freebase</td>
<td>90 EC2 small instances</td>
</tr>
</tbody>
</table>
MI: Mapper Input Cache

- **Provides:**
  - Access to non-local mapper input on later iterations
- **Data for:**
  - Map function
- **Assumes:**
  1. Mapper input does not change

- **PageRank**
  - No help
- **Transitive Closure**
  - No help
- **K-means**
  - Avoids non-local data reads on iterations > 0
Mapper Input Cache Benefit

5% non-local data reads; ~5% improvement

However, Facebook has 70% non-local data reads!!
Loop-aware Task Scheduling

**Input:** Node node, int iteration

**Global variable:** HashMap<Node, List<Partition>> last, HashMap<Node, List<Partition>> current

1: if (iteration == 0) {
2:     Partition part = StandardMapReduceSchedule(node);
3:     current.add(node, part);
4: } else {
5:     if (node.hasFullLoad()) {
6:         Node substitution = findNearbyNode(node);
7:         last.get(substitution).addAll(last.remove(node));
8:         return;
9:     }
10:     if (last.get(node).size() > 0) {
11:         Partition part = last.get(node).get(0);
12:         schedule(part, node);
13:         current.get(node).add(part);
14:         list.remove(part);
15:     }
16: }

The same as MapReduce

Find a substitution

Iteration-local Schedule
Fault-tolerance (task failures)

Cache reloading

Task failure

Task re-execution
Fault-tolerance (node failures)

Cache reloading

Task re-execution

node failure
Programming Model

- Mapper/reducer stay the same!
- Touch points
  - Input/Output: for each <iteration, step>
  - Cache filter: which tuple to cache?
  - Distance function: optional
- Nested job containing child jobs as loop body
- Minimize extra programming efforts
Related Work: Twister [Ekanayake HPDC 2010]

- Pipelining mapper/reducer
- Termination condition evaluated by main()

13. while(!complete){
14.  monitor = driver.runMapReduceBCast(cData);
15.  monitor.monitorTillCompletion();

16.  DoubleVectorData newCData = ((KMeansCombiner) driver
    .getCurrentCombiner()).getResults();
17.  totalError = getError(cData, newCData);
18.  cData = newCData;
19.  if (totalError < THRESHOLD) {
20.     complete = true;
21.     break;
22.  } break;
23.  }

$O(k)$
In Detail: PageRank (Twister)

while (!complete) {
  // start the pagerank map reduce process
  monitor = driver.runMapReduceBCast(new BytesValue(tmpCompressedDvd.getBytes()));
  monitor.monitorTillCompletion();
  // get the result of process
  newCompressedDvd = ((PageRankCombiner)
                        driver.getCurrentCombiner()).getResults();
  // decompress the compressed pagerank values
  newDvd = decompress(newCompressedDvd);
  tmpDvd = decompress(tmpCompressedDvd);
  totalError = getError(tmpDvd, newDvd);
  // get the difference between new and old pagerank values
  if (totalError < tolerance) {
    complete = true;
  }
  tmpCompressedDvd = newCompressedDvd;
}

O(N) in the size of the graph
Related Work: Spark [Zaharia HotCloud 2010]

- Reduction output collected at driver program
- “…does not currently support a grouped reduce operation as in MapReduce”

```scala
val spark = new SparkContext(<Mesos master>)
var count = spark.accumulator(0)
for (i <- spark.parallelize(1 to 10000, 10)) {
  val x = Math.random * 2 - 1
  val y = Math.random * 2 - 1
  if (x*x + y*y < 1) count += 1
}
println("Pi is roughly " + 4 * count.value / 10000.0)
```

all output sent to driver.
Related Work: Pregel [Malewicz SIGMOD 2010]

- Graphs only
  - clustering: k-means, canopy, DBScan
- Assumes each vertex has access to outgoing edges
- So an edge representation ...

\[
\text{Edge(from, to)}
\]

requires offline preprocessing
- perhaps using MapReduce
Related Work: BOOM [Alvaro EuroSys 10]

- Distributed computing based on Overlog (Datalog + temporal logic + more)
- Recursion supported naturally
  - app: API-compliant implementation of MR
Conclusions

- Relatively simple changes to MapReduce/Hadoop can support iterative/recursive programs
  - TaskTracker (Cache management)
  - Scheduler (Cache awareness)
  - Programming model (multi-step loop bodies, cache control)

- Optimizations
  - Caching reducer input realizes the largest gain
  - Good to eliminate extra MapReduce step for termination checks
  - Mapper input cache benefit inconclusive; need a busier cluster

- Future Work
  - Iteration & Recursion on top of Hyracks core!
Hyracks [Borkar et al., ICDE'11]

- Partitioned-Parallel Platform for data-intensive computing
  - Flexible (DAGs, location constraints)
  - Extensible (“micro-kernel” core, online-aggregation plugin (VLDB'11), B-tree plugin, R-tree plugin, Dataflow plugin...), an iteration/recursion plugin?

- Jobs
  - Dataflow DAG of operators and connectors
  - Can set location constraints
  - Can use a library of operators: joins, group-by, sort and so on

- V.s. “competitors”
  - V.s. Hadoop: more flexible model and less pressimistic
  - V.s. Dryad: support data as first class citizens
Questions?