Iterative MapReduce, part 2

Reza Farivar

Including slides from Yingyi Bo
Motivation

• MapReduce can’t express recursion/iteration
• Lots of interesting programs need loops
  – graph algorithms
  – clustering
  – machine learning
  – recursive queries
• Dominant solution: Use a driver program outside of MapReduce
• Hypothesis: making MapReduce loop-aware affords optimization
  – ...and lays a foundation for scalable implementations of recursive languages
Thesis – Make a Loop Framework

• **Observation:** MapReduce has proven successful as a *common runtime* for non-recursive declarative languages
  – HIVE (SQL)
  – Pig (RA with nested types)

• **Observation:** Many people roll their own loops
  – Graphs, clustering, mining, recursive queries
  – iteration managed by external script

• **Thesis:** With minimal extensions, we can provide an efficient common runtime for *recursive languages*
  – *Map, Reduce, Fixpoint*
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.d.com">www.d.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>

\[ R_{i+1} \]
\[ \pi_{url\_dest}(url\_dest) \gamma_{url\_dest}(url\_dest) \sum(rank) \]
\[ R_i.rank = R_i.rank / \gamma_{url}(url) \cdot \text{COUNT}(url) \]

\[ R_i.url = L.url\_src \]

\[ L \]
PageRank in MapReduce

Join & compute rank

L-split0
M
M
M

L-split1
M
M
M

r
r
r

r

Aggregate

r

fixpoint evaluation

M
M
M

R_i
M

i=i+1

Converged?

Client

done

Converged?
What’s the Problem?

$L$ is loop invariant, but

1. $L$ is loaded on each iteration
2. $L$ is shuffled on each iteration
3. Fixpoint evaluated as a separate MapReduce job per iteration
Example 2: Descendant Query

Find all friends within two hops of Eric

$R_0$  \{Eric, Eric\}

$R_1$  \{Eric, Elisa\}

$R_2$  \{Eric, Tom, Eric, Harry\}

$R_3$  {}
Descendant Query in MapReduce

(Compute next generation of friends)

Join

Friend0
Friend1

S_i

(M)

(r)

(M)

(M)

(M)

(Dupe-elim)

(Remove the ones we’ve already seen)

Anything new?

i = i + 1

Client

done

8
Descendant Query in MapReduce

What’s the Problem?

(Compute next generation of friends)
Join

(Si)

Friend0

Friend1

Friend is loop invariant, but

1. Friend is loaded on each iteration
2. Friend is shuffled on each iteration

(Dupe-elim)

(remove the ones we’ve already seen)
HaLoop – The Solution

HaLoop offers the following solutions to these problems:

1. A New Programming Model & Architecture for Iterative Programs
2. Loop-Aware Task Scheduling
3. Caching for Loop Invariant Data
4. Caching for Fixpoint Evaluation
HaLoop Architecture
HaLoop Architecture

Note: The loop control (i.e. determining when execution has finished) is pushed from the application into the infrastructure.
HaLoop Architecture

- HaLoop will work given the following is true:

\[ R_{i+1} = R_0 \cup (R_i \bowtie L) \]

- In other words, the next result is a join of the previous result and loop-invariant data \( L \).
HaLoop Programming Interface

- **AddMap and AddReduce**
  - Used to add a Map Reduce loop
- **SetFixedPointThreshold**
  - Set a bound on the distance between iterations
- **ResultDistance**
  - A function that returns the distance between iterations
- **SetMaxNumOfIterations**
  - Set the maximum number of iterations the loop can take.
• **SetIterationInput**
  - A function which returns the input for a certain iteration
• **AddStepInput**
  - A function will allows injection of additional data in between the Map and Reduce
• **AddInvariantTable**
  - Add a table which is loop invariant
API’s in Action – PageRank in HaLoop

**Map_Rank**
Input: Key k, Value v, int iteration
1: if v from L then
2: Output(v.url.src, v.url.dest, #1);
3: else
4: Output(v.url, v.rank, #2);
5: end if

**Reduce_Rank**
Input: Key key, Set values, Set invariantValues, int iteration
1: for url.dest in invariantValues do
2: Output(url.dest, values.get(0)/invariantValues.size());
3: end for

**Map_Aggregate**
Input: Key k, Value v, int iteration
1: Output(v.url, v.rank);

**Reduce_Aggregate**
Input: Key key, Set values, int iteration
1: Output(key, AggregateRank(values));

**ResultDistance**
Input: Key out_key, Set v_{i-1}, Set v_i
1: return |v_i.get(0) - v_{i-1}.get(0)|;

**IterationInput**
Input: int iteration
1: if iteration==1 then
2: return L ∪ R_0;
3: else
4: return R_{iteration-1}
5: end if

**Main**
1: Job job = new Job();
2: job.AddMap(Map_Rank, 1);
3: job.AddReducer(Reduce_Rank, 1);
4: job.AddMap(Map_Aggregate, 2);
5: job.AddReducer(Reduce_Aggregate, 2);
6: job.SetDistanceMeasure(ResultDistance);
7: job.AddInvariantTable(#1);
8: job.SetInput(IterationInput);
9: job.SetFixedPointThreshold(0.1);
10: job.SetMaxNumOfIterations(10);
11: job.SetReducerInputCache(true);
12: job.SetReducerOutputCache(true);
13: job.Submit();
PageRank In HaLoop

R₀

Map Rank

Reduce Rank

Map Aggregate

Reduce Aggregate

L
## PageRank In HaLoop

### Map Rank

<table>
<thead>
<tr>
<th>Source URL</th>
<th>Dest/Rank</th>
<th>Source File</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.com</td>
<td>1.0</td>
<td>#2</td>
</tr>
<tr>
<td>a.com</td>
<td>b.com,c.com, d.com</td>
<td>#1</td>
</tr>
<tr>
<td>b.com</td>
<td>1.0</td>
<td>#2</td>
</tr>
<tr>
<td>b.com</td>
<td></td>
<td>#1</td>
</tr>
<tr>
<td>c.com</td>
<td>1.0</td>
<td>#2</td>
</tr>
<tr>
<td>c.com</td>
<td>a.com, e.com</td>
<td>#1</td>
</tr>
<tr>
<td>d.com</td>
<td>1.0</td>
<td>#2</td>
</tr>
<tr>
<td>d.com</td>
<td>b.com</td>
<td>#1</td>
</tr>
<tr>
<td>e.com</td>
<td>1.0</td>
<td>#2</td>
</tr>
<tr>
<td>e.com</td>
<td>d.com,c.com</td>
<td>#1</td>
</tr>
</tbody>
</table>

### Reduce Rank

Only these values are given to reducer.
PageRank In HaLoop

<table>
<thead>
<tr>
<th>Destination</th>
<th>New Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>b.com</td>
<td>1.5</td>
</tr>
<tr>
<td>c.com</td>
<td>1.5</td>
</tr>
<tr>
<td>d.com</td>
<td>1.5</td>
</tr>
<tr>
<td>a.com</td>
<td>1.5</td>
</tr>
<tr>
<td>e.com</td>
<td>1.5</td>
</tr>
<tr>
<td>b.com</td>
<td>1.5</td>
</tr>
<tr>
<td>d.com</td>
<td>1.5</td>
</tr>
<tr>
<td>c.com</td>
<td>1.5</td>
</tr>
</tbody>
</table>
PageRank In HaLoop

<table>
<thead>
<tr>
<th>Destination</th>
<th>New Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>b.com</td>
<td>1.5</td>
</tr>
<tr>
<td>c.com</td>
<td>1.5</td>
</tr>
<tr>
<td>d.com</td>
<td>1.5</td>
</tr>
<tr>
<td>a.com</td>
<td>1.5</td>
</tr>
<tr>
<td>e.com</td>
<td>1.5</td>
</tr>
<tr>
<td>b.com</td>
<td>1.5</td>
</tr>
<tr>
<td>d.com</td>
<td>1.5</td>
</tr>
<tr>
<td>c.com</td>
<td>1.5</td>
</tr>
</tbody>
</table>
PageRank In HaLoop

```
<table>
<thead>
<tr>
<th>Destination</th>
<th>New Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.com</td>
<td>1.5</td>
</tr>
<tr>
<td>b.com</td>
<td>3.0</td>
</tr>
<tr>
<td>c.com</td>
<td>3.0</td>
</tr>
<tr>
<td>d.com</td>
<td>3.0</td>
</tr>
<tr>
<td>e.com</td>
<td>1.5</td>
</tr>
</tbody>
</table>
```
PageRank In HaLoop

**Destination** | **New Rank**
--- | ---
 a.com | 1.0  
 b.com | 1.0  
 c.com | 1.0  
 d.com | 1.0  
 e.com | 1.0

**Destination** | **New Rank**
--- | ---
 a.com | 1.5  
 b.com | 3.0  
 c.com | 3.0  
 d.com | 3.0  
 e.com | 1.5

Compare R0 and R1. If not under threshold, repeat.
PageRank In HaLoop

Map Rank

Reduce Rank

Calculate New Rank

Map Aggregate

Reduce Aggregate

<table>
<thead>
<tr>
<th>Source URL</th>
<th>Dest/Rank</th>
<th>Source File</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.com</td>
<td>1.5</td>
<td>#2</td>
</tr>
<tr>
<td>a.com</td>
<td>b.com,c.com, d.com</td>
<td>#1</td>
</tr>
<tr>
<td>b.com</td>
<td>1.5</td>
<td>#2</td>
</tr>
<tr>
<td>b.com</td>
<td></td>
<td>#1</td>
</tr>
<tr>
<td>c.com</td>
<td>1.5</td>
<td>#2</td>
</tr>
<tr>
<td>c.com</td>
<td>a.com, e.com</td>
<td>#1</td>
</tr>
<tr>
<td>d.com</td>
<td>1.5</td>
<td>#2</td>
</tr>
<tr>
<td>d.com</td>
<td>b.com</td>
<td>#1</td>
</tr>
<tr>
<td>e.com</td>
<td>1.5</td>
<td>#2</td>
</tr>
<tr>
<td>e.com</td>
<td>d.com,c.com</td>
<td>#1</td>
</tr>
</tbody>
</table>
One goal of HaLoop is to schedule map/reduce tasks on the same machine as the data.

- Scheduling the first iteration is no different than Hadoop.
- Subsequent iterations put tasks that access the same data on the same physical node.
**Task Scheduling**

Input: Node node

// The current iteration’s schedule; initially empty
Global variable: Map(Node, List(Partition)) current

// The previous iteration’s schedule
Global variable: Map(Node, List(Partition)) previous

1: if iteration == 0 then
2:    Partition part = hadoopSchedule(node);
3:    current.get(node).add(part);
4: else
5:   if node.hasFullLoad() then
6:      Node substitution = findNearestIdleNode(node);
7:      previous.get(substitution).addAll(previous.remove(node));
8:      return;
9:   end if
10:  if previous.get(node).size() > 0 then
11:    Partition part = previous.get(node).get(0);
12:    schedule(part, node);
13:  current.get(node).add(part);
14:  previous.remove(part);
15: end if
16: end if

---

**HaLoop Scheduling**

- The master node keeps a map of node ID → Filesystem Partition

- When a node becomes free, the master tries to assign a task related to data contained on that node.

- If a task is required on a node with a full load, it will utilize a nearby node.
Caching And Indexing

- Mapper Input Cache
- Reducer Input Cache
- Reducer Output Cache
- Why is there no Mapper Output Cache?
- Haloop Indexes Cached Data
  - Keys and values stored in separate local files
  - Reduces I/O seek time (forward only)
Approach: Inter-iteration caching

Mapper	
  input	
  cache	
  (MI)

Mapper	
  output	
  cache	
  (MO)

Reducer	
  input	
  cache	
  (RI)

Reducer	
  output	
  cache	
  (RO)

Loop body

Mapper output cache (MO)

Reducer output cache (RO)

Reducer input cache (RI)

Mapper input cache (MI)
RI: Reducer Input Cache

- **Provides:**
  - Access to loop invariant data without map/shuffle
- **Used By:**
  - Reducer function
- **Assumes:**
  1. Mapper output for a given table constant across iterations
  2. Static partitioning (implies: no new nodes)

- **PageRank**
  - Avoid shuffling the network at every step
- **Transitive Closure**
  - Avoid shuffling the graph at every step
- **K-means**
  - No help
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
• Provides:
  – Access to non-local mapper input on later iterations
• Used:
  – During scheduling of map tasks
• Assumes:
  1. Mapper input does not change

• PageRank
  – Subsumed by use of Reducer Input Cache
• Transitive Closure
  – Subsumed by use of Reducer Input Cache
• K-means
  – Avoids non-local data reads on iterations $> 0$
Cache Rebuilding

• Cache Must be Reconstructed:
  ▪ Hosting Node Fails
  ▪ Hosting Node has Full Node (M/R Job Needs to be Scheduled on a Different Substitution Node)

• Process is Transparent
Conclusions

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

• Optimizations
  – Caching loop invariant data realizes largest gain
  – Good to eliminate extra MapReduce step for termination checks
  – Mapper input cache benefit inconclusive; need a busier cluster

• Future Work
  – Analyze expressiveness of Map Reduce Fixpoint
  – Consider a model of Map (Reduce\(^+\)) Fixpoint
The Good ...

• Haloop extends MapReduce:
  – Easier programming of iterative algorithms
  – Efficiency improvement due to loop awareness and caching
  – Lets users reuse major building blocks from existing application implementations in Hadoop.
  – Fully backward compatible with Hadoop.
The Questionable ...

• Only useful for algorithms which can be expressed \( \tilde{R}_{i+1} = R_0 \cup (R_i \bowtie L) \)

• Imposes constraints: fixed partition function for each iteration.

• Does not improve asymptotic running time. Still \( O(M+R) \) scheduling decisions, keep \( O(M*R) \) state in memory. And more overhead…

• Not completely novel: iMapReduce and Twister.

• People still do iteration using traditional Map Reduce. Google, Nutch, Mahout…
A MapReduce Implementation of I.C.

Iterate {
  Map: for (each) $i = 1$ to $M$
  Compute();
  Reduce();
} Until converged();

- Repeated reads of constant (input) data in each iteration
- Run-time overheads in each iteration
- Intermediate Communication resulting from model updates
- Model Update Traffic
- Granularity of parallelism limited by iteration
PIC Programming Model: Best Effort Phase

- Key insight: Forgiving nature – exact numerical equivalence not required
Example 1: Page-Rank Best Effort

• **Original PageRank Phases**
  - Propagate rank values from nodes to outgoing edges
  - Re-compute rank values from incoming edges to nodes

• **PIC Best Effort**
  - Graph divided into $m$ subsets
  - $m$ node subsets and $m^2$ link subsets
  - Links matrix contains $m$ sets of intralinks and $m.(m-1)$ sets of interlinks
  - Each subproblem uses one set of vertices and associated intralinks
  - Merge
    - compute the scores for all outgoing edges
    - Update the PageRanks of the destination vertices of all outgoing edges
Example 2: Neural Network Best Effort

• Original NN Training:
  – Trains a 3 layer neural network using the back-propagation algorithm
  – Uses input dataset of 200,000 vectors for OCR
  – Model: Set of weights between neurons in the network

• PIC Best Effort:
  – Attempt is made to converge each 64MB chunk locally
  – Sub Model: similar to the model (set of weights between neurons)
  – Merge phase: averages the weights between neurons from all sub-models
  – The iterations stop when $\text{Global}_{\text{error}} < \text{Threshold}$
PIC Programming Template

**IC** (input data d, model m) {
  do {
    m = reduce(map(d,m));
  } until converged (m);
}

**kmeans_IC** (input data d, model m) {
  do {
    map: for each point in d
        emit(key: closest centroid; value: point)
    reduce: for each key
        m[key] = average(all values for key);
  } until converged {m};
}

**kmeans_PIC** (input data d, model m) {
  do {
    // Best-effort phase
    do {
      // Partition input data points,
      // copy m to each partition
      p_1 ... p_z = partition (d,m)
      IC(d_i,m_i)
      m = merge(m_1, m_2, ... , m_z);
    } until BE_converged (m);
    // Top-off phase
    IC(d,m);
  }
  // Average closest centroids from z sub-problems
  // as new centroid in merge
  reduce: m = merge(values m_1 ... m_z for key=1)
} until BE_converged (m);

// Top-off phase
kmeans_IC(d,m);
## PIC Programming API

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>abstract class PIC_Main</strong></td>
<td>This is the main entry class into the PIC library. The user has to sub-class it, create an object of its class and call the run() function to get a PIC job started.</td>
</tr>
<tr>
<td>Constructor</td>
<td>Identifies directories in HDFS where input data and model are stored, where the output models is to be stored, and whether the user is requesting the PIC library to partition the input data set. Otherwise users can partition the files themselves, (see PIC_Partitioner class).</td>
</tr>
<tr>
<td>BE_converged</td>
<td>Provides the user with the old and new best-effort iteration models, easing the user’s programming overhead.</td>
</tr>
<tr>
<td><strong>abstract class PIC_Job</strong></td>
<td>Mainly embeds the local computations.</td>
</tr>
<tr>
<td>map</td>
<td>Similar semantics to Hadoop's map(). The current model is also provided to the map() function for ease of programming. Baseline IC implementations typically need extra programming to perform this functionality.</td>
</tr>
<tr>
<td>reduce</td>
<td>Similar semantics to Hadoop's reduce().</td>
</tr>
<tr>
<td>converged</td>
<td>Provides the user with the old and new local iteration models for a convenient convergence threshold decision.</td>
</tr>
<tr>
<td>merge</td>
<td>Finds the corresponding elements of each sub-model, groups them together and passes them to the user to merge. This reduces the programmers task of identifying corresponding elements in different sub-problems.</td>
</tr>
<tr>
<td><strong>abstract class PIC_Partitioner</strong></td>
<td>This is intended for simple partitioning algorithms. The Library includes some implementation of this class, for example simple modulo partitioning and random partitioning. Alternatively the user can use more complex partitioners outside PIC and instruct PIC not to attempt partitioning.</td>
</tr>
<tr>
<td>partition</td>
<td>should return the partition number for one input point.</td>
</tr>
</tbody>
</table>
Insight into PIC Best Effort: K-means

- Fewer global iterations

<table>
<thead>
<tr>
<th></th>
<th>Baseline It.</th>
<th>Global It.</th>
<th>Max Local It.</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>31</td>
<td>3</td>
<td>33</td>
<td>6.5X</td>
</tr>
</tbody>
</table>

- Fewer Data Reads
- Reduced I/O overhead

<table>
<thead>
<tr>
<th></th>
<th>Total Baseline</th>
<th>Total Best Effort</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mapper Input</td>
<td>157 GB</td>
<td>15 GB</td>
</tr>
<tr>
<td>Mapper Output</td>
<td>286 GB</td>
<td>81 GB</td>
</tr>
<tr>
<td>Model Size</td>
<td>30 KB</td>
<td>30 KB</td>
</tr>
<tr>
<td>Reducer Output</td>
<td>959 KB</td>
<td>92 KB</td>
</tr>
</tbody>
</table>

Bar graph showing total I/O vs. number of points.
Experimental results: Performance Improvement

Small Cluster (48 cores)

Large Cluster (448 cores)
Strong Scaling on a large cluster

- Amazon Elastic MapReduce using 256 extra large instances (256 nodes, 8 cores each)
- Fixed dataset size, image smoothing application
- Result: the speedups of PIC over the baseline is maintained
- PIC library does not have negative impact on the scalability of Hadoop
Accuracy of Best Effort Results vs. Time

Neural Network Training  
K-Means Clustering  
System of Linear Equations

<table>
<thead>
<tr>
<th>Case Study</th>
<th>K-means</th>
<th>PageRank</th>
<th>Sys. Linear Equations</th>
<th>Neural Net. Training</th>
<th>Image smoothing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Measure</td>
<td>Jagotta Index</td>
<td>Top 30 pages</td>
<td>Numerical Answer</td>
<td>% of passed Vectors</td>
<td>Pixel values</td>
</tr>
<tr>
<td>Final difference</td>
<td>2%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>3.6%</td>
</tr>
</tbody>
</table>

K-Means Clustering

<table>
<thead>
<tr>
<th></th>
<th>Dataset 1</th>
<th>Dataset 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>IC K-means</td>
<td>2.109</td>
<td>2.146</td>
</tr>
<tr>
<td>PIC BE Phase K-means</td>
<td>2.112</td>
<td>2.205</td>
</tr>
<tr>
<td>Difference(%)</td>
<td>0.14%</td>
<td>2.75%</td>
</tr>
</tbody>
</table>

PageRank

<table>
<thead>
<tr>
<th>Top k results</th>
<th>PIC vs. IC (10)</th>
<th>IC (9) vs. IC (10)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>30</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>60</td>
<td>5%</td>
<td>3.3%</td>
</tr>
</tbody>
</table>
Analysis of the Best Effort Phase as a Schwartz Preconditioner

- The convergence rate of the best-effort phase can be found from the convergence rate of the baseline algorithm with a scaling factor, as follows:

\[
\left( \frac{\omega \beta}{\alpha} \right)^{\frac{k-1}{k}}
\]

- \( \beta/\alpha \) is the ratio of the maximum length of input partitioned vectors to the length of the unpartitioned vector. \( \omega \) is a measure of the converging power of the iterative function, and can be derived from the “local stability” condition, and \( k \) is the number of local iterations.

- More partitions translate to a slower convergence rate in the best-effort phase, but as we have seen earlier, the increased locality in the problems allows much faster local iterations by reducing network traffic, and performing computations locally.
Conclusions

• PIC attempts to harness the forgiving nature of iterative algorithms in a “Best Effort” phase
  – Results of the Best Effort phase are used as a starting point for the regular iterations
• Aims to converge locally before merging and checking for global convergence
  – Can increase locality of the algorithms → Faster execution
• Reduces time taken to re-distribute data (shuffle phase) and task instantiation
• PIC is implemented as a library → links to applications → Can be run on any existing Hadoop installation (e.g. Amazon Elastic MapReduce)
  – Open Source (soon)

Contact: Reza Farivar
farivar2@illinois.edu
Iterative Computations using MapReduce

<table>
<thead>
<tr>
<th></th>
<th>Haloop</th>
<th>Twister</th>
<th>Spark</th>
<th>iMapReduce</th>
<th>PrIter</th>
<th>Asynch. MapReduce</th>
<th>PIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Repeated Input Reading</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>×</td>
<td>✔</td>
</tr>
<tr>
<td>Caching input data in memory</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>×</td>
<td>✔</td>
<td>×</td>
<td>✔</td>
</tr>
<tr>
<td>Initialization overhead in each iteration</td>
<td>×</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>×</td>
<td>✔</td>
</tr>
<tr>
<td>Intermediate shuffle I/O</td>
<td>✔-</td>
<td>×</td>
<td>×</td>
<td>✔-</td>
<td>✔</td>
<td>×</td>
<td>✔</td>
</tr>
<tr>
<td>Granularity of parallelism finer than 1 iteration</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Increase the locality</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>✔-</td>
<td>✔</td>
</tr>
</tbody>
</table>