Distributed Computing using Spark
(leveraging data-parallel program structure)
The transformation of the code in program 1 to the code in program 2 is called “loop fusion.” The idea of loop fusion is to reorganize the computation to improve arithmetic intensity.
The previous example involved globally restructuring the order of computation to improve producer-consumer locality.

(improve arithmetic intensity of program)
Parallel computers in class so far

Many cores connected to a single shared memory system

Warehouse scale computing
Scale out cluster computing

- Inexpensive way to realize a high core count, high memory (in aggregate) computer
  - Made from (somewhat*) commodity Linux servers (commodity processors, networking, and storage)
  - Private per-server address space
  - Relatively low bandwidth connectivity between servers

*Cloud vendors like AWS, Google, MS Azure, Facebook make significant customizations in their datacenters.*
Typically commodity server

- DRAM (64-512 GB)
- CPU x2 (8-64 cores)
- Disks x10 (10-30 TB)

Network:
- 50 GB/s
- 50 MB/s each
- 1-4 GB/s

Top-of-Rack Switch:
- 1.4 GB/s
- 0.1-4 GB/s

Nodes:
- Same Rack: 1.4 GB/s
- Other Racks: 0.1-4 GB/s

~40 1RU servers per rack
Why write an application for a cluster?

- **Motivating problem:**
  - Consider processing 100 TB of data
  - On one node with one disk: scanning at 50 MB/s = 23 das
  - On 1000 nodes, each scanning at 50 MBs = 33 min!

- **Challenge: it can be hard to effectively utilize 1000 nodes**
  - Need to program 1000 x cores_per_node total cores
  - Have to worry about machine failures
  - Or machines that are faster or slower than others
  - It would be nice to have parallel programming frameworks that make it easier to utilize resources at this scale! *

*We've already seen programming languages/frameworks to help us with SIMD, multi-core, and GPU-based programming.
Today I need you to assume cluster storage systems exist

- If nodes can fail, how do we store data persistently?

- Modern solution: distributed storage systems
  - Provide a global namespace for files
  - Examples: Google GFX, Hadoop HDFS, Amazon S3

- Typical usage patterns
  - Huge files (100s GB to TBs)
  - Data is rarely updated in place
  - Reads and appends are common (e.g., log files)
Example: Hadoop distributed FS (HDFS)

- Global namespace
- Files split into ~200MB blocks
- Each block replicated on multiple DataNodes
- Intelligent client
  - Finds locations of blocks from NameNode; requests data from DataNode
Let’s say website gets very popular...
The log of page views gets quite large...

Assume cog.txt is a large file, stored in a distributed file system, like HDFS

Below: cluster of 4 nodes, each node with a 1 TB disk
Contents of log.txt are distributed evenly in blocks across the cluster
Imagine your professors want to know a bit more about the students visiting the 431 web site...

For example:
“What type of mobile phone are all these students using?”
“When did they first download the handout for the homework assignment due tomorrow?”
Consider a simple programming model

// this function is called once per line in input file by runtime
// input: string (line of input file)
// output: adds (user_agent, 1) entry to list
void mapper(string line, multimap<string,string>& results) {
    string user_agent = parse_requester_user_agent(line);
    if (is_mobile_client(user_agent))
        results.add(user_agent, 1);
}

// this function is called once per unique key (user_agent) in results
// values is a list of values associated with the given key
void reducer(string key, list<string> values, int& result) {
    int sum = 0;
    for (v in values)
        sum += v;
    result = sum;
}

// iterator over lines of text file
LineByLineReader input("hdfs://431log.txt");

// stores output
Writer output("hdfs://...");

// do stuff
runMapReduceJob(mapper, reducer, input, output);

(The code above computes the count of page views by each type of mobile phone.)
Let’s design an implementation of runMapReduceJob
Step 1: running the mapper function

```cpp
// called once per line in file
void mapper(string line, multimap<string,string>& results) {
    string user_agent = parse_requester_user_agent(line);
    if (is_mobile_client(user_agent))
        results.add(user_agent, 1);
}
```

```cpp
// called once per unique key in results
void reducer(string key, list<string> values, int& result) {
    int sum = 0;
    for (v in values)
        sum += v;
    result = sum;
}
```

```cpp
LineByLineReader input(“hdfs://431log.txt”);
Writer output(“hdfs://…”);
runMapReduceJob(mapper, reducer, input, output);
```

**Step 1: run mapper function on all lines of file**

**Question: How to assign work to nodes?**

**Idea 1:** use work queue for list of input blocks to process

**Dynamic assignment:** free node takes next available block

**Idea 2:** data distribution based assignment: Each node processes lines in blocks of input file that are stored locally.
Steps 2 and 3: gathering data, running the reducer

```cpp
// called once per line in file
void mapper(string line, map<string,string> results) {
    string user_agent =
        parse_requester_user_agent(line); if
        (is_mobile_client(user_agent))
        results.add(user_agent, 1);
}
// called once per unique key in results
void reducer(string key, list<string> values, int& result) {
    int sum = 0;
    for (v in values)
        sum += v;
    result = sum;
}
```

Step 2: Prepare intermediate data for reducer
Step 3: Run reducer function on all keys

Question: how to assign reducer tasks?
Question: how to get all data for key onto the correct worker node?

Keys to reduce:
(generated by mapper):
- Safari iOS
- Chrome
- Safari iWatch
- Chrome Glass

```
LineByLineReader input("hdfs://431log.txt");
Writer output("hdfs://...");
runMapReduceJob(mapper, reducer, input, output);
```
Steps 2 and 3: gathering data, running the reducer

// gather all input data for key, then execute reducer
// to produce final result
void runReducer(string key, reducer, result) {
    list<string> inputs;
    for (n in nodes) {
        filename = get_filename(key, n);
        read lines of filename, append into inputs;
    }
    reducer(key, inputs, result);
}

Step 2: Prepare intermediate data for reducer.
Step 3: Run reducer function on all keys.
Question: how to assign reducer tasks?
Question: how to get all data for key onto the correct worker node?

Keys to reduce:
(generated by mapper):

Example:
Assign Safari iOS to Node 0

Node 0

Node 1

Node 2

Node 3
Additional implementation challenges at scale

Nodes may fail during program execution

Some nodes may run slower than others (due to different amounts of work, heterogeneity in the cluster, etc..)
Job scheduler responsibilities

- **Exploit data locality: “move computation to the data”**
  - Run mapper jobs on nodes that contain input files
  - Run reducer jobs on nodes that already have most of data for a certain key

- **Handling node failures**
  - Scheduler detects job failures and reruns job on new machines
    - This is possible since inputs reside in persistent storage (distributed file system)
  - Scheduler duplicates jobs on multiple machines (reduce overall processing latency incurred by node failures)

- **Handling slow machines**
  - Scheduler duplicates jobs on multiple machines
runMapReduceJob problems?

- **Permits only a very simple program structure**
  - Programs must be structured as: map, followed by reduce by key
  - See DryadLINQ for generalization to DAGs

- **Iterative algorithms must load from disk each iteration**
  - No primitives for sharing data directly between jobs

Iterative Job:
in-memory, fault-tolerant distributed computing

http://spark.apache.org/

[Zaharia et al. NSDI 2012]
Goals

- Programming model for cluster-scale computations where there is significant reuse of intermediate datasets
  - Iterative machine learning and graph algorithms
  - Interactive data mining: load large dataset into aggregate memory of cluster and then perform multiple ad-hoc queries

- Don’t want incur inefficiency of writing intermediates to persistent distributed file system (want to keep it in memory)
  - Challenge: efficiently implementing fault tolerance for large-scale distributed in-memory computations.
Fault tolerance for in-memory calculations

- Replicate all computations
  - Expensive solution: decreases peak throughput

- Checkpoint and rollback
  - Periodically save state of program to persistent storage
  - Restart from last checkpoint on node failure

- Maintain log of updates (commands and data)
  - High overhead for maintaining logs

Recall map-reduce solutions:
- Checkpoints after each map/reduce step by writing results to file system
- Scheduler’s list of outstanding (but not yet complete) jobs is a log
- Functional structure of programs allows for restart at granularity of a single mapper or reducer invocation (don’t have to restart entire program)
Resilient distributed dataset (RDD)

Spark’s key programming abstraction:

- Read-only ordered collection of records (immutable)
- RDDs can only be created by deterministic transformations on data in persistent storage or on existing RDDs
- *Actions* on RDDs return data to application

```scala
// create RDD from file system data
val lines = spark.textFile("hdfs://431log.txt");

// create RDD using filter() transformation on lines
val mobileViews = lines.filter((x: String) => isMobileClient(x));

// another filter() transformation
val safariViews = mobileViews.filter((x: String) => x.contains("Safari"));

// then count number of elements in RDD via count() action
val numViews = safariViews.count();
```
Repeating the map-reduce example

// 1. create RDD from file system data
// 2. create RDD with only lines from mobile clients
// 3. create RDD with elements of type (String, Int) from line string
// 4. group elements by key
// 5. call provided reduction function on all keys to count views

```scala
val perAgentCounts = spark.textFile("hdfs://431log.txt")
  .filter(x => isMobileClient(x))
  .map(x => (parseUserAgent(x), 1))
  .reduceByKey((x, y) => x + y)
  .collect();
```

"Lineage": Sequence of RDD operations needed to compute output
Another Spark program

```scala
// create RDD from file system data
var lines = spark.textFile("hdfs://431log.txt");

// create RDD using filter() transformation on lines
var mobileViews = lines.filter((x: String) => isMobileClient(x));

// instruct Spark runtime to try to keep mobileViews in memory
mobileViews.persist();

// create a new RDD by filtering mobileViews
// then count number of elements in new RDD via count() action
var numViews = mobileViews.filter(_.contains("Safari")).count();

// 1. create new RDD by filtering only Chrome views
// 2. for each element, split string and take timestamp of page view
// 3. convert RDD to a scalar sequence (collect() action)
var timestamps = mobileViews.filter(_.contains("Chrome")).map(_.split(" ")(0)).collect();
```

```
431log.txt
   .textFile(...)
     .filter(isMobileClient(...))
       lines

mobileViews
   .filter(contains("Safari");
   .filter(contains("Chrome");
         .count()
    .map(split(...))
     .collect();
numViews
  .collect()
timestamps
```
**RDD transformations and actions**

**Transformations:** (data parallel operators taking an input RDD to a new RDD)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>map(f: T → U)</code></td>
<td><code>RDD[T] → RDD[U]</code></td>
</tr>
<tr>
<td><code>filter(f: T → Bool)</code></td>
<td><code>RDD[T] → RDD[T]</code></td>
</tr>
<tr>
<td><code>flatMap(f: T → Seq[U])</code></td>
<td><code>RDD[T] → RDD[U]</code></td>
</tr>
<tr>
<td><code>sample(fraction: Float)</code></td>
<td><code>RDD[T] → RDD[T]</code> (Deterministic sampling)</td>
</tr>
<tr>
<td><code>groupByKey()</code></td>
<td><code>RDD[(K, V)] → RDD[(K, Seq[V])]</code></td>
</tr>
<tr>
<td><code>reduceByKey(f: (V, V) → V)</code></td>
<td><code>RDD[(K, V)] → RDD[(K, V)]</code></td>
</tr>
<tr>
<td><code>union()</code></td>
<td><code>(RDD[T], RDD[T]) → RDD[T]</code></td>
</tr>
<tr>
<td><code>join()</code></td>
<td><code>(RDD[(K, V)], RDD[(K, W)]) → RDD[(K, (V, W))]</code></td>
</tr>
<tr>
<td><code>cogroup()</code></td>
<td><code>(RDD[(K, V)], RDD[(K, W)]) → RDD[(K, (Seq[V], Seq[W]))]</code></td>
</tr>
<tr>
<td><code>crossProduct()</code></td>
<td><code>(RDD[T], RDD[U]) → RDD[(T, U)]</code></td>
</tr>
<tr>
<td><code>mapValues(f: V → W)</code></td>
<td><code>RDD[(K, V)] → RDD[(K, W)]</code> (Preserves partitioning)</td>
</tr>
<tr>
<td><code>sort(c: Comparator[K])</code></td>
<td><code>RDD[(K, V)] → RDD[(K, V)]</code></td>
</tr>
<tr>
<td><code>partitionBy(p: Partitioner[K])</code></td>
<td><code>RDD[(K, V)] → RDD[(K, V)]</code></td>
</tr>
</tbody>
</table>

**Actions:** (provide data back to the “host” application)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>count()</code></td>
<td><code>RDD[T] → Long</code></td>
</tr>
<tr>
<td><code>collect()</code></td>
<td><code>RDD[T] → Seq[T]</code></td>
</tr>
<tr>
<td><code>reduce(f: (T, T) → T)</code></td>
<td><code>RDD[T] → T</code></td>
</tr>
<tr>
<td><code>lookup(k: K)</code></td>
<td><code>RDD[(K, V)] → Seq[V]</code> (On hash/range partitioned RDDs)`</td>
</tr>
<tr>
<td><code>save(path: String)</code></td>
<td>Outputs RDD to a storage system, e.g., HDFS</td>
</tr>
</tbody>
</table>
How do we implement RDDs?
In particular, how should they be restored?

```javascript
var lines = spark.textFile("hdfs://431log.txt");
var lower = lines.map(_.toLower());
var mobileViews = lower.filter(x => isMobileClient(x));
var howMany = mobileViews.count();
```

**Question:** should we think of RDD's like arrays?
How do we implement RDDs?

In particular, how should they be stored?

```scala
var lines = spark.textFile("hdfs://431log.txt");
var lower = lines.map(_.toLower());
var mobileViews = lower.filter(x => isMobileClient(x));
var howMany = mobileViews.count();
```

In-memory representation would be huge! (larger than original file on disk)
RDD partitioning and dependencies

```scala
val lines = spark.textFile("hdfs://431log.txt");
val lower = lines.map(_.toLower());
val mobileViews = lower.filter(x => isMobileClient(x));
val howMany = mobileViews.count();
```

Black lines show dependencies between RDD partitions.
Implementing sequence of RDD ops efficiently

```scala
var lines = spark.textFile("hdfs://431log.txt");
var lower = lines.map(_.toLower());
var mobileViews = lower.filter(x => isMobileClient(x));
var howMany = mobileViews.count();
```

Recall “loop fusion” example from opening slides of lecture

The following code stores only a line of the log file in memory, and only reads input data from disk once (“streaming” solution)

```java
int count = 0;
while (inputFile.eof()) {
    string line = inputFile.readLine();
    string lower = line.toLower;
    if (isMobileClient(lower))
        count++;
}
```
A simple interface for RDDs

```java
var lines = spark.textFile("hdfs://431log.txt");
var lower = lines.map(_.toLowerCase);
var mobileViews = lower.filter(x => isMobileClient(x));
var howMany = mobileViews.count();
```

// create RDD by mapping map_func onto input (parent) RDD
RDD::map(RDD parent, map_func) {
    return new RDDFromMap(parent, map_func);
}

// create RDD by filtering input (parent) RDD
RDD::filter(RDD parent, filter_func) {
    return new RDDFromFilter(parent, filter_func);
}

// create RDD from text file on disk
RDD::textFile(string filename) {
    return new RDDFromTextFile(open(filename));
}

// count action (forces evaluation of RDD)
RDD::count() {
    int count = 0;
    while (hasMoreElements()) {
        var el = next(); count +
    }
}

RDD::hasMoreElements() {
    parent.hasMoreElements
();
}

// overloaded since no parent exists
RDDFromTextFile::hasMoreElements() {
    return !inputFile.eof();
}

RDDFromTextFile::next() {
    return
    inputFile.readLine();
}

RDDFromMap::next() {
    var el = parent.next();
    return map_func(el);
}

RDD::filter(RDD parent, filter_func) {
    return new RDDFromFilter(parent, filter_func);
}

// create RDD by filtering input (parent) RDD
RDD::filter(RDD parent, filter_func) {
    return new RDDFromFilter(parent, filter_func);
}

// create RDD from text file on disk
RDD::textFile(string filename) {
    return new RDDFromTextFile(open(filename));
}

// count action (forces evaluation of RDD)
RDD::count() {
    int count = 0;
    while (hasMoreElements()) {
        var el = next(); count +
    }
}
```
Narrow dependencies

```
var lines = spark.textFile("hdfs://431log.txt");
var lower = lines.map(_.toLower());
var mobileViews = lower.filter(x => isMobileClient(x));
var howMany = mobileViews.count();
```

“Narrow dependencies” = each partition of parent RDD referenced by at most one child RDD partition

- Allows for fusing of operations (here: can apply map and then filter all at once on input element)
- In this example: no communication between nodes of cluster (communication of one int at end to perform count() reduction)
Wide dependencies

groupByKey: RDD[(K,V)] → RDD[(K,Seq[V)]]

“Make a new RDD where each element is a sequence containing all values from the parent RDD with the same key.”

- Wide dependencies = each partition of parent RDD referenced by multiple child RDD partitions

- **Challenges:**
  - Must compute all of RDD_A before computing RDD_B
    - Example: groupByKey() may induce all-to-all communication as shown above
  - May trigger significant recomputation of ancestor lineage upon node failure (I will address resilience in a few slides)
Cost of operations depends on partitioning

join: RDD[[(K,V)], RDD[(K,W)]] → RDD[[(K,(V,W))]]

Assume data in RDD_A and RDD_B are partitioned by key: hash username to partition id

RDD_A and RDD_B have different hash partitions: join creates wide dependencies

RDD_A and RDD_B have same hash partition: join only creates narrow dependencies
### PartitionBy() transformation

- **Inform Spark on how to partition an RDD**
  - e.g., HashPartitioner, RangePartitioner

  ```scala
  // create RDD from file system data
  var lines = spark.textFile("hdfs://431log.txt");
  var clientInfo = spark.textFile("hdfs://clientsSupported.txt"); // (useragent, "yes"/"no")

  // create RDD using filter() transformation on lines
  var mobileViews = lines.filter(x => isMobileClient(x)).map(x => parseUserAgent(x));

  // HashPartitioner maps keys to integers
  var partitioner = spark.HashPartitioner(100);

  // inform Spark of partition
  // .persist() also instructs Spark to try to keep dataset in memory
  var mobileViewPartitioned = mobileViews.partitionBy(partitioner)
    .persist();
  var clientInfoPartitioned = clientInfo.partitionBy(partitioner)
    .persist();

  // join useragents with whether they are supported or not supported
  // Note: this join only creates narrow dependencies due to the explicit partitioning above
  void joined = mobileViewPartitioned.join(clientInfoPartitioned);
  ```

- **.persist():**
  - Inform Spark this RDD's contents should be retained in memory
  - .persist(RELIABLE) = store contents in durable storage (like a checkpoint)
Scheduling Spark computations

**Stage 1 Computation**
- RDD_A part0
- RDD_A part1
- RDD_A part2

 группа́я by Key() ->
- RDD_B part0
- RDD_B part1
- RDD_B part2

**Stage 2 Computation**
- .map()
- .union()
- .join()

- RDD_C part0
- RDD_C part1

- RDD_D part0
- RDD_D part1

- RDD_F part0
- RDD_F part1
- RDD_F part2
- RDD_F part3

**Stage 3 Computation**
- RDD_G part0
- RDD_G part1
- RDD_G part2

save()

block 0
block 1
block 2

### Actions (e.g., save()) trigger evaluation of Spark lineage graph.

- **Stage 1 Computation**: do nothing since input already materialized in memory
- **Stage 2 Computation**: evaluate map in fused manner, only actually materialize RDD_F
- **Stage 3 Computation**: execute join (could stream the operation to disk, do not need to materialize )
Implementing resilience via lineage

- RDD transformations are bulk, deterministic, and func.
  - Implication: runtime can always reconstruct contents of RDD from its lineage (the sequence of transformations used to create it)
  - Lineage is a log of transformations
  - Efficient: since the log records bulk data-parallel operations, overhead of logging is low (compared to logging fine-grained operations, like in a database)

// create RDD from file system data
var lines = spark.textFile("hdfs://431log.txt");

// create RDD using filter() transformation on lines
var mobileViews = lines.filter((x: String) => isMobileClient(x));

// 1. create new RDD by filtering only Chrome views
// 2. for each element, split string and take timestamp of
//   page view (first element)
// 3. convert RDD To a scalar sequence (collect() action)
var timestamps = mobileView.filter(_.contains("Chrome"))
  .map(_.split(" ")(0));
Upon node failure: recompute lost RDD partitions from lineage

var lines = spark.textFile("hdfs://431log.txt");
var mobileViews = lines.filter((x: String) => isMobileClient(x));
var timestamps = mobileView.filter(_.contains("Chrome"))
  .map(_.split(" "))(0);

Must reload required subset of data from disk and recompute entire sequence of operations given by lineage to regenerate partitions 2 and 3 of RDD timestamps.

Note: (not shown): recall file system data is replicated so assume blocks 2 and 3 remain accessible to all nodes
Upon node failure: recompute lost RDD partitions from lineage

```scala
var lines = spark.textFile("hdfs://431log.txt");
var mobileViews = lines.filter((x: String) =>
    isMobileClient(x));
var timestamps = mobileView.filter(_.contains("Chrome"))
    .map(_.split(" "))(0);
```

Must reload required subset of data from disk and recompute entire sequence of operations given by lineage to regenerate partitions 2 and 3 of RDD timestamps.

Note: (not shown): file system data is replicated so assume blocks 2 and 3 remain accessible to all nodes.
HadoopBM = Hadoop Binary In-Memory (convert text input to binary, store in in-memory version of HDFS)

Q. Wait, the baseline parses text input in each iteration of an iterative algorithm? A. Yes. Sigh...

Anything else puzzling here?

HadoopBM’s first iteration is slow because it runs an extra Hadoop job to copy binary form of input data to in memory HDFS

Accessing data from HDFS, even if in memory, has high overhead:
- Multiple mem copies in file system + a checksum
Caution: “scale out” is not the entire story

- Distributed systems designed for cloud execution address many difficult challenges, and have been instrumental in the explosion of “big-data” computing and large-scale
  - Scale-out parallelism to many machines
  - Resiliency in the face of failures
  - Complexity of managing clusters of machines

- But “scale out” is not the whole story. At the end of the day you want good performance.

### 20 Iterations of Page Rank (iterative graph algorithm)

<table>
<thead>
<tr>
<th>Scalable System</th>
<th>Cores</th>
<th>Twitter</th>
<th>Uk-2007-05</th>
</tr>
</thead>
<tbody>
<tr>
<td>GraphChi [10]</td>
<td>2</td>
<td>3160s</td>
<td>6972s</td>
</tr>
<tr>
<td>Stratosphere [6]</td>
<td>16</td>
<td>2250s</td>
<td>-</td>
</tr>
<tr>
<td>X-Stream [17]</td>
<td>16</td>
<td>1488s</td>
<td>-</td>
</tr>
<tr>
<td>Spark [8]</td>
<td>128</td>
<td>857s</td>
<td>1759s</td>
</tr>
<tr>
<td>Giraph [8]</td>
<td>128</td>
<td>596s</td>
<td>1235s</td>
</tr>
<tr>
<td>GraphLab [8]</td>
<td>128</td>
<td>249s</td>
<td>833s</td>
</tr>
<tr>
<td>GraphX [8]</td>
<td>128</td>
<td>419s</td>
<td>462s</td>
</tr>
<tr>
<td>Single Thread (SSD)</td>
<td>1</td>
<td>300s</td>
<td>651s</td>
</tr>
<tr>
<td>Single Thread (RAM)</td>
<td>1</td>
<td>275s</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes</td>
<td>41,652,230</td>
<td>105,896,555</td>
</tr>
<tr>
<td>Edges</td>
<td>1,468,365,182</td>
<td>3,738,733,648</td>
</tr>
<tr>
<td>Size</td>
<td>5.76GB</td>
<td>14.72GB</td>
</tr>
</tbody>
</table>

Further optimization of the baseline brought time down to 110s

[“Scalability! At what COST?” McSherry et al. HotOS 2015]
Caution: “scale out” is not the entire story

**Label Propagation**  
[McSherry et al. HotOS 2015]

<table>
<thead>
<tr>
<th>scalable system</th>
<th>cores</th>
<th>twitter</th>
<th>uk-2007-05</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stratosphere [6]</td>
<td>16</td>
<td>950s</td>
<td>-</td>
</tr>
<tr>
<td>X-Stream [17]</td>
<td>16</td>
<td>1159s</td>
<td>-</td>
</tr>
<tr>
<td>Spark [8]</td>
<td>128</td>
<td>1784s</td>
<td>≤ 8000s</td>
</tr>
<tr>
<td>Giraph [8]</td>
<td>128</td>
<td>200s</td>
<td>≤ 8000s</td>
</tr>
<tr>
<td>GraphLab [8]</td>
<td>128</td>
<td>242s</td>
<td>714s</td>
</tr>
<tr>
<td>GraphX [8]</td>
<td>128</td>
<td>251s</td>
<td>800s</td>
</tr>
</tbody>
</table>

Single thread (SSD) | 1 | 153s | 417s

from McSherry 2015:

“The published work on big data systems has fetishized scalability as the most important feature of a distributed data processing platform. While nearly all such publications detail their system’s impressive scalability, few directly evaluate their absolute performance against reasonable benchmarks. To what degree are these systems truly improving performance, as opposed to parallelizing overheads that they themselves introduce?”

**COST= “Configuration that Outperforms a Single Thread”**

Perhaps surprisingly, many published systems have unbounded COST—i.e., no configuration outperforms the best single-threaded implementation—for all of the problems to which they have been applied.

**BID Data Suite (1 GPU accelerated node)**  
[Canny and Zhao, KDD13]

<table>
<thead>
<tr>
<th>System</th>
<th>Graph VxE</th>
<th>Time(s)</th>
<th>Gflops</th>
<th>Procs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hadoop</td>
<td>1.1B</td>
<td>198</td>
<td>0.015</td>
<td>50x8</td>
</tr>
<tr>
<td>Spark</td>
<td>40Mx1.5B</td>
<td>97.4</td>
<td>0.03</td>
<td>50x2</td>
</tr>
<tr>
<td>Twister</td>
<td>50Mx1.4B</td>
<td>36</td>
<td>0.09</td>
<td>60x4</td>
</tr>
<tr>
<td>PowerGraph</td>
<td>40Mx1.4B</td>
<td>3.6</td>
<td>0.8</td>
<td>64x8</td>
</tr>
<tr>
<td>BIDMat</td>
<td>60Mx1.4B</td>
<td>6</td>
<td>0.5</td>
<td>1x8</td>
</tr>
<tr>
<td>BIDMat+disk</td>
<td>60Mx1.4B</td>
<td>24</td>
<td>0.16</td>
<td>1x8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>System</th>
<th>Docs/hr</th>
<th>Gflops</th>
<th>Procs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smola[15]</td>
<td>1.6M</td>
<td>0.5</td>
<td>100x8</td>
</tr>
<tr>
<td>PowerGraph</td>
<td>1.1M</td>
<td>0.3</td>
<td>64x16</td>
</tr>
<tr>
<td>BIDMach</td>
<td>3.6M</td>
<td>30</td>
<td>1x8</td>
</tr>
</tbody>
</table>

**Page Rank**

**Latency Dirichlet Allocation (LDA)**
Performance improvements to Spark

- With increasing DRAM sizes and faster persistent storage (SSD), there is interest in improving the CPU utilization of Spark applications
  - Goal: reduce “COST”

- Efforts looking at adding efficient code generation to Spark ecosystem (e.g., generate SIMD kernels, target accelerators like GPUs, etc.) to close the gap on single node performance
  - RDD storage layouts must change to enable high-performance SIMD processing (e.g., struct of arrays instead of array of structs)
  - See Spark’s Project Tungsten, Weld [Palkar Cidr ’17], IBM’s SparkGPU

- High-performance computing ideas are influencing design of future performance-oriented distributed systems
  - Conversely: the scientific computing community has a lot to learn from the distributed computing community about elasticity and utility computing
Spark summary

- Opaque sequence abstraction (RDD) to encapsulate intermediates of cluster computations (previously... frameworks like Hadoop/MapReduce stored intermediates in the file system)
  - Observation: “files are a poor abstraction for intermediate variables in large-scale data-parallel programs”
  - RDDs are read-only, and created by deterministic data-parallel operators
  - Lineage tracked and used for locality-aware scheduling and fault-tolerance (allows recomputation of partitions of RDD on failure, rather than restore from checkpoint *)
    - Bulk operations —> overhead of lineage tracking (logging) is low

- Simple, versatile abstraction upon which many domain-specific distributed computing frameworks are being implemented.
  - See Apache Spark project: spark.apache.org

*Note that .persist(RELIABLE) allows programmer to request checkpointing in long lineage situations.*
Modern Spark ecosystem

Compelling feature: enables integration/composition of multiple domain-specific frameworks (since all collections implemented under the hood with RDDs and scheduled using Spark scheduler)

```
sqlCtx = new HiveContext(sc)
results = sqlCtx.sql("SELECT * FROM people")
names = results.map(lambda p: p.name)
```

Interleave computation and database query
Can apply transformations to RDDs produced by SQL queries

```
points = spark.textFile("hdfs://...")
  .map(parsePoint)
model = KMeans.train(points, k=10)
```

```
graph = Graph(vertices, edges)
messages = spark.textFile("hdfs://...")
graph2 = graph.joinVertices(messages) {
  (id, vertex, msg) => ...
}
```

Spark SQL

Spark MLlib

Spark GraphX

Machine learning library build on top of Spark abstractions.

GraphLab-like library built on top of Spark abstractions.