Lecture 12:

Spark
(leverage bulk-granularity program structure)

Parallel Computing
Stanford CS149, Fall 2020
Warehouse Size Cluster
Commodity Clusters

- Standard architecture:
  - Cluster of commodity Linux nodes (multicore x86)
  - Private memory ⇒ separate address spaces & separate OS
  - Ethernet network ⇒ (relatively) low bandwidth

- Cheap
  - Built from commodity processors, networks & storage
  - 1000s of nodes for < $10M

- How to organize computations on this architecture?
  - Mask issues such as load balancing and failures
Why Use A Cluster?

- Want to process 100TB of log data (1 day @Facebook)
- On 1 node: scanning @ 50MB/s = 23 days
- On 1000 nodes: scanning @ 50MB/s = 33 min
- But, very hard to utilize 1000 nodes!
  - Hard to program 16,000 cores
  - Something breaks every hour
  - Need efficient, reliable and usable framework
Commodity Cluster Node

Top-of-rack switch

Network

Nodes in other racks

0.1–2 GB/s

1–2 GB/s

Nodes in same rack

1–2 GB/s

Node 0

DRAM
128 GB–1 TB

100 GB/s

16–32 cores

1–4 GB/s

SSD
10–30 TB

Node 1000

Node 0
Storage Systems

- First order problem: if nodes can fail, how can we store data persistently?

- Answer: Distributed File System
  - Provides global file namespace
  - Google GFS, Hadoop HDFS

- Typical usage pattern
  - Huge files (100s of GB to TB)
  - Data is rarely updated in place
  - Reads and appends are common (e.g. log files)
Distributed File System

- **Chunk servers**
  - a.k.a. DataNodes in HDFS
  - File is split into contiguous chunks (usually 64-256 MB)
  - Each chunk replicated (usually 2x or 3x)
  - Try to keep replicas in different racks

- **Master node**
  - a.k.a. NameNode in HDFS
  - Stores metadata; might be replicated

- **Client library for file access**
  - Talks to master to find chunk (data) servers
  - Connects directly to chunk servers to access data
Hadoop Distributed File System (HDFS)

- **Global namespace**
- **Files broken into blocks**
  - Typically 128 MB each
  - Each block replicated on multiple DataNodes
- **Intelligent Client**
  - Client finds locations of blocks from NameNode
  - Client accesses data directly from DataNode

**HDFS Architecture**

- Metadata (Name, replicas, ...): /home/foo/data, 3, ...
- Client finds locations of blocks from NameNode
- Client accesses data directly from DataNode
Let’s say CS149 gets very popular...
A log of page views on the course web site

Stanford CS149, Fall 2020
The log of page views gets quite large...

Assume cs149log.txt is a large file, stored in a distributed file system, like HDFS

Below: cluster of 4 nodes, each node with a 10 TB disk
Contents of cs149log.txt are distributed evenly in blocks across the cluster
Imagine your professors want to know a bit more about the glut of students visiting the CS149 web site...

For example:
“What type of mobile phone are all these students using?”
Consider a simple programming model

```c++
// 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);
}

// 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://cs149log.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);
}

// 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;
}

LineByLineReader input("hdfs://cs149log.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

// 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;
}

LineByLineReader input("hdfs://cs149log.txt");
Writer output("hdfs://…");
runMapReduceJob(mapper, reducer, input, output);

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
- ...

Node 0
- Disk
  - Safari iOS values 0
  - Chrome values 0
  - cs149log.txt
  - block 0

Node 1
- Disk
  - Safari iOS values 1
  - Chrome values 1
  - cs149log.txt
  - block 2

Node 2
- Disk
  - Safari iOS values 2
  - Chrome values 2
  - cs149log.txt
  - block 4

Node 3
- Disk
  - Safari iOS values 3
  - Safari iWatch values 3
  - cs149log.txt
  - block 6
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?

Example: Assign Safari iOS to Node 0

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

Nodes:
- Node 0: CPU, Disk, Safari iWatch values 3, Chrome values 3, cs149log.txt block 7
- Node 1: CPU, Disk, Safari iOS values 2, Chrome values 2, cs149log.txt block 5
- Node 2: CPU, Disk, Safari iOS values 1, Chrome values 1, cs149log.txt block 3
- Node 3: CPU, Disk, Safari iOS values 0, Chrome values 0, cs149log.txt block 1

Stanford CS149, Fall 2020
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
  - Example graph processing:
    ```java
    void pagerank_mapper(graphnode n, map<string,string> results) {
      float val = compute update value for n
      for (dst in outgoing links from n)
        results.add(dst.node, val);
    }

    void pagerank_reducer(graphnode n, list<float> values, float& result) {
      float sum = 0.0;
      for (v in values)
        sum += v;
      result = sum;
    }

    for (i = 0 to NUM_ITERATIONS) {
      input = load graph from last iteration
      output = file for this iteration output
      runMapReduceJob(pagerank_mapper, pagerank_reducer, result[i-1], result[i]);
    }
    ```
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.
2009: Application Trends

- Despite huge amounts of data, many working sets in big data clusters fit in memory

<table>
<thead>
<tr>
<th>Memory (GB)</th>
<th>Facebook (% jobs)</th>
<th>Microsoft (% jobs)</th>
<th>Yahoo! (% jobs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>69</td>
<td>38</td>
<td>66</td>
</tr>
<tr>
<td>16</td>
<td>74</td>
<td>51</td>
<td>81</td>
</tr>
<tr>
<td>32</td>
<td>96</td>
<td>82</td>
<td>97.5</td>
</tr>
<tr>
<td>64</td>
<td>97</td>
<td>98</td>
<td>99.5</td>
</tr>
<tr>
<td>128</td>
<td>98.8</td>
<td>99.4</td>
<td>99.8</td>
</tr>
<tr>
<td>192</td>
<td>99.5</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>256</td>
<td>99.6</td>
<td>100</td>
<td>100</td>
</tr>
</tbody>
</table>

*G Ananthanarayanan, A. Ghodsi, S. Shenker, I. Stoica, "Disk-Locality in Datacenter Computing Considered Irrelevant", HotOS 2011*
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://cs149log.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
val perAgentCounts = spark.textFile("hdfs://cs149log.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

// create RDD from file system data
val lines = spark.textFile("hdfs://cs149log.txt");

// create RDD using filter() transformation on lines
val 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
val 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)
val timestamps = mobileViews.filter(_.contains("Chrome"))
  .map(_.split(" ")(0))
  .collect();
RDD transformations and actions

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

- `map(f : T ⇒ U)`: \( \text{RDD}[T] ⇒ \text{RDD}[U] \)
- `filter(f : T ⇒ \text{Bool})`: \( \text{RDD}[T] ⇒ \text{RDD}[T] \)
- `flatMap(f : T ⇒ \text{Seq}[U])`: \( \text{RDD}[T] ⇒ \text{RDD}[U] \)
- `sample(fraction : \text{Float})`: \( \text{RDD}[T] ⇒ \text{RDD}[T] \) (Deterministic sampling)
- `groupByKey()`: \( \text{RDD}[(K, V)] ⇒ \text{RDD}[(K, \text{Seq}[V])] \)
- `reduceByKey(f : (V, V) ⇒ V)`: \( \text{RDD}[(K, V)] ⇒ \text{RDD}[(K, V)] \)
- `union()`: \( \text{RDD}[T], \text{RDD}[T] ⇒ \text{RDD}[T] \)
- `join()`: \( \text{RDD}[(K, V)], \text{RDD}[(K, W)] ⇒ \text{RDD}[(K, (V, W))] \)
- `cogroup()`: \( \text{RDD}[(K, V)], \text{RDD}[(K, W)] ⇒ \text{RDD}[(K, (\text{Seq}[V], \text{Seq}[W]))] \)
- `crossProduct()`: \( \text{RDD}[T], \text{RDD}[U] ⇒ \text{RDD}[(T, U)] \)
- `mapValues(f : V ⇒ W)`: \( \text{RDD}[(K, V)] ⇒ \text{RDD}[(K, W)] \) (Preserves partitioning)
- `sort(c : \text{Comparator}[K])`: \( \text{RDD}[(K, V)] ⇒ \text{RDD}[(K, V)] \)
- `partitionBy(p : \text{Partitioner}[K])`: \( \text{RDD}[(K, V)] ⇒ \text{RDD}[(K, V)] \)

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

- `count()`: \( \text{RDD}[T] ⇒ \text{Long} \)
- `collect()`: \( \text{RDD}[T] ⇒ \text{Seq}[T] \)
- `reduce(f : (T, T) ⇒ T)`: \( \text{RDD}[T] ⇒ T \)
- `lookup(k : K)`: \( \text{RDD}[(K, V)] ⇒ \text{Seq}[V] \) (On hash/range partitioned RDDs)
- `save(path : \text{String})`: Outputs RDD to a storage system, e.g., HDFS

Stanford CS149, Fall 2020
How do we implement RDDs?
In particular, how should they be stored?

```scala
val lines = spark.textFile("hdfs://cs149log.txt");
val lower = lines.map(_.toLowerCase);
val mobileViews = lower.filter(x => isMobileClient(x));
val 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
val lines = spark.textFile("hdfs://cs149log.txt");
val lower = lines.map(_.toLowerCase);
val mobileViews = lower.filter(x => isMobileClient(x));
val 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://cs149log.txt");
val lower = lines.map(_.toLower());
val mobileViews = lower.filter(x => isMobileClient(x));
val howMany = mobileViews.count();
```

Black lines show dependencies between RDD partitions.
Review: which program performs better?

Program 1

```c
void add(int n, float* A, float* B, float* C) {
    for (int i=0; i<n; i++)
        C[i] = A[i] + B[i];
}

void mul(int n, float* A, float* B, float* C) {
    for (int i=0; i<n; i++)
        C[i] = A[i] * B[i];
}

// assume arrays are allocated here
// compute E = D + ((A + B) * C)
add(n, A, B, tmp1);
mul(n, tmp1, C, tmp2);
add(n, tmp2, D, E);
```

Two loads, one store per math op
(arithmetic intensity = 1/3)

Two loads, one store per math op
(arithmetic intensity = 1/3)

Overall arithmetic intensity = 1/3

Program 2

```c
void fused(int n, float* A, float* B, float* C, float* D, float* E) {
    for (int i=0; i<n; i++)
        E[i] = D[i] + (A[i] + B[i]) * C[i];
}

// compute E = D + (A + B) * C
fused(n, A, B, C, D, E);
```

Four loads, one store per 3 math ops
(arithmetic intensity = 3/5)

The transformation of the code in program 1 to the code in program 2 is called “loop fusion”
Review: why did we perform this transform?

Program 1

```c
int WIDTH = 1024;
int HEIGHT = 1024;
float input[(WIDTH+2) * (HEIGHT+2)];
float tmp_buf[WIDTH * (HEIGHT+2)];
float output[WIDTH * HEIGHT];
float weights[] = {1.0/3, 1.0/3, 1.0/3};

// blur image horizontally
for (int j=0; j<HEIGHT; j++) {
    for (int i=0; i<WIDTH; i++) {
        float tmp = 0.f;
        for (int ii=0; ii<3; ii++)
            tmp += input[(j+j2)*(WIDTH+2) + i+ii] * weights[ii];
        tmp_buf[j*WIDTH + i] = tmp;
    }
}

// blur tmp_buf vertically
for (int j=0; j<HEIGHT; j++) {
    for (int i=0; i<WIDTH; i++) {
        float tmp = 0.f;
        for (int jj=0; jj<3; jj++)
            tmp += tmp_buf[(j+jj)*WIDTH + i] * weights[jj];
        output[j*WIDTH + i] = tmp;
    }
}
```

Program 2

```c
int WIDTH = 1024;
int HEIGHT = 1024;
float input[(WIDTH+2) * (HEIGHT+2)];
float tmp_buf[WIDTH * (CHUNK_SIZE+2)];
float output[WIDTH * HEIGHT];
float weights[] = {1.0/3, 1.0/3, 1.0/3};

for (int j=0; j<HEIGHT; j++) {
    for (int i=0; i<WIDTH; i++) {
        float tmp = 0.f;
        for (int ii=0; ii<3; ii++)
            tmp += input[(j+j2)*(WIDTH+2) + i+ii] * weights[ii];
        tmp_buf[j*WIDTH + i] = tmp;
    }
}
```

Stanford CS149, Fall 2020
Both of the previous examples involved globally restructuring the order of computation to improve producer-consumer locality (improve arithmetic intensity of program)
Fusion with RDDs

- Why is it possible to fuse RDD transformations such as map and filter but not possible with transformations such as groupByKey and Sort?
Implementing sequence of RDD ops efficiently

```scala
val lines = spark.textFile("hdfs://cs149log.txt");
val lower = lines.map(_.toLowerCase);
val mobileViews = lower.filter(x => isMobileClient(x));
val howMany = mobileViews.count();
```

Recall “loop fusion” examples

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.toLowerCase;
    if (isMobileClient(lower))
        count++;
}
```
Narrow dependencies

“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)

```scala
val lines = spark.textFile("hdfs://cs149log.txt");
val lower = lines.map(_.toLower());
val mobileViews = lower.filter(x => isMobileClient(x));
val howMany = mobileViews.count();
```
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: $\text{RDD}[(K,V), \text{RDD}[(K,W)] \rightarrow \text{RDD}[(K,(V,W))]$

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

Stanford CS149, Fall 2020
PartitionBy() transformation

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

```scala
// create RDD from file system data
val lines = spark.textFile("hdfs://cs149log.txt");
val clientInfo = spark.textFile("hdfs://clientsSupported.txt"); // (useragent, "yes"/"no")

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

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

// inform Spark of partition
// .persist() also instructs Spark to try to keep dataset in memory
val mobileViewPartitioned = mobileViews.partitionBy(partitioner)
  .persist();
val 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)
Implementing resilience via lineage

- RDD transformations are bulk, deterministic, and functional
  - 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)

```scala
// create RDD from file system data
val lines = spark.textFile("hdfs://cs149log.txt");

// create RDD using filter() transformation on lines
val 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)
val timestamps = mobileView.filter(_.contains("Chrome")).map(_.split(" ")(0));
```
Upon node failure: recomputarec🍌 RDD partitions from lineage

val lines = spark.textFile("hdfs://cs149log.txt");
val mobileViews = lines.filter((x: String) => isMobileClient(x));
val timestamps = mobileViews.filter(_.contains("Chrome"))
  .map(_.split(" ")(0));

Must reload required subset of data from disk and recomput
entire sequence of operations given by lineage to regenerate
delicious 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

Stanford CS149, Fall 2020
val lines = spark.textFile("hdfs://cs149log.txt");
val mobileViews = lines.filter((x: String) => isMobileClient(x));
val timestamps = mobileViews.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.

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
- Conversion from serialized form to Java object
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 analytics
  - 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:

### 20 Iterations of Page Rank

<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>6072s</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>251s</td>
<td>800s</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, KDD 13]

Page Rank

<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>?x1.1B</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>

Latency Dirichlet Allocation (LDA)

<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>1x8x1</td>
</tr>
</tbody>
</table>

Stanford CS149, Fall 2020
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

- Introduces 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 allow overhead of lineage tracking (logging) to be 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)

```scala
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

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

Machine learning library build on top of Spark abstractions.

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

GraphLab-like library built on top of Spark abstractions.