Concurrency / Data Centers: MPI – MapReduce Spark (maybe)

Chris Rossbach

cs378 Fall 2018

11/14/18
Pro Forma

• Questions?
• Administrivia:
  • FPGA Lab due next Monday
  • Project plan due next Wednesday
  • Citadel visit at Systems Lunch
  • Also Next week:
    • Vijay Chidambaram on NVM and crash consistency
    • Arthur Peters on Concurrent Languages and Auto-deparallelization
  • Remaining material: Spark, NoSQL, Lock-freedom, Race Detection
  • Schedule shift for exam/presentation
• Concurrency at scale
  • MPI
  • MapReduce
  • Spark
Review: Design Space

- Internet
- Private data center
- Latency
- Throughput
- Data-parallel
- Shared memory

- Grid
- Search
- Transaction
- HPC MPI
- Spark
- Dryad
- MapReduce
Review: Distributed Memory Multiprocessors

... Or how to program 10,000,000 Cores

• Each processor has a local memory
  • Physically separated memory address space

• Processors communicate to access non-local data
  • Message communication (message passing)
    • Message passing architecture
    • Processor interconnection network

• Parallel applications partitioned across
  • Processors: execution units
  • Memory: data partitioning

• Scalable architecture
  • Incremental cost to add hardware (cost of node)

Network

- Nodes: complete computer
  • Including I/O
- Nodes communicate via network
  • Standard networks (IP)
  • Specialized networks (RDMA, fiber)
Message Passing Programming

- Defined by communication requirements
  - MPI == *Message-Passing library (Interface) specification*
    - Extended message-passing model
    - Not a language or compiler specification
    - Not a specific implementation or product
- Targeted for parallel computers, clusters, and NOWs
  - NOWs = network of workstations
- Specified in C, C++, Fortran 77, F90
- Message Passing Interface (MPI) Forum
  - [http://www.mpi-forum.org/docs/docs.html](http://www.mpi-forum.org/docs/docs.html)

- Two flavors for communication
  - Cooperative operations
  - One-sided operations
Cooperative Operations

• Data is cooperatively exchanged in message-passing
• Explicitly sent by one process and received by another
• Advantage of local control of memory
  • Change in the receiving process’s memory made with receiver’s explicit participation
• Communication and synchronization are combined

Process 0
Send(data)

Process 1
Receive(data)

Familiar argument?
One-Sided Operations

- One-sided operations between processes
  - Include remote memory reads and writes
- Only one process needs to explicitly participate
  - There is still agreement implicit in the SPMD program
- Implication:
  - Communication and synchronization are decoupled

Are 1-sided operations better for performance?
# MPI “Hello World”

```c
#include <mpi.h>
#include <stdio.h>

int main( int argc, char *argv[] )
{
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf( "I am %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

- What does this program do?

- Wait...how do we even run this?
- MPI-1 does not specify how to run an MPI program
- Starting an MPI program is dependent on implementation
  - Scripts, program arguments, and/or environment variables
- `% mpirun -np <procs> a.out`
  - For MPICH under Linux
- `mpiexec <args>`
  - Recommended part of MPI-2, as a recommendation
  - `mpiexec` for MPICH (distribution from ANL)
  - `mpirun` for SGI’s MPI
**MPI_Init**

- Hardware resources allocated
  - MPI-managed ones anyway...
- Start procs on different nodes
  - Where does their executable program come from?
- Give procs what they must know
  - Wait...what do they need to know?
- Configure OS-level resources
- Configure tools that are running with MPI
- ...

**MPI_Finalize**

- Why do we need to finalize MPI?
- What is necessary for a "graceful" MPI exit?
  - Can bad things happen otherwise?
  - Suppose one process exits...
- How do resources get de-allocated?

- By default, an error causes all processes to abort
- The user can cause routines to return (with an error code)
  - In C++, exceptions are thrown (MPI-2)
- A user can also write and install custom error handlers
- Libraries may handle errors differently from applications

**Executive Summary**

- Undo all of init
- Be able to do it on success or failure exit
Programming MPI with Only Six Functions

• Many if not all parallel programs can be written using:
  • MPI_INIT()
  • MPI_FINALIZE()
  • MPI_COMM_SIZE()
  • MPI_COMM_RANK()
  • MPI_SEND()
  • MPI_RECV()

• Point-to-point (send/recv) isn’t the only way...
if (myID == RootProcess) {
    FILE * fp = fopen(argv[1], "r");
    res = fscanf(fp, "%d\n", &nums);
    for (int p = 0; p < num_procs - 1; p++) {
        for (int i = 0; i < shard_length; i++) {
            res = fscanf(fp, "%d\n", &values[i]);
            assert(res != EOF);
        }
        // _info("ID-%d: sending shard_length:%d to ID-%d\n", RootProcess, shard_length, p+1);
        MPI_Send(&shard_length, 1, MPI_INT, p+1, stag++, MPI_COMM_WORLD);
        // _info("ID-%d: sending shard:%d to ID-%d\n", RootProcess, shard_length, p+1);
        MPI_Send(values, shard_length, MPI_INT, p+1, stag++, MPI_COMM_WORLD);
    }
    /* ... */
} else {
    MPI_Recv(&shard_length, 1, MPI_INT, RootProcess, tag, MPI_COMM_WORLD, &status);
    values = (int*)calloc(shard_length, sizeof(int));
    MPI_Recv(values, shard_length, MPI_INT, RootProcess, tag, MPI_COMM_WORLD, &status);
    mylength = shard_length;
}
int ctr=nLocalOriginal;
int offset=nLocalOriginal-nLocal;
for(i=0;i<worldSize;i++){
  if(i==rank){
    MPI_Bcast(s_particles,N_POS_ELEMS*nLocalMax+1,MPI_DOUBLE,i,MPI_COMM_WORLD);
  } else {
    MPI_Bcast(l_particles,N_POS_ELEMS*nLocalMax+1,MPI_DOUBLE,i,MPI_COMM_WORLD);
    for(k=0;k<l_particles[0];k++, ctrl++){
      if(l_particles[MASS(k)]<0){
        offset++;
        nparticles--;
      } else {
        s_particles[FX(ctr)]=l_particles[FX(k)];
        s_particles[PY(ctr)]=l_particles[PY(k)];
        s_particles[PZ(ctr)]=l_particles[PZ(k)];
        s_particles[MASS(ctr)]=l_particles[MASS(k)];
        indexes[ctr-offset]=ctr;
      }
    }
  }
}
Reduce/Allreduce

\[
\begin{array}{ccc}
A0 & B0 & C0 \\
A1 & B1 & C1 \\
A2 & B2 & C2 \\
\end{array}
\quad \rightarrow \quad \begin{array}{ccc}
A0+A1+A2 & B0+B1+B2 & C0+C1+C2 \\
\end{array}
\]

\[
\begin{array}{ccc}
A0 & B0 & C0 \\
A1 & B1 & C1 \\
A2 & B2 & C2 \\
\end{array}
\quad \rightarrow \quad \begin{array}{ccc}
A0+A1+A2 & B0+B1+B2 & C0+C1+C2 \\
A0+A1+A2 & B0+B1+B2 & C0+C1+C2 \\
A0+A1+A2 & B0+B1+B2 & C0+C1+C2 \\
\end{array}
\]
Reduce_scatter/Scan

\[
\begin{array}{ccc}
A0 & B0 & C0 \\
A1 & B1 & C1 \\
A2 & B2 & C2 \\
\end{array}
\quad \text{reduce-scatter} \quad
\begin{array}{c}
A0+A1+A2 \\
B0+B1+B2 \\
C0+C1+C2 \\
\end{array}
\]

\[
\begin{array}{ccc}
A0 & B0 & C0 \\
A1 & B1 & C1 \\
A2 & B2 & C2 \\
\end{array}
\quad \text{scan} \quad
\begin{array}{ccc}
A0 & B0 & C0 \\
A0+A1 & B0+B1 & C0+C1 \\
A0+A1+A2 & B0+B1+B2 & C0+C1+C2 \\
\end{array}
\]
MPI_Reduce

- **MPI_Reduce** (void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
  - IN sendbuf (address of send buffer)
  - OUT recvbuf (address of receive buffer)
  - IN count (number of elements in send buffer)
  - IN datatype (data type of elements in send buffer)
  - IN op (reduce operation)
  - IN root (rank of root process)
  - IN comm (communicator)

- MPI_Reduce combines elements specified by send buffer and performs a reduction operation on them.
- There are a number of predefined reduction operations: MPI_MAX, MPI_MIN, MPI_SUM, MPI_LAND, MPI_BAND, MPI_LOR, MPI_BOR, MPI_LXOR, MPI_BXOR, MPI_MAXLOC, MPI_MINLOC
MPI_Allreduce

```
int MPI::COMM_WORLD.Allreduce(
    void* operand /* in */,
    void* result /* out */,
    int count /* in */,
    MPI::Datatype datatype /* in */,
    MPI::Op operator /* in */)
```
MPI_Scan

• **MPI_Scan** (void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
  • IN  sendbuf  (address of send buffer)
  • OUT recvbuf  (address of receive buffer)
  • IN  count   (number of elements in send buffer)
  • IN  datatype (data type of elements in send buffer)
  • IN  op      (reduce operation)
  • IN  comm    (communicator)

• Note: **count** refers to total number of elements that will be received into receive buffer after operation is complete
MapReduce faux quiz (5 min, any 2):

• Have you ever written a MapReduce program? If so, what was it?
• What phenomena can slow down a map task?
• Do reducers wait for all their mappers before starting? Why/why not?
• What machine resources does the shuffle phase consume most?
• Is it safe to re-execute a failed map/reduce job sans cleanup? Why [not]?
• How does MR handle master failure? What are the alternatives?
What is GroupBy?

- Group a collection by key
- Lambda function maps elements $\rightarrow$ key

```csharp
var res = ints.GroupBy(x => x);

foreach (T elem in PF(ints))
{
    key = KeyLambda(elem);
    group = GetGroup(key);
    group.Add(elem);
}
```

Note: sorting is VERY similar
What is Join?

• Equi-join / Inner-join: “workhorse”

```java
foreach(T a in A) {
    foreach(T b in B) {
        if(joinkey(a) == joinkey(b)) {
            rs.add(joinfields(a,b));
        }
    }
}
```

• Note similarity to GroupBy
• Lots of implementations
• How to do this at scale?
Hash Join

Read entire inner relation into hash table (join attributes as key)
For each tuple from outer, look up in hash table & join

Note:
• same idea hashes data onto cluster nodes
• removes all:all data exchange
• similar effect with sorting
The Gamma Database Machine Project (1990)
The Gamma Database Machine Project (1990)
You are an engineer at: Hare-brained-scheme.com

Your boss, comes to your office and says:

“We’re going to be hog-nasty rich! We just need a program to search for strings in text files...”

Input: <search_term>, <files>
Output: list of files containing <search_term>
One solution

```java
public class StringFinder {
    int main(...) {
        foreach (File f in getInputFiles()) {
            if (f.contains(searchTerm)) {
                results.add(f.getFileName());
            }
        }
        System.out.println("Files:" + results.toString());
    }
}
```
1. How do we distribute the searchable files on our machines?
2. What if our webserver goes down?
3. What if a StringFinder machine dies? How would you know it was dead?
4. What if marketing comes and says, “well, we also want to show pictures of the earth from space too! Ooh..and the moon too!”
StringFinder was the easy part!

You really need general infrastructure.

• Many different tasks
• Want to use hundreds or thousands of PC’s
• Continue to function if something breaks
• Must be easy to program...

*Should you use Gamma?*
MapReduce

- Programming model + infrastructure
- Write programs that run on lots of machines
  - Automatic parallelization and distribution
  - Fault-tolerance
  - I/O and jobs Scheduling
  - Status and monitoring
MapReduce Programming Model

• Input & Output: sets of <key, value> pairs

• Programmer writes 2 functions:

  \[
  \text{map (in\_key, in\_value)} \rightarrow \text{list(out\_key, intermediate\_value)}
  \]
  • Processes <k,v> pairs
  • Produces intermediate pairs

  \[
  \text{reduce (out\_key, list(interm\_val))} \rightarrow \text{list(out\_value)}
  \]
  • Combines intermediate values for a key
  • Produces a merged set of outputs
Example: Counting Words...

```java
map(String input_key, String input_value):
    // input_key: document name
    // input_value: document contents
    for each word w in input_value:
        EmitIntermediate(w, "1");

reduce(String output_key, Iterator intermediate_values):
    // output_key: a word
    // output_values: a list of counts
    int result = 0;
    for each v in intermediate_values:
        result += parseInt(v);
    Emit(AsString(result));

MapReduce handles all the other details!
```
Example: Indexing (2)

```java
public void map() {
    String line = value.toString();
    StringTokenizer itr = new StringTokenizer(line);
    if(itr.countTokens() >= N) {
        while(itr.hasMoreTokens()) {
            word = itr.nextToken() + "|" + key.getFileName();
            output.collect(word, 1);
        }
    }
}
```

Input: a line of text, e.g. “mistakes were made” from `myfile.txt`

Output:

```
  mistakes | myfile.txt
  were | myfile.txt
  made | myfile.txt
```
Example: Indexing (3)

```java
public void reduce() {
    int sum = 0;
    while (values.hasNext()) {
        sum += values.next().get();
    }
    output.collect(key, sum);
}
```

Input: a <term,filename> pair, list of occurrences (e.g. {1, 1,..1})

Output:

- mistakes|myfile.txt  10
- were|myfile.txt   45
- made|myfile.txt    2
More Relatable Example: K-Means

```java
/*
 * Map: find minimum distance center for point, emit to reducer
 */
@Override
public void map(LongWritable key, Text value,
          OutputCollector<DoubleWritable, DoubleWritable> output,
          Reporter reporter) throws IOException {
    String line = value.toString();
    double point = Double.parseDouble(line);
    double min1, min2 = Double.MAX_VALUE, nearest_center = mCenters.get(0);
    // Find the minimum center from a point
    for (double c : mCenters) {
        min1 = c - point;
        if (Math.abs(min1) < Math.abs(min2)) {
            nearest_center = c;
            min2 = min1;
        }
    }
    // Emit the nearest center and the point
    output.collect(new DoubleWritable(nearest_center),
                   new DoubleWritable(point));
}
```
More Relatable Example: K-Means

```java
/*
 * Reduce: collect all points per center and calculate
 * the next center for those points
 */
@Override
public void reduce(
    DoubleWritable key, Iterator<DoubleWritable> values,
    OutputCollector<DoubleWritable, Text> output, Reporter reporter)
    throws IOException {
    double newCenter;
    double sum = 0;
    int no_elements = 0;
    String points = "";
    while (values.hasNext()) {
        double d = values.next().get();
        points = points + " " + Double.toString(d);
        sum = sum + d;
        ++no_elements;
    }

    // We have a new center now
    newCenter = sum / no_elements;

    // Emit new center and point
    output.collect(new DoubleWritable(newCenter), new Text(points));
}
```
How does parallelization work?
Implementation

• 1000s of 2 core x86, machines 2-4GM RAM
• Limited bisection bandwidth
  • Who wants to explain bisection bandwidth?
• Local IDE disks + GFS
• Scheduling: job = set of task, scheduler assigns to machines
Execution

Input

Intermediate

Group by Key

Grouped

Output
Parallel Execution
Task Granularity And Pipelining

| map tasks | >> | machines | -- why?

- Minimize fault recovery time
- Pipeline map with other tasks
- Easier to load balance dynamically
Fault Tolerance

• What failures to handle?
• How to detect failures?
• How respond?
  • For workers?
  • For master?
• How to know tasks complete?

• Worker failures:
  • Detect via heartbeat
  • Re-execute completed and in-progress map
  • Re-execute in-progress reducers (why?)

• Master failures: reexecute all!
• Task completion committed through master
Redundant Execution

• Slow worker can throttle performance: why?
• What makes a worker slow?
  • Other Jobs on machine (how could we fix)
  • Bad disks, soft errors
  • Exotica (processor caches disabled!)
• Solution: spawn backups near end of phase
The end of your career at: Hare-brained-scheme.com

Your boss, comes to your office and says:

“I can’t believe you used MapReduce!!!
You’re fired...”

Why might he say this?
MapReduce: A major step backwards | The Database Column


on Jan 17 in Database architecture, Database history, Database innovation posted by DeWitt

[Note: Although the system attributes this post to a single author, it was written by David J. DeWitt and Michael Stonebraker]

On January 8, a Database Column reader asked for our views on new distributed database research efforts, and we’ll begin here with our views on MapReduce. This is a good time to discuss it, since the recent trade press has been filled with news of the revolution of so-called “cloud computing.” This paradigm entails harnessing large numbers of (low-end) processors working in parallel to solve a computing problem. In effect, this suggests constructing a data center by lining up a large number of “jelly beans” rather than utilizing a much smaller number of high-end servers.

For example, IBM and Google have announced plans to make a 1,000 processor cluster available to a few select universities to teach students how to program such clusters using a software tool called MapReduce [1]. Berkeley has gone so far as to plan on teaching their freshman how to program using the MapReduce framework.

As both educators and researchers, we are amazed at the hype that the MapReduce proponents have spread about how it represents a paradigm shift in the development of scalable, data-intensive applications. MapReduce may be a good idea for writing certain types of general-purpose computations, but to the database community, it is:

1. A giant step backward in the programming paradigm for large-scale data intensive applications
Why is MapReduce backwards?

• Backwards step in programming paradigm
• Sub-optimal: brute force, no indexing
• Not novel: 25 year-old ideas from DBMS lit
• Missing most DBMS features
• Incompatible with most DBMS tools
What’s the problem with MR?

- Map == group-by
- Reduce == aggregate

```
SELECT job, COUNT(*) as "numemps"
FROM employees
WHERE salary > 1000
GROUP BY job;
```

- Where is the aggregate in this example?
- DBMS analog make sense? (hello, Lisp?)
Backwards programming model

• Schemas are good (what’s a schema?)
• Separation of schema from app is good (why?)
• High-level access languages are good (why?)
MapReduce is sub-optimal

- Modern DBMSs: hash + B-tree indexes to accelerate data access.
  - Indexes are user-defined
  - Could MR do this?
- No query optimizer! (oh my, terrible…but good for researchers! 😊)
- Skew: wide variance in distribution of keys
  - E.g. “the” more common than “zyzzyva”
- Materializing splits
  - N=1000 mappers → M=500 keys = 500,000 local files
  - 500 reducer instances “pull” these files
  - DBMSs push splits to sockets (no local temp files)
MapReduce: !novel

• Partitioning data sets (map) == Hash join
• Parallel aggregation == reduce
• User-supplied functions differentiates from SQL:
  • POSTGRES user functions, user aggregates
  • PL/SQL: Stored procedures
  • Object databases
MapReduce is feature-poor

Absent features:
• Bulk-loading
• Indexing
• Update operator
• Transactions
• Integrity constraints, referential integrity
• Views

Which of these are important?
Why is it OK for MR to elide them?
MapReduce incompatible with tools

- Report writers
- Business intelligence tools
- Data-mining tools
- Replication tools
- Design tools (UML, embarcadero)

How important are these?
Are these accusations fair?
MapReduce and Dataflow

- MR is a **dataflow** engine
- Lots of others
  - Dryad
  - DryadLINQ
  - Dandelion
  - CIEL
  - GraphChi/PowerGraph/Pregel
  - Spark
- Keep this in mind over next few papers
Spark faux quiz (5 min, any 2):

• What is the difference between *transformations* and *actions* in Spark?
• Spark supports a persist API. When should a programmer want to use it? When should she [not] use the “RELIABLE” flag?
• Compare and contrast fault tolerance guarantees of Spark to those of MapReduce. How are[n’t] the mechanisms different?
• Is Spark a good system for indexing the web? For computing page rank over a web index? Why [not]?
• List aspects of Spark’s design that help/hinder multi-core parallelism relative to MapReduce. If the issue is orthogonal, explain why.
Goal
Design Space
Data-Intensive Systems

```java
lines = LOAD '/user/hadoop/HDFS_File.txt' AS (line:chararray);
words = FOREACH lines GENERATE FLATTEN(TOKENIZE(line)) as word;
grouped = GROUP words BY word;
wordcount = FOREACH grouped GENERATE group, COUNT(words);
DUMP wordcount;
```

---

**Language**
- SQL
- Sawzall
- ≈SQL
- LINQ, SQL

**Execution**
- Parallel Computation Systems
- Storage
- DryadLINQ
- Dryad
- Hadoop
- HDFS
- S3
- Pig, Hive
- BigTable
- Cosmos
- HPC, Azure
- Spark

**Parallel Databases**
- SQL
- LINQ, SQL
- Sawzall

---

```
-- import the file as lines
CREATE EXTERNAL TABLE lines(line string)
LOAD DATA INPATH 'books' OVERWRITE INTO TABLE lines;

-- create a virtual view that splits the lines
SELECT word, count(*) FROM lines
LATERAL VIEW explode(split(text, ' ')) lTable as word
GROUP BY word;
```

```java
count: table sum of int;
total: table sum of float;
sum_of_squares: table sum of float;
x: float = input;
emit count ← 1;
emit total ← x;
emit sum_of_squares ← x * x;
```
MapReduce vs Dryad

- DAG instead of BSP
- Interface variety
  - Memory FIFO
  - Disk
  - Network
- Flexible Modular Composition
Dryad (2007): 2-D Piping

• Unix Pipes: 1-D
  grep | sed | sort | awk | perl

• Dryad: 2-D
  grep\(^{1000}\) | sed\(^{500}\) | sort\(^{1000}\) | awk\(^{500}\) | perl\(^{50}\)
Dataflow Engines
Virtualized 2-D Pipelines
Virtualized 2-D Pipelines
Virtualized 2-D Pipelines
Virtualized 2-D Pipelines
Virtualized 2-D Pipelines

- 2D DAG
- multi-machine
- virtualized
Dryad Job Structure

Input files
- grep
- grep
- grep

Channels
- sed
- sort
- awk
- perl

Stage
- awk
- perl

Output files
- grep
- sort

Vertices (processes)

How to implement?
Channels

Finite streams of items

- distributed filesystem files (persistent)
- SMB/NTFS files (temporary)
- TCP pipes (inter-machine)
- memory FIFOs (intra-machine)
LINQ => DryadLINQ (2008)
LINQ = .Net+ Queries

```csharp
Collection<T> collection;
bool IsLegal(Key);
string Hash(Key);

var results = from c in collection
    where IsLegal(c.key)
    select new { Hash(c.key), c.value };
```
Collections and Iterators

class Collection<T> : IEnumerable<T>;

public interface IEnumerable<T> {
    IEnumerator<T> GetEnumerator();
}

public interface IEnumerator<T> {
    T Current { get; }
    bool MoveNext();
    void Reset();
}
DryadLINQ Data Model

Partition

.Net objects

Collection
DryadLINQ = LINQ + Dryad

```csharp
Collection<T> collection;
bool IsLegal(Key k);
string Hash(Key);

var results = from c in collection
               where IsLegal(c.key)
               select new { Hash(c.key), c.value};
```

Vertex code

Query plan (Dryad job)

Data

collection

results
Language Summary

Where
Select
GroupBy
OrderBy
Aggregate
Join
Apply
Materialize
Example: Histogram

```csharp
public static IQueryable<Pair> Histogram(
    IQueryable<LineRecord> input, int k)
{
    var words = input.SelectMany(x => x.line.Split(' '));
    var groups = words.GroupBy(x => x);
    var counts = groups.Select(x => new Pair(x.Key, x.Count()));
    var ordered = counts.OrderByDescending(x => x.count);
    var top = ordered.Take(k);
    return top;
}
```

"A line of words of wisdom"

```
["A", "line", "of", "words", "of", "wisdom"]

["A", ["line"], ["of", "of"], ["words"], ["wisdom"]]

[ {"A", 1}, {"line", 1}, {"of", 2}, {"words", 1}, {"wisdom", 1}]

[{"of", 2}, {"A", 1}, {"line", 1}, {"words", 1}, {"wisdom", 1}]

[{"of", 2}, {"A", 1}, {"line", 1}]
```
Histogram Plan
public static IQueryable<S> MapReduce<T,M,K,S>(
   this IQueryable<T> input,
   Func<T, IEnumerable<M>> mapper,
   Func<M,K> keySelector,
   Func<IGrouping<K,M>,S> reducer)
{
   var map = input.SelectMany(mapper);
   var group = map.GroupBy(keySelector);
   var result = group.Select(reducer);
   return result;
}
Spark (2012) Background

• Commodity clusters: important platform
  • In industry: search, machine translation, ad targeting, ...
  • In research: bioinformatics, NLP, climate simulation, ...

• Cluster programming models (e.g. MR) de facto standard
  • Fault tolerance through replicated durable storage
  • Dataflow is the common theme

• Multi-core
• Iteration
Motivation

Current popular programming models for clusters transform data flowing from stable storage to stable storage

E.g., MapReduce:
Motivation

- Current popular programming models for clusters transform data flowing from stable storage to stable storage
- E.g., MapReduce:

**Benefits of data flow:** runtime can decide where to run tasks and can automatically recover from failures
Iterative Computations: PageRank

1. Start each page with a rank of 1
2. On each iteration, update each page’s rank to
   \[ \sum_{i \in \text{neighbors}} \frac{\text{rank}_i}{|\text{neighbors}_i|} \]

```
links = // RDD of (url, neighbors) pairs
ranks = // RDD of (url, rank) pairs

for (i <- 1 to ITERATIONS) {
    ranks = links.join(ranks).flatMap {
        (url, (links, rank)) =>
        links.map(dest => (dest, rank/links.size))
    }.reduceByKey(_ + _)
}
```
Iterative Computations: PageRank

1. Start each page with a rank of 1
2. On each iteration, update each page's rank to
   \[ \Sigma_{i \in \text{neighbors}} \frac{\text{rank}_i}{|\text{neighbors}|} \]

```
links = // RDD of (url, neighbors) pairs
ranks = // RDD of (url, rank) pairs

for (i <= 1 to ITERATIONS) {
    ranks = links.join(ranks).flatMap {
        (url, (links, rank)) =>
        links.map(dest => (dest, rank/links.size))
    }.reduceByKey(_ + _)
}
```
Motivation

• Acyclic data flow is a powerful abstraction, but is not efficient for applications that repeatedly reuse a *working set* of data:
  • **Iterative** algorithms (many in machine learning)
  • **Interactive** data mining tools (R, Excel, Python)
• Spark makes working sets a first-class concept to efficiently support these apps
Spark Goal

• Provide distributed memory abstractions for clusters to support apps with working sets

• Retain the attractive properties of MapReduce:
  • Fault tolerance (for crashes & stragglers)
  • Data locality
  • Scalability

Solution: augment data flow model with “resilient distributed datasets” (RDDs)
Generality of RDDs

• Conjecture: Spark’s data flow + RDDs unifies many proposed cluster programming models
  • General data flow models: MapReduce, Dryad, SQL
  • Specialized models for stateful apps: Pregel (BSP), HaLoop (iterative MR), Continuous Bulk Processing

• Instead of specialized APIs for one type of app, give user first-class control of distrib. datasets
Programming Model

• Resilient distributed datasets (RDDs)
  • Immutable collections partitioned across cluster that can be rebuilt if a partition is lost
  • Created by transforming data in stable storage using data flow operators (map, filter, group-by, ...)
  • Can be cached across parallel operations

• Parallel operations on RDDs
  • Reduce, collect, count, save, ...

• Restricted shared variables
  • Accumulators, broadcast variables
Example: Log Mining

• Load error messages from a log into memory, then interactively search for various patterns

```scala
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split("\t")(2))
cachedMsgs = messages.cache()
cachedMsgs.filter(_.contains("foo")).count
cachedMsgs.filter(_.contains("bar")).count
...
```

**Result:** full-text search of Wikipedia in <1 sec (vs 20 sec for on-disk data)
RDDs in More Detail

- An RDD is an immutable, partitioned, logical collection of records
  - Need not be materialized, but rather contains information to rebuild a dataset from stable storage
- Partitioning can be based on a key in each record (using hash or range partitioning)
- Built using bulk transformations on other RDDs
- Can be cached for future reuse
# RDD Operations

<table>
<thead>
<tr>
<th>Transformations (define a new RDD)</th>
<th>Parallel operations (return a result to driver)</th>
</tr>
</thead>
<tbody>
<tr>
<td>map</td>
<td>reduce</td>
</tr>
<tr>
<td>filter</td>
<td>collect</td>
</tr>
<tr>
<td>sample</td>
<td>count</td>
</tr>
<tr>
<td>union</td>
<td>save</td>
</tr>
<tr>
<td>groupByKey</td>
<td>lookupKey</td>
</tr>
<tr>
<td>reduceByKey</td>
<td>...</td>
</tr>
<tr>
<td>join</td>
<td>...</td>
</tr>
<tr>
<td>persist/cache</td>
<td>...</td>
</tr>
</tbody>
</table>

- Where
- Select
- GroupBy
- OrderBy
- Aggregate
- Join
- Apply
- Materialize
RDDs maintain *lineage* information that can be used to reconstruct lost partitions

Ex:
```scala
cachedMsgs = textFile(...).filter(_.contains("error"))
  .map(_.split('t')(2))
  .persist()
```

![Diagram showing the flow of RDDs from HdfsRDD through FilteredRDD and MappedRDD to CachedRDD](image)
Benefits of RDD Model

• Consistency is easy due to immutability
• Inexpensive fault tolerance (log lineage rather than replicating/checkpointing data)
• Locality-aware scheduling of tasks on partitions
• Despite being restricted, model seems applicable to a broad variety of applications
# RDDs vs Distributed Shared Memory

<table>
<thead>
<tr>
<th>Concern</th>
<th>RDDs</th>
<th>Distr. Shared Mem.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reads</td>
<td>Fine-grained</td>
<td>Fine-grained</td>
</tr>
<tr>
<td>Writes</td>
<td>Bulk transformations</td>
<td>Fine-grained</td>
</tr>
<tr>
<td>Consistency</td>
<td>Trivial (immutable)</td>
<td>Up to app / runtime</td>
</tr>
<tr>
<td>Fault recovery</td>
<td>Fine-grained and low-overhead using lineage</td>
<td>Requires checkpoints and program rollback</td>
</tr>
<tr>
<td>Straggler mitigation</td>
<td>Possible using speculative execution</td>
<td>Difficult</td>
</tr>
<tr>
<td>Work placement</td>
<td>Automatic based on data locality</td>
<td>Up to app (but runtime aims for transparency)</td>
</tr>
</tbody>
</table>
Example: Logistic Regression

• Goal: find best line separating two sets of points
Logistic Regression Code

• val data = spark.textFile(...).map(readPoint).persist()

• var w = Vector.random(D)

• for (i <- 1 to ITERATIONS) {
  • val gradient = data.map(p =>
    • (1 / (1 + exp(-p.y*(w dot p.x))) - 1) * p.y * p.x
  • ).reduce(_ + _)
  • w -= gradient
• }

• println("Final w: " + w)
Logistic Regression Performance

- First iteration: 174 s
- Further iterations: 6 s
- Average iteration time: 127 s
Example: MapReduce

- MapReduce data flow can be expressed using RDD transformations

```scala
res = data.flatMap(rec => myMapFunc(rec))
    .groupByKey()
    .map((key, vals) => myReduceFunc(key, vals))
```

Or with combiners:

```scala
res = data.flatMap(rec => myMapFunc(rec))
    .reduceByKeyByKey(myCombiner)
    .map((key, val) => myReduceFunc(key, val))
```
Word Count in Spark

val lines = spark.textFile("hdfs://...")

val counts = lines.flatMap(_.split("\s"))
  .reduceByKey(_ + _)

counts.save("hdfs://...")
Example: Pregel

• Graph processing framework from Google that implements Bulk Synchronous Parallel model
• Vertices in the graph have state
• At each superstep, each node can update its state and send messages to nodes in future step
• Good fit for PageRank, shortest paths, ...
Pregel Data Flow

Input graph

Vertex state 1

Messages 1

Superstep 1

Vertex state 2

Messages 2

Superstep 2

...
PageRank in Pregel

Input graph

Vertex ranks 1

Contributions 1

Superstep 1 (add contribs)

Vertex ranks 2

Contributions 2

Superstep 2 (add contribs)

Group & add by vertex

Group & add by vertex

...
Pregel in Spark

• Separate RDDs for immutable graph state and for vertex states and messages at each iteration
• Use groupByKey to perform each step
• Cache the resulting vertex and message RDDs
• Optimization: co-partition input graph and vertex state RDDs to reduce communication
Other Spark Applications

• Twitter spam classification
• EM alg. for traffic prediction (Mobile Millennium)
• K-means clustering
• Alternating Least Squares matrix factorization
• In-memory OLAP aggregation on Hive data
• SQL on Spark (future work)
Overview

• Spark runs on the Mesos cluster manager [NSDI 11], letting it share resources with Hadoop & other apps

• Can read from any Hadoop input source (e.g. HDFS)

• ~6000 lines of Scala code thanks to building on Mesos
Language Integration

• Scala closures are Serializable Java objects
  • Serialize on driver, load & run on workers

• Not quite enough
  • Nested closures may reference entire outer scope
  • May pull in non-Serializable variables not used inside
  • Solution: bytecode analysis + reflection

• Shared variables implemented using custom serialized form (e.g. broadcast variable contains pointer to BitTorrent tracker)
Interactive Spark

- Modified Scala interpreter to allow Spark to be used interactively from the command line

- Required two changes:
  - Modified wrapper code generation so that each “line” typed has references to objects for its dependencies
  - Place generated classes in distributed filesystem

- Enables in-memory exploration of big data