Distributed Systems

15-640 (Section B)
Fall 2017
14 – Data Intensive Computing + MapReduce
Announcements

• P2 Release (PAXOS): release 10/28
  • Individual Project

• Mid Term Pickup and Regrade Request
  • Andrew ID starts from “a”–“n”, Lisa or Victoria (WEAN 5125)
    • Wednesday 3-5pm. Friday 9-11am, 3-5pm. Monday 9-12, 1-5pm.
  • Andrew ID Starts from “o” – ”z”, Erin (GHC 7123)
    • Tuesday – Friday -- 8:30am-10:30am, 11:15-1:15pm, 2:30 – 4:30pm
  • All Regrade requests must be submitted in writing, explaining why you think you deserve more points, latest by Nov 3rd
  • Midterm solutions will be posted Wednesday / Thursday
Today’s Topics

• Super computers
  • Traditional high-performance computing (HPC)

• Cluster computing
  • MapReduce
  • Implementation

• Alternatives to MapReduce
Typical HPC Machine

- Compute Nodes
  - High end processor(s)
  - Lots of RAM
- Network
  - Specialized
  - Very high performance
- Storage Server
  - RAID-based disk array
HPC Machine Example

Sunway TaihuLight

- Cores: 10,649,600
- Memory: 1,310,720 GB
- Architecture: Sunway SW26010
  - No caches, 65 cores / on-chip group @ 1.45 GHz
  - Sunway RaiseOS 2.0.5
- 93,014.6 TFlop/s
HPC Programming Model

- Programs described at very low level
  - Specify detailed control of processing & communications
- Rely on small number of software packages
  - Written by specialists
  - Limits classes of problems & solution methods
Bulk Synchronous Programming

• Solving Problem Over Grid
  • E.g., finite-element computation

• Partition into Regions
  • p regions for p processors

• Map Region per Processor
  • Local computation sequential
  • Periodically communicate boundary values with neighbors
Typical HPC Operation

- **Characteristics**
  - Long-lived processes
  - Make use of spatial locality
  - Hold all program data in memory (no disk access)
  - High bandwidth communication

- **Strengths**
  - High utilization of resources
  - Effective for many scientific applications

- **Weaknesses**
  - Requires careful tuning of application to resources
  - Intolerant of any variability
HPC Fault Tolerance

- **Checkpoint**
  - Periodically store state of all processes
  - Significant I/O traffic

- **Restore**
  - When failure occurs
  - Reset state to that of last checkpoint
  - All intervening computation wasted

- **Performance Scaling**
  - Very sensitive to number of failing components
Today’s Topics

• Super computers
  • Traditional high-performance computing (HPC)
• Cluster computing
  • MapReduce
  • Implementation
• Alternatives to MapReduce
Google Data Centers

- Dalles, Oregon
- Hydroelectric power @ 2¢ / KW Hr
- $600M, 50 Megawatts
- Enough to power 60,000 homes

- Engineered for maximum modularity & power efficiency
- Container: 1160 servers, 250KW
- Server: 2 disks, 2 processors
Typical Cluster Machine

- Collocate Compute + Storage
  - Medium-performance processors
  - Modest memory
  - 1-2 disks
- Network
  - Conventional Ethernet switches
  - 10s-100 Gb/s
Machines with Disks

• Lots of storage for cheap
  • 3 TB @ $150
    (5¢ / GB)
  • Compare 2007:
    0.75 TB @ $266
    35¢ / GB

• Drawbacks
  • Long and highly variable delays
  • Not very reliable

• HPC: no local disks
Oceans of Data, Skinny Pipes

- 1 Terabyte
  - Easy to store
  - Hard to move

<table>
<thead>
<tr>
<th>Disks</th>
<th>MB / s</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seagate Barracuda</td>
<td>115</td>
<td>2.3 hours</td>
</tr>
<tr>
<td>Seagate Cheetah</td>
<td>125</td>
<td>2.2 hours</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Networks</th>
<th>MB / s</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Home Internet</td>
<td>&lt; 16</td>
<td>&gt; 1 day</td>
</tr>
<tr>
<td>Gigabit Ethernet</td>
<td>&lt; 125</td>
<td>&gt; 2.2 hours</td>
</tr>
<tr>
<td>PSC Teragrid Connection</td>
<td>&lt; 3,750</td>
<td>&gt; 4.4 minutes</td>
</tr>
</tbody>
</table>
Data-Intensive System Challenge

• For Computation That Accesses 1 TB in 5 minutes
  • Data distributed over 100+ disks
    • Assuming uniform data partitioning
  • Compute using 100+ processors
  • Connected by gigabit Ethernet (or equivalent)

• System Requirements
  • Lots of disks
  • Lots of processors
  • Located in close proximity
    • Within reach of fast, local-area network
Cluster Programming Model

• Application programs written in terms of high-level operations on data
• Runtime system controls scheduling, load balancing, ...

• This is idealized. In practice, no general cluster programming model.
Map/Reduce Cluster Model

- Map computation across many objects
- Flexible aggregation of results
- System solves resource allocation & reliability

Dean & Ghemawat: “MapReduce: Simplified Data Processing on Large Clusters”, OSDI 2004
MapReduce Example

- Calculate word frequency of set of documents
MapReduce Example

- **Map:** generate \( \langle \text{word, count} \rangle \) pairs for all words in document
- **Reduce:** sum word counts across documents

\[
\begin{align*}
\text{dick} & \quad \sum 1 \\
\text{and} & \quad \sum 3 \\
\text{come} & \quad \sum 6 \\
\text{see} & \quad \sum 3 \\
\text{spot} & \quad \sum 1 \\
\end{align*}
\]

\[
\begin{align*}
\langle \text{dick, 1} \rangle & \quad \langle \text{come, 1} \rangle & \quad \langle \text{see, 1} \rangle & \quad \langle \text{spot, 1} \rangle \\
\langle \text{come, 1} \rangle & \quad \langle \text{and, 1} \rangle \\
\langle \text{come, 1} \rangle & \quad \langle \text{see, 1} \rangle \\
\langle \text{come, 2} \rangle & \quad \langle \text{and, 1} \rangle \\
\langle \text{come, 1} \rangle & \quad \langle \text{and, 1} \rangle \\
\langle \text{and, 1} \rangle & \quad \langle \text{see, 1} \rangle \\
\langle \text{and, 1} \rangle & \quad \langle \text{spot, 1} \rangle \\
\end{align*}
\]

- Extract Word-Count Pairs
- Sum
Hadoop Project

- File system with files distributed across nodes

- Store multiple (typically 3 copies of each file)
  - If one node fails, data still available

- Logically, any node has access to any file
  - May need to fetch across network (ideally, leverage locality for perf.)

- Map / Reduce programming environment
  - Software manages execution of tasks on nodes
Hadoop MapReduce API

• Requirements
  • Programmer must supply Mapper & Reducer classes

• Mapper
  • Steps through file one line at a time
  • Code generates sequence of <key, value> pairs
    • Call output.collect(key, value)
  • Default types for keys & values are strings
    • Lots of low-level machinery to convert to & from other data types
    • But can use anything “writable”

• Reducer
  • Given key + iterator that generates sequence of values
  • Generate one or more <key, value> pairs
    • Call output.collect(key, value)
public class WordCountMapper extends MapReduceBase
    implements Mapper {

    private final static Text word = new Text();

    private final static IntWritable count = new IntWritable(1);

    public void map(WritableComparable key, Writable values,
        OutputCollector output, Reporter reporter)
        throws IOException {
        /* Get line from file */
        String line = values.toString();
        /* Split into tokens */
        StringTokenizer itr = new StringTokenizer(line.toLowerCase(),
            " \	!.?:()[]',&-;|0123456789");
        while(itr.hasMoreTokens()) {
            word.set(itr.nextToken());
            /* Emit <token,1> as key + value */
            output.collect(word, count);
        }
    }
}
public class WordCountReducer extends MapReduceBase
    implements Reducer {

    public void reduce(WritableComparable key, Iterator values,
        OutputCollector output, Reporter reporter)
        throws IOException {
        int cnt = 0;
        while(values.hasNext()) {
            IntWritable ival = (IntWritable) values.next();
            cnt += ival.get();
        }
        output.collect(key, new IntWritable(cnt));
    }
}
Cluster Scalability Advantages

- Application of distributed system design principles
- Dynamically scheduled tasks with state in replicated files

Provisioning Advantages
- Can use consumer-grade components
  - maximizes cost-performance
- Can have heterogeneous nodes
  - More efficient technology refresh

Operational Advantages
- Minimal staffing
- No downtime
Example: Sparse-matrix Product

• Task: Compute product $C = A \cdot B$
• Assume most matrix entries are 0

• Motivation
  • Core problem in scientific computing
  • Challenging for parallel execution
  • Demonstrate expressiveness of Map/Reduce

\[
\begin{pmatrix}
10 & 20 \\
30 & 40 \\
50 & 60 & 70
\end{pmatrix}
\times
\begin{pmatrix}
-2 & -3 \\
-4
\end{pmatrix}
=
\begin{pmatrix}
-60 & -250 \\
-170 & -460
\end{pmatrix}
\]
Representing Sparse Matrices

- Represent matrix as a list of nonzero entries
  \[ \langle \text{row}, \text{col}, \text{value}, \text{matrixID} \rangle \]

- Strategy
  - Phase 1: Compute all products \( a_{i,k} \cdot b_{k,j} \)
  - Phase 2: Sum products for each entry \( i,j \)
  - Each phase involves a Map/Reduce
Phase 1 Map

- Group values $a_{i,k}$ and $b_{k,j}$ according to key $k$
Phase 1 Reduce

- Generate all products $a_{i,k} \cdot b_{k,j}$
Phase 2 Map

- Group products $a_{i,k} \cdot b_{k,j}$ with matching values of $i$ and $j$

Key = row,col
Phase 2 Reduce

- Sum products to get final entries
public class P1Mapper extends MapReduceBase implements Mapper {
    public void map(WritableComparable key, Writable values, OutputCollector output, Reporter reporter) throws IOException {
        try {
            GraphEdge e = new GraphEdge(values.toString());
            IntWritable k;
            if (e.tag.equals("A"))
                k = new IntWritable(e.toNode);
            else
                k = new IntWritable(e.fromNode);
            output.collect(k, new Text(e.toString()));
        } catch (BadGraphException e) {}
public class P1Reducer extends MapReduceBase implements Reducer {

    public void reduce(WritableComparable key, Iterator values, 
                        OutputCollector output, Reporter reporter) 
            throws IOException 
    {
        Text outv = new Text("""); // Don't really need output values

        /* First split edges into A and B categories */
        LinkedList<GraphEdge> alist = new LinkedList<GraphEdge>();
        LinkedList<GraphEdge> blist = new LinkedList<GraphEdge>();
        while(values.hasNext) {
            try {
                GraphEdge e =
                    new GraphEdge(values.next().toString());
                if (e.tag.equals("A")) {
                    alist.add(e);
                } else {
                    blist.add(e);
                }
            } catch (BadGraphException e) {}
        }

        // Continued
// Continued
}
Phase 1 Reduce (Code) cntd.

// Continuation

Iterator<GraphEdge> aset = alist.iterator();
// For each incoming edge
while (aset.hasNext()) {
    GraphEdge aedge = aset.next();
    // For each outgoing edge
    Iterator<GraphEdge> bset = blist.iterator();
    while (bset.hasNext()) {
        GraphEdge bedge = bset.next();
        GraphEdge newe = aedge.contractProd(bedge);
        // Null would indicate invalid contraction
        if (newe != null) {
            Text outk = new Text(newe.toString());
            output.collect(outk, outv);
        }
    }
}
}
public class P2Mapper extends MapReduceBase implements Mapper {

    public void map(WritableComparable key, Writable values, OutputCollector output, Reporter reporter)
    throws IOException {
        String es = values.toString();
        try {
            GraphEdge e = new GraphEdge(es);
            // Key based on head & tail nodes
            String ks = e.fromNode + " " + e.toNode;
            output.collect(new Text(ks), new Text(e.toString()));
        } catch (BadGraphException e) {}}
}

public class P2Reducer extends MapReduceBase implements Reducer {

    public void reduce(WritableComparable key, Iterator values,
                        OutputCollector output, Reporter reporter)
                        throws IOException {

        GraphEdge efinal = null;
        while (efinal == null && values.hasNext()) {
            try {
                efinal = new GraphEdge(values.next().toString());
            } catch (BadGraphException e) {
            }
        }
        if (efinal != null) {
            while(values.hasNext()) {
                try {
                    GraphEdge eother =
                    new GraphEdge(values.next().toString());
                    efinal.weight += eother.weight;
                } catch (BadGraphException e) {
                }
            }
            if (efinal.weight != 0)
                output.collect(new Text(efinal.toString()),
                                new Text('"'"));
        }
    }
}
Lessons from Sparse Matrix

• Associative Matching is Powerful Communication Primitive
  • Intermediate step in Map/Reduce

• Similar Strategy Applies to Other Problems
  • Shortest path in graph
  • Database join

• Many Performance Considerations
  • Kiefer, Volk, Lehner, TU Dresden
  • Should do systematic comparison to other sparse matrix implementations
MapReduce Implementation

• Built on Top of Parallel File System
  • Google: GFS, Hadoop: HDFS
  • Provides global naming
  • Reliability via replication (typically 3 copies)

• Breaks work into tasks
  • Master schedules tasks on workers dynamically
  • Typically #tasks >> #processors

• Net Effect
  • Input: Set of files in reliable file system
  • Output: Set of files in reliable file system
MapReduce Execution

Dean & Ghemawat: “MapReduce: Simplified Data Processing on Large Clusters”, OSDI 2004
Mapping

- **Hash Function** $h$
  - Maps each key $K$ to integer $i$ such that $0 \leq i < R$

- **Mapper Operation**
  - Reads input file blocks
  - Generates pairs $\langle K, V \rangle$
  - Writes to local file $h(K)$

\[ h(K) \in \{0, \ldots, R-1\} \]
• Dynamically map input file blocks onto mappers
• Each generates key/value pairs from its blocks
• Each writes R files on local file system
Shuffling

• Each Reducer:
  • Handles 1/R of the possible key values
  • Fetches its file from each of M mappers
  • Sorts all of its entries to group values by keys
Reducing

• Each Reducer:
  • Executes reducer function for each key
  • Writes output values to parallel file system
MapReduce Effect

MapReduce Step
- Reads set of files from file system
- Generates new set of files
- Can iterate to do more complex processing

Input Files (Partitioned into Blocks)

R Output Files
Map/Reduce Operation

- Characteristics
  - Computation broken into many, short-lived tasks
    - Mapping, reducing
  - Use disk storage to hold intermediate results

- Strengths
  - Great flexibility in placement, scheduling, and load balancing
  - Can access large data sets

- Weaknesses
  - Higher overhead
  - Lower raw performance
Example Parameters

• Sort Benchmark
  • $10^{10}$ 100-byte records
  • Partition into $M = 15,000$ 64MB pieces
    • Key = value
    • Partition according to most significant bytes
  • Sort locally with $R = 4,000$ reducers

• Machine
  • 1800 2Ghz Xeons
  • Each with 2 160GB IDE disks
  • Gigabit ethernet
  • 891 seconds total
Interesting Features

• Fault Tolerance
  • Assume reliable file system
  • Detect failed worker
    • Heartbeat mechanism
  • Reschedule failed task

• Stragglers
  • Tasks that take long time to execute
  • Might be bug, flaky hardware, or poor partitioning
  • When done with most tasks, reschedule any remaining executing tasks
    • Keep track of redundant executions
    • Significantly reduces overall run time
Map/Reduce Fault Tolerance

• Data Integrity
  • Store multiple copies of each file
  • Including intermediate results of each Map / Reduce
    • Continuous checkpointing

• Recovering from Failure
  • Simply recompute lost result
    • Localized effect
  • Dynamic scheduler keeps all processors busy
Exploring Parallel Computation Models

- Map/Reduce Provides Coarse-Grained Parallelism
  - Computation done by independent processes
  - File-based communication

- Observations
  - Relatively “natural” programming model
  - Research issue to explore full potential and limits
Beyond Map/Reduce

• Typical Map/Reduce Applications
  • Sequence of steps, each requiring map & reduce
  • Series of data transformations
  • Iterating until reach convergence

• Strengths of Map/Reduce
  • User writes simple functions, system manages complexities of mapping, synchronization, fault tolerance
  • Very general
  • Good for large-scale data analysis

• Limitations
  • No locality of data or activity
  • Each map/reduce step must complete before next begins
Conclusions

• Distributed Systems Concepts Lead to Scalable Machines
  • Loosely coupled execution model
  • Lowers cost of procurement & operation

• Map/Reduce Gaining Widespread Use
  • Hadoop makes it widely available
  • Great for some applications, good enough for many others

• Lots of Work to be Done
  • Richer set of programming models and implementations
  • Expanding range of applicability
    • Problems that are data and compute intensive
    • The future of supercomputing?
Beyond Map/Reduce
Generalizing Map/Reduce

• Microsoft Dryad Project

• Computational Model
  • Acyclic graph of operators
    • But expressed as textual program
  • Each takes collection of objects and produces objects
    • Purely functional model

• Implementation Concepts
  • Objects stored in files or memory
  • Any object may be lost; any operator may fail
  • Replicate & recompute for fault tolerance
  • Dynamic scheduling
    • # Operators >> # Processors
CMU GraphLab

• Carlos Guestrin, et al.

• Graph algorithms used in machine learning

• View Computation as Localized Updates on Graph
  • New value depends on own value + those of neighbors
  • Update repeatedly until converge
Machine Learning Example

- PageRank Computation
  - Larry Page & Sergey Brinn, 1998
- Rank “Importance” of Web Pages
PageRank Computation

• Initially
  • Assign weight 1.0 to each page

• Iteratively
  • Select arbitrary node and update its value

• Convergence
  • Results unique, regardless of selection ordering

\[ R_1 \leftarrow 0.1 + 0.9 \times (\frac{1}{2} R_2 + \frac{1}{4} R_3 + \frac{1}{3} R_5) \]
PageRank with Map/Reduce

- Each Iteration: Update all nodes
  - Map: Generate values to pass along each edge
    - Key value 1: \((1, \frac{1}{2} R)\) \((1, \frac{1}{4} R)\) \((1, \frac{1}{3} R)\)
    - Similar for all other keys
  - Reduce: Combine edge values to get new rank
    - \(R_1 \leftarrow 0.1 + 0.9 \times \left( \frac{1}{2} R_2 + \frac{1}{4} R_3 + \frac{1}{3} R_5 \right)\)
    - Similar for all other nodes

- Performance
  - Very slow!
  - Altavista Webgraph 2002
    - 1.4B vertices, 6.7B edges

| Hadoop | 800 cores | 9000s |
PageRank with GraphLab

• Operation
  • Graph partitioned across multiple processors
    • Each doing updates to its portion of graph
    • Exploits locality
    • Greater asynchrony
    • Only iterate over portions of graph where values are changing

• Performance
  • Altavista Webgraph 2002
    • 1.4B vertices, 6.7B edges

<table>
<thead>
<tr>
<th></th>
<th>Hadoop</th>
<th>Prototype GraphLab2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>800 cores</td>
<td>512 cores</td>
</tr>
<tr>
<td></td>
<td>9000s</td>
<td>431s</td>
</tr>
</tbody>
</table>