Big Data processing with Hadoop

Luca Pireddu

CRS4—Distributed Computing Group

April 18, 2012
1 Motivation
   - Big Data
   - Parallelizing Big Data problems

2 MapReduce and Hadoop
   - MapReduce
   - Hadoop DFS
   - Cloud resources

3 Simplified Hadoop
   - Pydoop
   - Other high-level tools

4 Sample Hadoop use case: high throughput sequencing
   - HT sequencing at CRS4
   - Seal

5 Conclusion
Motivation
Data set sizes are growing. But why?
Data set sizes are growing. But why?

**Incentive:**
- Larger sizes tend to improve the sensitivity of analyses

**Ability:**
- More easily accessible sources of data
  - e.g., Internet, Twitter firehose
- Technology enables more ambitious science
  - e.g., LHC, whole-genome sequencing
- Cheaper and faster acquisition/tracking methods
  - e.g., cell phones, RFID tags, customer cards at the stores
Data sets can grow so big that it is difficult or impossible to handle them with conventional methods

- Too big to load into memory
- Too big to store on your desktop workstation
- Too long to compute with a single CPU
- Too long to read from a single disk

Problems that require the analysis of such data sets have taken the name of “Big Data” problems
Many big data problems are loosely-coupled and are easily parallelized. They may require high I/O throughput as large quantities of data is read/written. Do not require real-time communication between batch jobs. How should we parallelize them?
Poor man’s parallel processing

- manual data splitting into batches
- ad hoc scripting to automate, at least partially
- queueing system to distribute jobs to multiple machines
- shared storage to pass intermediate data sets
Poor man’s parallel processing

Presents many weaknesses

- High effort, low code re-use
- No robustness to equipment failure
- Failures typically require human intervention to recover
  - raises operator effort and therefore operating costs
- Usually less-than-desirable parallelism
  - Getting high-parallelism (especially more than per-file) can get complicated
- I/O done to/from shared storage
  - Limits scalability in number of nodes
  - Storage can become the bottleneck; alternatively, storage becomes very expensive
  - High network use as data is typically read and written remotely
  - Raises infrastructure costs
MapReduce and Hadoop
MapReduce

- A programming model for large-scale distributed data processing
- Aims to solve many of the issues just mentioned

- Breaks algorithms into two steps:
  1. *Map*: map a set of input key/value pairs to a set of intermediate key/value pairs
  2. *Reduce*: apply a function to all values associated to the same intermediate key; emit output key/value pairs

- Functions don’t have side effects; (k,v) pairs are the only input/output
- Functions don’t share data structures
Consider a program to calculate word frequency in a document.

The quick brown fox ate the lazy green fox.

<table>
<thead>
<tr>
<th>Word</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>ate</td>
<td>1</td>
</tr>
<tr>
<td>brown</td>
<td>1</td>
</tr>
<tr>
<td>fox</td>
<td>2</td>
</tr>
<tr>
<td>green</td>
<td>1</td>
</tr>
<tr>
<td>lazy</td>
<td>1</td>
</tr>
<tr>
<td>quick</td>
<td>1</td>
</tr>
<tr>
<td>the</td>
<td>2</td>
</tr>
</tbody>
</table>
A possible MapReduce algorithm:

**Map**
- Input: part of text
- For each word write a tuple \((\text{word}, 1)\)

**Reduce**
- Input: word \(w\), list of 1’s emitted for \(w\)
- Sum all 1’s into \(\text{count}\)
- Write tuple \((\text{word}, \text{count})\)
The quick brown fox ate the lazy green fox.

Here’s some pseudo code for a MapReduce word counting algorithm:

**Map**

```java
map(key, value):
    foreach word in value:
        emit(word, 1)
```

**Reduce**

```java
reduce(key, value_list):
    int wordcount = 0
    foreach count in value_list:
        wordcount += count
    emit(key, wordcount)
```
MapReduce Example – Word Count

The quick brown fox ate the lazy green fox

Mapper

- the, 1
- quick, 1
- brown, 1

Mapper

- fox, 1
- ate, 1
- lazy, 1

Mapper

- green, 1
- fox, 1

Map
MapReduce Example – Word Count

the quick brown fox ate the lazy green fox

Mapper -> Mapper -> Mapper

Shuffle & Sort

Reducer -> Reducer

Reduce
The lack of side effects and shared data structures is the key.

- No multi-threaded programming
- No synchronization, locks, mutexes, deadlocks, etc.
- No shared data implies no central bottleneck.
- Failed functions can be retried—their output only being committed upon successful completion.

MapReduce allows you to put much of the parallel programming into a reusable framework, outside of the application.
The MapReduce model needs an implementation
Hadoop is arguably the most popular open-source MapReduce implementation
Born out of Yahoo! Currently used by many very large operations
A MapReduce framework goes hand-in-hand with a distributed file system

- Multiplying the number of nodes poses challenges
  - multiplied network traffic
  - multiplied disk accesses
  - multiplied failure rates
A MapReduce framework goes hand-in-hand with a distributed file system

- Multiplying the number of nodes poses challenges
  - multiplied network traffic
  - multiplied disk accesses
  - multiplied failure rates

Hadoop provides the Hadoop Distributed File System (HDFS)

- Stores blocks of the data on each node.
  - Move computation to the data and decentralize data access
- Uses the disks on each node
  - Aggregate I/O throughput scales with the number of nodes
- Replicates data on multiple nodes
  - Resistance to node failure
Components

THE CAST

- **CLIENT**: People sit in front of me and ask me to read/write data
- **NAMENODE**: There is only ONE of me.. ..and I coordinate everything around here
- **DATANODES**: We store data.. ..there are MANY of us sometimes even thousands!

Image courtesy of Maneesh Varshney
Files split into large blocks (e.g., 64 MB)

Namenode maintains file system metadata
- Directory structure
- File names
- The ids of the blocks that compose the files
- The locations of those blocks
- The list of data nodes

Datanode stores, serves and deletes blocks
So, you have a big data problem
You’ve written the next great MapReduce application
You need a few hundred machines to run it...now what?
So, you have a big data problem
You’ve written the next great MapReduce application
You need a few hundred machines to run it... now what?

Rent them!
Lately there’s been a growth of Infrastructure as a Service (IaaS)
Rent infrastructure from companies that specialize in providing and maintaining them
e.g., Amazon Web Services (AWS), IBM
You can rent as many nodes as you need for as long as you need
even as little as one hour
pay as you go
elastic
Makes sense in many cases
peaky loads or temporary requirements—i.e., low average use
need to quickly grow capacity
don’t want to create an HPC group within the company
In April 2011 Amazon suffered a major service outage.
Simplified Hadoop
At CRS4 we’ve written a Python API for Hadoop called **Pydoop**

- Allows one to access most of Hadoop’s features with the simplicity of Python
- Lets you bridge Hadoop and C code
- Lets you script!!

---

**Pydoop script to turn text to lower case in Hadoop**

```python
def mapper(k, value, writer):
    writer.emit('', value.lower())
```
simple text-processing jobs reduced to two Python functions in a module
makes it easy to solve simple problems
makes it feasible to write simple (even throw-away) parallel programs

Pydoop wordcount script

def mapper(k, text, writer):
    for word in text.split():
        writer.emit(word, 1)

def reducer(word, count, writer):
    writer.emit(word, sum(map(int, count)))

Then run it with:
    pydoop_script wordcount.py hdfs_input hdfs_output
- If you’re interested, Pydoop is available on entu/oghe
- Use the python installation on els5
- At the next release we’ll ask our kind administrators to install it centrally
Other high-level tools

Other high-level Hadoop-based tools exist as well. E.g.,

- Pig
- Hive
- Cascading
- Cascalog
- Scalding
- Scrunch
- Spark
Sample Hadoop use case: high throughput sequencing
HT Sequencing

Genomics data growth

- Trend in sequencing technologies:
  - Lower costs
  - Increasing speed
  - Higher resolution
- Sequencing rate is growing exponentially
- Processing capacity is *not*

We are here

Processing capacity

Sequencing capacity

Time
Currently the largest sequencing center in Italy
Created to enable a number of studies on the Sardinian population

**Equipment:** 4 HiSeq2000 and 2 GAIIx Illuminas

**Capacity:** about 7000 Gbases/month

As its sequencing capacity grew, the operation faced scalability problems in its processing pipeline.

- Used “traditional” programs (some multi-threaded, not distributed)
- Data shared exclusively through Lustre volume
- Based on the “poor man’s parallelism”, with the consequential shortcomings
To solve those problems we began working on *Seal*.
To solve those problems we began working on Seal.

Seal is:

- a suite of distributed tools for processing HT sequencing data
- based on a proven technology: the Hadoop MapReduce framework
- used in production at the CRS4 Sequencing Center
- Released under GPLv3 license
- Web site: http://biodoop-seal.sf.net

Key goals

- **Scalable**
  - In cluster size
  - In data size
- **Robust**
  - Resilient to node failure and transient cluster problems
Currently featured tools

Seal currently has tools to perform distributed:

- read demultiplexing (Demux)
- read alignment (Seqal)
- sorting of read alignments (ReadSort)
- compute base quality statistics (RecabTable).

- These tools are implemented as MapReduce programs that run on Hadoop
Important features:

- distributed
- scalable
- robust
- open source

Part of these benefits are a direct consequence of Seal being based on Hadoop.

Others are thanks to implementation details:
- algorithm design
- shared (read-only) memory
- etc.
Evaluation

Important criteria

- throughput per node
- efficiency of the distribution mechanism
- scalability w.r.t. nodes and data size

Evaluation steps:

1. Establish a single-node baseline throughput measure
2. Compare throughput/node of baseline, old CSGP workflow and Seal equivalent
3. Compare wall-clock runtimes
4. Evaluate scalability characteristics
Baseline

Reflects what can be easily achieved on a workstation with no programming effort.

- Use multi-threaded programs where available
- 8-disk GPFS volume used for storage
## Data sets

### Baseline input data sets

<table>
<thead>
<tr>
<th>Dataset</th>
<th>No. tiles</th>
<th>No. pairs</th>
<th>Size (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dataset B3</td>
<td>10</td>
<td>3.6 \cdot 10^7</td>
<td>15.7</td>
</tr>
</tbody>
</table>

### Realistic data sets

<table>
<thead>
<tr>
<th>Dataset</th>
<th>No. lanes</th>
<th>No. pairs</th>
<th>Size (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dataset MR1</td>
<td>1</td>
<td>1.2 \cdot 10^8</td>
<td>51</td>
</tr>
<tr>
<td>Dataset MR3</td>
<td>3</td>
<td>3.3 \cdot 10^8</td>
<td>147</td>
</tr>
<tr>
<td>Dataset MR8</td>
<td>8</td>
<td>9.2 \cdot 10^8</td>
<td>406</td>
</tr>
</tbody>
</table>
Throughput per Node

- **Baseline**: B3 dataset
- **Old CSGP**: MR3 (16 nodes)
- **Seal**: MR3

- Nodes used efficiently (mainly because of improved parallelism)
- The overhead paid by Seal for distributing the work is minimal
### Scenario | No. nodes | Runtime (h)
--- | --- | ---
Old CSGP | 16 | 29.1
Seal | 16 | 5.4
Seal | 32 | 2.7
Seal | 64 | 1.4
Seal | 96 | 0.9

**Table:** Wall clock times, Dataset MR3.

- Significant speed-up over old workflow on same number of nodes (16)
- Evident linear scalability
Scalability

- Ideal system would produce a flat line
- 1-lane case starves at more than 16 nodes
Related publications:


- Various posters...

In addition, we’ve been invited to the SeqAhead Next Generation Sequencing Data Analysis Network.
MapReduce

- Is often a good solution for Big Data problems that present loosely coupled parallelism
- Especially true for I/O-bound problems
- Simplifies development of parallel programs
  - Especially true when using Pydoop
- Successfully used in Seal and many companies
- The robustness added of the system is essential for automation
- Automation is very important for scaling sample throughput and maximizing the R.O.I. in any large-scale operation.
Conclusion

MapReduce

- Is often a good solution for Big Data problems that present loosely coupled parallelism
- Especially true for I/O-bound problems
- Simplifies development of parallel programs
  - Especially true when using Pydoop
- Successfully used in Seal and many companies
- The robustness added of the system is essential for automation
- Automation is very important for scaling sample throughput and maximizing the R.O.I. in any large-scale operation.

Questions?

luca.pireddu@crs4.it (CRS4)

Big Data processing with Hadoop

April 18, 2012