GREAT 2011 Summer School

C2: How to work with a petabyte

Matthew J. Graham (Caltech, VAO)

Overview

- Strategy
- MapReduce
- Hadoop family
- GPUs

Divide-and-conquer strategy

- Most problems in astronomy are embarrassingly parallalizable
- Better technology just leads to scope scaling:
  - Better detectors ➔ increase number of pixels ➔ image coaddition
  - Better surveys ➔ increase number of objects in catalogs ➔ N-point correlation function
  - Better memory/processors ➔ increase number of simulation points ➔ cluster finding

MapReduce

- Primary choice for fault-tolerant and massively parallel data crunching
- Invented by Google fellows
- Based on functional programming map() and reduce() functions
- Reliable processing even though machines die
- En-large parallelization – thousands of machines for tera/petasort

What is MapReduce?

- Algorithm:
  - Input data is partitioned and processed independently by map tasks with each one emitting a list of <key, value> pairs as output
  - Pairs grouped by keys, yielding for each unique key k a list of values v_1, …, v_n of all values belonging to same key
  - “per-key” lists are processed independently by reduce tasks which collectively create final output
- Analogy to SQL:
  - Map is a group-by clause of an aggregate query
  - Reduce is an aggregate function computed over all rows with same group-by attribute

MapReduce canonical example

- Word count:
  - Map(key:uri, value:text)
    for word in tokenize(value):
    emit(word, 1)
  - Reduce(key:word type, value:list of 1s)
    emit(key, sum(value))

- Workthrough:
  - Map(key:"http://…", value:"Space: the final frontier…")
    -> ("Space", 1), ("the", 1), ("final", 1), ...
  - Group keys
    -> ("Space", (1)), ("the", (1, 1, 1)), …
  - Reduce(key, value)
    -> ("Space", 1), ("the", 3), ("new", 3), …

Use of MapReduce in astronomy

- Image Coaddition Pipeline (Wiley et al. 2011)
  - Evaluated image coaddition of 100000 SDSS images using Hadoop
  - Five possible methods of implementation with progressive improvements
  - Intend to develop full petascale data-reduction pipeline for LSST
- Berkeley Transient Classification Pipeline (Starr et al. 2010)
  - Make probabilistic statements about transients making use of their light curves the event occurs on the sky ("context") particularly with minimal data from survey of interest
  - Resampled ("noisified") well-sampled well-classified sources with precomputed candences, models for observing depths, sky brightness, etc. + generate classifiers for different PTF cadences
  - Uses Java classifiers from Weka direct with Hadoop; Python code with Hadoop Streaming; Cascading package; plan to use Mahout and Hive
- Large Survey Database (AAS 217 poster)
  - $>10^9$ rows, >1 TB data store for PS1 data analysis
  - In-house MapReduce system

Hadoop family (hadoop.apache.org)

- HDFS: distributed file system
- HBase: column-based db (webtable)
- Hive: Pseudo-RDB with SQL
- Pig: Scripting language
- Zookeeper: Coordination service
- Whirr: Running cloud services
- Cascading: Pipes and filters
- Sqoop: RDB interface
- Mahout: ML/DM library

Using Hadoop

- Java API
- Hadoop streaming supports other languages (anything that supports input from stdin, output to stdout):

```
> $HADOOP_HOME/bin/hadoop jar $HADOOP_HOME/hadoop-streaming.jar \
  -input myInputDirs \
  -output myOutputDir \
  -mapper myMap.py \
  -reducer myReduce.py
```

- Run locally, on remote cluster, in cloud
- Test first locally on small subset of data then deploy to expensive resources on full data set:

```
> cat data | map | sort | reduce
```

- Canonical example left as an exercise to the student

Pig (pig.apache.org)

- Pig Latin is a language which abstracts MapReduce programming (a la SQL for RDMBS)
- A Pig Latin program describes a series of operations (as statements) which are applied to the input data to produce output
- Process terabytes of data on a cluster with just a few lines of code in a terminal window
- Operators:
  - Loading/storing - LOAD, STORE, DUMP
  - Filtering - FILTER, DISTINCT, FOREACH…GENERATE, STREAM, SAMPLE
  - Grouping and joining - JOIN, COGROUP, GROUP, CROSS
  - Sorting - ORDER, LIMIT
  - Combining/splitting - UNION, SPLIT
  - Diagnostic – DESCRIBE, EXPLAIN, ILLUSTRATE
  - UDF – REGISTER, DEFINE

Pig canonical example

```
grunt>
A = LOAD '/mydata/mybook.txt';
B = FOREACH A GENERATE FLATTEN
   (TOKENIZE((chararray)$0)) AS word;
C = FILTER B BY word MATCHES '\\w+';
D = GROUP C by word;
E = FOREACH D GENERATE COUNT(C) AS
   count, GROUP AS word;
F = ORDER E BY count DESC;STORE F into
   ’/mydata/mybook.counts';
```

Hive (hive.apache.org)

- Hive organizes data into tables and provides HiveQL, a dialect of SQL but not full SQL-92, to run against them
- Queries are converted into a series of MapReduce jobs
- Maintains a metastore for service and table metadata
- Differences from traditional RDBMS:
  - Verifies data when a query is issued (schema on read)
  - Full table scans are the norm so updates, transactions and updates are currently unsupported
  - High latency (minutes not milliseconds)
  - Supports complex data types: ARRAY, MAP, and STRUCT
  - Tables can be partitioned and bucketed in multiple dimensions
  - Specific storage formats
  - Multitable inserts
  - UDFs/UDTFs/UDAFs in Java

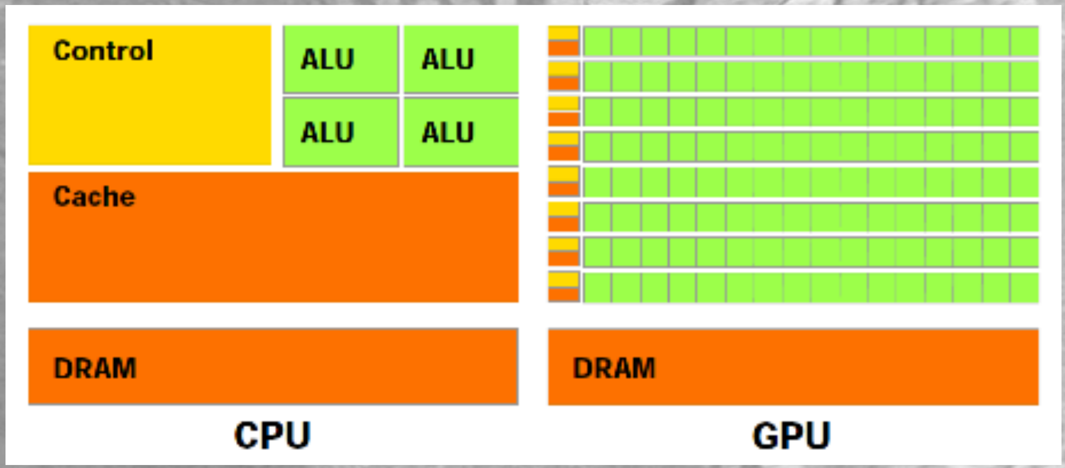
Hive canonical example

```
CREATE TABLE docs(contents STRING)
    ROW FORMAT DELIMITED
    LOCATION '/mydata/mybook.txt';

FROM (
    MAP docs.contents
        USING 'tokenizer_script' AS word, cnt
        FROM docs
        CLUSTER BY word) map_output

REDUCE map_output.word, map_output.cnt
    USING 'count_script' AS word, cnt;
```

Alternates to MapReduce (NoHadoop)

- Percolator
  - Incrementally update massive data set continuouosly
- Apache Hama
  - Implementation of BSP (Bulk Synchronous Parallel)
  - Alternate to MPI, smaller API, impossibility of deadlocks, evaluate computational cost of an algorithm as function of machine parameters
- Pregel:
  - Very large graphs (billions of nodes, trillions of edges)
  - Uses BSP
  - Computations are applied at each node until
  - Cross-matched catalogs (GAIA, LSST, SKA)

GPUs



- 1536 cores per multiprocessor (high-end)
- Each core can run 16 threads (~25k threads/GPU)
- Threads are lightweight so can easily launch ~billion threads/sec

Programming GPUs

- Favours brute force approach rather than ported smart algorithms
- CUDA (NVIDIA) and OpenCL libraries for C
- Various libraries available: sorting, BLAS, FFT, …
- Thrust for C++
- PyCUDA/PyOpenCL for Python
- Mathematica/MATLAB

PyCUDA example

```
import numpy as np
from pycuda import driver, compiler, gpuarray, tools
from pycuda.curandom import rand as curand
import pycuda.autoinit

kernel_code = """
__global__ void multiply (float *dest, float *a, float *b)
{
  const int i = threadIdx.x;
  dest[i] = a[i] * b[i];
}
"""

mod = compiler.SourceModule(kernel_code)
multiply = mod.get_function("multiply")
a = np.random.randn(400).astype(np.float32)
b = np.random.randn(400).astype(np.float32)
ans = np.zeros_like(a)
multiply(
        driver.Out(ans), driver.In(a), driver.In(b),
        block=(400,1,1))
print dest-a*b
```