Map Reduce

Last Time …

- MPI I/O
- Randomized Algorithms
  - Parallel k-Select
  - Graph coloring
- Assignment 2

- **Questions ?**

Today …

- Map Reduce
  - Overview
  - Matrix multiplication
  - Complexity theory

MapReduce programming interface

- Two-stage data processing
  - Data can be divided into many chunks
  - A map task processes input data & generates local results for one or a few chunks
  - A reduce task aggregates & merges local results from multiple map tasks
- Data is always represented as a set of key-value pairs
  - Key helps grouping for the reduce tasks
  - Though key is not always needed (for some applications, or for the input data), a consistent data represent eases the programming interface

Motivation & design principles

- Fault tolerance
  - Loss of a single node or an entire rack
  - Redundant file storage
- Files can be enormous
- Files are rarely updated
  - Read data, perform calculations
  - Append rather than modify
- Dominated by communication costs and I/O
  - Computation is cheap compared with data access
- Dominated by input size

Example

- Count the occurrences of individual words in bunch of web pages
- Map task: find words in one or a few files
  - Input: <key = page url, value = page content>
  - Output: <key = word, value = word count>
- Reduce task: compute total word counts across multiple files
  - Input/output: <key = word, value = word count>

Dependency in MapReduce

- Map tasks are independent from each other, can all run in parallel
- A map task must finish before the reduce task that processes its result
- In many cases, reduce tasks are commutative

- Acyclic graph model

Implementations

- Original mapreduce implementation at Google
  - Not publicly available
  - C/C++
- Hadoop
  - Open Source Implementation – Yahoo!, Apache
  - Java
- Phoenix++
  - Open Source – Stanford
  - C++ - Shared memory

Applications that don't fit

- MapReduce supports limited semantics
  - The key success of MapReduce depends on the assumption that the dominant part of data processing can be divided into a large number of independent tasks
- What applications don't fit this?
  - Those with complex dependencies – Gaussian elimination, k-means clustering, iterative methods, n-body problems, graph problems, …

MapReduce

- Map: chunks from DFS → (key, value)
  - User code to determine (k, v) from chunks (files/data)
- Sort: (k, v) from each map task are collected by a master controller and sorted by key and divided among the reduce tasks
- Reduce: work on one key at a time and combine all the values associated with that key
  - Manner of combination is determined by user code

MapReduce – word counting

- Input → set of documents
- Map:
  - reads a document and breaks it into a sequence of words
    $$w_1, w_2, \dots, w_n$$
  - Generates (k, v) pairs,
    $$(w_1, 1), (w_2, 1), \dots, (w_n, 1)$$
- System:
  - group all (k, v) by key
  - Given r reduce tasks, assign keys to reduce tasks using a hash function
- Reduce:
  - Combine the values associated with a given key
  - Add up all the values associated with the word → total count for that word

Parallelism

- Reducers
- Reduce Tasks
- Compute nodes
- Skew (load imbalance)

- Easy to implement
- Hard to get performance

Node failures

- Master node fails
  - Restart mapreduce job
- Node with Map worker fails
  - Redo all map tasks assigned to this worker
    - Set this worker as idle
    - Inform reduce tasks about change of input location
- Node with Reduce worker fails
  - Set the worker as idle

Matrix-vector multiplication

- $n \times n$ matrix M with entries m_{ij}
- Vector $\boldsymbol{v}$ of length n with values v_j
- We wish to compute

$$x_i = \sum_{j=1}^{n} m_{ij} v_j$$

- If $\boldsymbol{v}$ can fit in memory
  - Map: generate $(i, m_{ij} v_j)$
  - Reduce: sum all values of i to produce (i, x_i)
- If $\boldsymbol{v}$ is too large to fit in memory? Stripes? Blocks?
- What if we need to do this iteratively?

Matrix-Matrix Multiplication

- $P = MN \rightarrow p_{ik} = \sum_j m_{ij} n_{jk}$
- 2 mapreduce operations
  - Map 1: produce (k, v), $\left(j, (M, i, m_{ij})\right)$ and $\left(j, (N, k, n_{jk})\right)$
  - Reduce 1: for each $j \rightarrow \left(i, k, m_{ij}, n_{jk}\right)$
  - Map 2: identity
  - Reduce 2: sum all values associated with key (i, k)

Matrix-Matrix multiplication

- In one mapreduce step
- Map:
  - generate $(k, v) \rightarrow \big((i,k),(M,j,m_{ij})\big)$ & $\big((i,k),(N,j,n_{jk})\big)$
- Reduce:
  - each key (i,k) will have values $\big((i,k),(M,j,m_{ij})\big)$ & $\big((i,k),(N,j,n_{jk})\big)$ $\forall j$
  - Sort all values by j
  - Extract m_{ij} & n_{jk} and multiply, accumulate the sum

Complexity Theory for mapreduce

Communication cost

- Communication cost of a task is the size of the input to the task
- We do not consider the amount of time it takes each task to execute when estimating the running time of an algorithm
- The algorithm output is rarely large compared with the input or the intermediate data produced by the algorithm

Reducer size & Replication rate

- Reducer size (q)
  - Upper bound on the number of values that are allowed to appear in the list associated with a single key
    - By making the reducer size small, we can force there to be many reducers
      - High parallelism → low wall-clock time
    - By choosing a small q we can perform the computation associated with a single reducer entirely in the main memory of the compute node
      - Low synchronization (Comm/IO) → low wall clock time
- Replication rate (r)
  - number of (k, v) pairs produced by all the Map tasks on all the inputs, divided by the number of inputs
  - r is the average communication from Map tasks to Reduce tasks

Example: one-pass matrix mult

- Assume matrices are $n \times n$
- r – replication rate
  - Each element m_{ij} produces n keys
  - Similarly each n_{jk} produces n keys
  - Each input produces exactly n keys → load balance
- q – reducer size
  - Each key has n values from M and n values from N
  - $2n$

Example: two-pass matrix mult

- Assume matrices are $n \times n$
- r – replication rate
  - Each element m_{ij} produces 1 key
  - Similarly each n_{jk} produces 1 key
  - Each input produces exactly 1 key (2nd pass)
- q – reducer size
  - Each key has n values from M and n values from N
  - $2n$ (1st pass), n (2nd pass)

Real world example: Similarity Joins

- Given a large set of elements X and a similarity measure $s(x, y)$
- Output: pairs whose similarity exceeds a given threshold t
- Example: given a database of 10^6 images of size 1MB each, find pairs of images that are similar
- Input: (i, P_i), where i is an ID for the picture and P_i is the image
- Output: (P_i, P_j) or simply (i, j) for those pairs where $s(P_i, P_j) > t$

Approach 1

- Map: generate (k, v)

$$\big((i,j),(P_i,P_j)\big)$$

- Reduce:
  - Apply similarity function to each value (image pair)
  - Output pair if similarity above threshold t

- Reducer size – q → 2 (2MB)
- Replication rate – r → $10^6 - 1$
- Total communication from map→reduce tasks?
  - $10^6 \times 10^6 \times 10^6$ bytes → 10^{18} bytes → 1 Exabyte (kB MB GB TB PB EB)
  - Communicate over GigE → 10^{10} sec → 300 years

Approach 2: group images

- Group images into g groups with $\frac{10^6}{g}$ images each
- Map: Take input element (i, P_i) and generate
  - $(g-1)$ keys $(u, v) \mid P_i \in \mathcal{G}(u),\ \ v \in \{1, \dots, g\} \setminus \{u\}$
  - Associated value is (i, P_i)
- Reduce: consider key (u, v)
  - Associated list will have $2 \times \frac{10^6}{g}$ elements (j, P_j)
  - Take each (i, P_i) and (j, P_j) where i, j belong to different groups and compute $s(P_i, P_j)$
  - Compare pictures belonging to the same group
    - heuristic for who does this, say reducer for key $(u, u+1)$

Approach 2: group images

- Replication rate: $r = g - 1$
- Reducer size: $q = 2 \times 10^6/g$
- Input size: $2 \times 10^{12}/g$ bytes
- Say $g = 1000$,
  - Input is 2GB
  - Total communication: $10^6 \times 999 \times 10^6 = 10^{15}$ bytes → 1 petabyte