

# Introduction to DISC and Hadoop

Alice E. Fischer

April 24, 2009

- 1 Data Intensive Scalable Computing
  - Goals and Applications
  - Parallel Computing – Past and Present
  - Applications of Hadoop
  - Hadoop cannot speed up every application.
  
- 2 Hadoop and Google's Map-Reduce
  - History
  - Hadoop provides a three-layer paradigm

# Data Intensive Scalable Computing

- Uses parallel computation to process masses of data quickly.
- Scales easily to larger or smaller data sets.
- Is robust and can survive failure of one or more nodes during a computation.

# Hadoop

- Is a framework for implementing DISC.
- Program segments may be written in Java or Python.
- Runs on cluster computers. (One master node, several or many slaves.)
- Is based on a distributed and replicated file system.
- Automatically manages the hardware facility, which frequently expands or contracts.
- Allows use of cloud computing, that is, hardware that non-local and not owned.

## Programmer-directed Parallelism

A language can provide primitives for creating and managing parallelism:

- Fine-grained, programmer directed:  
    `parbegin` and `parend` (ALGOL-68)
- Functional languages with lazy evaluation avoid specification of order.
- Course-grained parallelism: `fork` and `join` (UNIX)  
    Java threads and synchronization primitives.

## Hardware and Compiler-based Parallelism

Computer systems often provide parallel hardware:

- Floating-point co-processors: 1980's, 8086 chip.
- Pipelined architecture: 1980's
- Dual-core processors: Pentium Extreme, 2005
- Quad-core: AMD Phenom, 2008

The extent to which these can be exploited depends on the cleverness of compiler designers.

## Parallelism Based on Memory Architecture

- A Hypercube has a large number of memory nodes cross-connected so that a node is “adjacent” to others in 2, 3, or 4 dimensions. Communication between two adjacent nodes is fast; between distant nodes is slower.
- One process controls the computation and flow of information among the many nodes.

## Parallelism Based on Cluster Computing

The Linda system, based on David Gelernter's *tuple spaces* has been implemented on multiple computers sharing a local network.

A tuple space is an implementation of associative memory for parallel/distributed computing.

It provides a repository of tuples that can be accessed concurrently.

For example, consider a group of processors that produce data and post it as tuples in the space, and a group of processors that retrieve and use data from the space that match a certain pattern.

## Is Hadoop Different?

Yes.

- The distributed / replicated file system goes beyond Linda.
- Unlike Java threads, and UNIX processes, the programmer does not need to worry about synchronization.
- Configuration is easy and management of resources automatic.
- It runs on ordinary computer hardware.
- It is open source and therefore free.

## What kind of applications are appropriate?

Hadoop is useful for some kinds of problems:

- Data records must be largely independent of each other—the order in which they are processed must not matter.
- Data must require too much space and/or too much processing time to be handled by one computer.
- At some final stage, results from the parallel computations need to be related to each other.

## Possible Hadoop applications

- Google's page-rank algorithm: masses of web-links are reduced daily to ordered page ranks.
- Large-scale searches; data mining. (The results of parallel searches can be related to each other.)
- Typesetting a massive LaTeX document. (Most typesetting actions are local, the final pagination relates them.)
- IRS: matching employer copies of W2 forms to employee returns. (Data can be reordered, then collated.)
- Cinematography: Process the frames of a movie to create a ghost by making a character transparent in every screen. The frames would be processed in parallel and assembled in order at the last stage.

## Sequential algorithms.

- Problems that are inherently sequential benefit little from parallel computation. Example: simulating a sequential process.
- Some problems involve massive unstructured data, but do not need a final step that integrates or correlates the data. Example: Processing the IRS refund checks. These can simply be done by a battery of separate machines.

## Hadoop and Map-reduce do not fit the bill.

Hadoop provides the wrong kind of parallelism here:

- Edge detection, object detection, and smoothing operations on a digital image.
- Solving systems of partial differential equations by simulation. Earthquake calculations involve processing 8-dimensional earth models. There is a huge amount of data, but each cell in the model affects computations at the next time step on all adjacent cells. The data is too structured for Hadoop. (Use a hypercube.)
- Simulating the growth of a population in a non-uniform and limited environment. Hadoop does not fit the iterative nature of the problem, the increase in nodes at each generation or the non-uniform processing of births, deaths, and migrations.

## Commercial Involvement

- Google released an early and partial description of map-reduce.
- Google wants the open source community to develop a competing product.
- Google wants students to learn about map-reduce and how to program in Hadoop.
- IBM and Yahoo are supporting the development and teaching of Hadoop.

# The File System

The Google file system is fundamental to the map-reduce paradigm. Hadoop is based on an implementation of the same idea. A file is:

- Distributed: Segments of a file are kept on many different machines.
- Replicated: Each segment is stored three times on nodes that are unlikely to fail simultaneously.
- Managed: If a node fails, its segments are lost, but two copies remain of each. A third copy of each segment is promptly made on another node.

## Phase 1: Mapping

For each segment of data in the input file, one of the machines that stores it is assigned to run the mapping code.

The output of a mapper is a set of  $\langle \text{key}, \text{record} \rangle$  pairs.

Users can optionally specify a Combiner to perform local aggregation of the intermediate outputs on the mapper nodes.

This helps to cut down the amount of data transferred from the Mapper to the Reducers.

## Phase 2: Collection and Redistribution

This step involves Shuffle and Sort actions that occur simultaneously and probably run on the same nodes as Phase 3. This stage processes the output of the Mappers and produces input for the reduce() step.

The key-groups, are partitioned among the Reducers. Users can control which keys (and hence which records) go to which Reducer by implementing a custom Partitioner.

**Shuffle:** For each Reducer, the framework fetches the relevant portion of the output of all the mappers, via HTTP.

**Sort:** The framework groups Reducer inputs by keys (different mappers typically output the same keys). The overall action is like a mergesort.

## Phase 3: Reducing

The reducing code can be written in Java, Python, etc.

A key group becomes the input to the `reduce()` function running on each Reducer. Each Reducer handles one key, or several, if there are more keys than reducers.

It reduces the set of intermediate values which share a key to a smaller set of values.

The smaller set, often with derived information added and data fields rearranged in a useful manner, becomes the output.

This output might become the input to another map-reduce step.