Chapter 23

Real-life environments for big-data computations (MapReduce etc.)

First 2/3rd based upon the guest lecture of Kai Li, and the other 1/3rd upon Sanjeev's lecture

23.1 Parallel Processing

These days many algorithms need to be run on huge inputs; gigabytes still can fit in RAM on a single computer, but terabytes or more invariably require a multiprocessor architecture. This state of affairs seems here to stay since (a) Moore's law has slowed a lot, and there is no solution in sight. So processing speeds and RAM sizes are no longer growing as fast as in the past. (b) Data sets are growing very fast.

Multiprocessors (aka parallel computers) involve multiple CPUs operating on some form of distributed memory. This often means that processors may compete in writing to the same memory location, and the system design has to take this into account. Parallel systems have been around for many decades, and continue to evolve with changing needs and technology.

There are three major types of systems based upon their design:

Shared memory multiprocessor. Multiple processors operate on the same memory; there are explicit mechanisms for handling conflicting updates. The underlying memory architecture may be complicated but the abstraction presented to the programmer is that of a single memory. The programming abstraction often involves *threads* that use *synchronization* primitives to handle conflicting updates to a memory location. *Pros:* The programming is relatively easy; data structures are familiar. *Cons:* Hard to scale to very large sizes. In particular, cannot handle hardware failure (all hell can break lose otherwise).

Message passing models: Different processors control their own memory; data movement is via message passing (this provides implicit synchronization). *Pros:* Such systems are easier to scale. Can use checkpoint/recovery to deal with node failures. *Cons:* No clean data structures.

Commodity clusters: Large number of off-the-shelf computers (with their own memories) linked together with a LAN. There is no shared memory or storage. *Pros:* Easy to scale; can easily handle the petabyte-size or larger data sets. *Cons:* Programming model has to deal explicitly with failures.

Tech companies and data centers have gravitated towards commodity clusters with tens of thousands or more processors. The power consumption may approach that of a small town. In such massive systems failures —processor, power supplies, hard drives etc.—are inevitable. The software must be designed to provide reliability on top of such frequent failures. Some techniques: (a) replicate data on multiple disks/machines (b) replicate computation by splitting into smaller subtasks (c) use good data placement to avoid long latency.

Google pioneered many such systems for their data centers and released some of these for general use. MapReduce is a notable example. The open source community then came up with its own versions of such systems, such as Hadoop. SPARK is another programming environment developed for ML applications.

23.2 MapReduce

MapReduce is Google's programming interface for commodity computing. It is evolved from older ideas in functional programming and databases. It is easy to pick up, but achieving high performance requires mastery of the system.

It abstracts away issues of data replication, processor failure/retry etc. from the programmer. One consequence is that there is no guarantee on running time.

The programming abstraction is rather simple: the data resides in an unsorted clump of (*key, value*) pairs. We call this a *database* to ease exposition. (The programmer has to write a MAPPER function that produces this database from the data.) Starting with such a database, the system applies a SORT that moves all pairs with the same *key* to the same physical location. Then it applies a REDUCE operation –provided by the programmer—that takes a bunch of pairs with the same *key* and applies some combiner function to produce a new single pair with that key and whose *value* is some specified combination of the old values.

EXAMPLE 50 (Word Count) The analog to the usual *Hello World* program in the MapReduce world is the program to count the number of repetitions of each word. The programmer provides the following.

MAPPER Input: a text corpus. Output: for each word w, produce the pair $(w, 1)$. This gives a database.

REDUCE: Given a bunch of pairs of type $(w, count)$ produces a pair of type (w, C) where C is the sum of all the counts.

EXAMPLE 51 (Matrix Vector Multiplication)

MAPPER: Input is an $n \times n$ matrix M, and a $n \times 1$ vector V. Output: Pairs $(i, m_{ij} \cdot v_j)$ for all $\{i, j\}$ for which $m_{ij} \neq 0$.

REDUCER: Again, just adds up all pairs with the same *key* and sum up their values.

One can similarly do other linear algebra operations.

Some other examples of mapreduce programs appear in Jelani Nelson's notes
`http://people.seas.harvard.edu/ minilek/cs229r/lec/lec24.pdf`

The MapReduce paradigm was introduced in the following paper:

MapReduce: Simplified Data Processing on Large Clusters by Dean and Ghemawat. OSDI 2004.

While it has been very influential, it is not suited for all applications. A critical appraisal appears in a blog post *MapReduce: a major step backwards,* by DeWitt and Stonebraker, which argues that MapReduce ignores many important lessons learnt in decades of Database Design. However, in the years since, the MapReduce paradigm has been extended to incorporate some of those lessons.