Programming Models for processing large data sets in distributed systems

Aris Paphitis
CUT

Key Point

A brief survey of programming models used for mining and generating large datasets in modern distributed systems. Data sets are growing continuously. This imposes the devise of more flexible approaches in data retrieval and processing to produce meaningful information. The ideas described in this text were developed in different time periods, trying each time to satisfy different objectives.

1 Motivation

In the modern world the accumulation of data has become an intrinsic characteristic of a wide range of scientific, social, and economic applications. In order to be able to extract useful information from all the data one gathers, in a reasonable ammount of time, the computations have to be distributed accross hundreds, or even thousands, of machines. The problem rises when programmers try to parallelize the computation in a coordinated manner, taking care of data distribution and fault tolerance. All these programming models have as a goal to simplify the interface of the parallelization task assigned to the programmer and hide the complex code needed to deal with these issues. They manage to achieve this by introducing abstractions that allow the programmer to express simple computations but handle automatically parallelization, distribution, fault tolerance and load balancing.

2 MapReduce

Created and developed in *Google Inc*, MapReduce was inspired by the `map` and `reduce` primitives present in many functional languages.

MapReduce provides automatic parallelization and distribution, fault-tolerance, I/O scheduling, status and monitoring. The computation takes a set of input key/value pairs and produces a set of output key/value pairs. The computation is expressed as two seperate functions, `Map` and `Reduce`; hence the name. The `Map` takes the input and produces an intermediate output of pairs. This in turn is *shuffled*; the MapReduce library groups together all intermediate values associated with the same intermediate key and passes them to the `Reduce` function. Next the `Reduce` function, accepts the intermediate key with its set of values and merges them to produce a smaller set of values. Fault-tolerance is handled via re-execution. Refinements on completion time using redundant execusions near the end of a phase. Master scheduling policy which optimizes locality (same machine/same rack) so thousand machines read input at local disk speeds (bypassing network latency).

When performing the reduce part you have to repartition and redistribute your data. Limited programming model: Map, Reduce, Sort.

3 DryadLINQ

Dryad is a general purpose distributed computing engine for coarse-grain data-parallel applications, on clusters of Windows-based machines. Dryad requires user created parallel tasks. That responsibility was automated through the introduction of DryadLINQ. DryadLINQ allows developers to express Dryad applications in managed code by using the LINQ (Language-Integrated Query) programming model and API. A DryadLINQ program is a sequential program composed of LINQ expressions performing transformations on datasets. The DryadLINQ system automatically and transparently translates the data-parallel portions of the program into a distributed execution plan which is passed to the Dryad execution platform. DruadLINQ is designed to provide flexible and efficient distributed computation in any LINQ-enabled language inluding C#, VB and F#. The Dryad system does not raise restrictions in an application's communication flow, as MapReduce and parallel

databases do. Instead, it allows the developer to fine control the communication graph as well as the subroutings that live at its vertices. The developer can specify an arbitrary DAG (Directed Acyclic Graph) to describe the application's communication patterns and express the data transport mechanisms between computations.

Dryad: framework for distributed computing/parallel programming. Automatic fault tolerance, load balancing, bring data to computations.

The idea behind Dryad: Many porgrams can be represented as a distributed execution graph. Dryad acts as a middleware abstraction that runs this execution graph. Is independent of the programming model above. It's implementation is more general than that of MapReduce.

Runtime figures out how to virtualize the job -¿ creates a static job graph and then assigns the job, moves the data to the underlying computation. Dryad binds the virtual graph to runtime.

Fault - tolerance JobManager monitors the health of each individual job.

3.1 Runtime

Every Job is represented as a Direct Acyclic Graph. The edges of this graph (channels) denote data movement and vertices denote programs processing the data. Channels can be tcp pipes, disk files or shared memory.

3.2 Model

4 Piccolo

Unlike previous data-flow models, Piccolo allows computations running on different machines to share distributed, mutable state via a key-value table interface. Data-flow models do not expose global state, so existing approaches (MapReduce, Dryad) have to write to the distributed storage at the end of each iteration. Piccolo is described as a data-centric model, differentiating from other models using message passing primitives. The communication-centric model provided by message passing models is a very low-level abstraction which makes data processing a burden - it mainly focuses on data location and data transfer. The programmer explicitly defines partitioning and internode communications.

Piccolo's goal is to provide performance (as in message passing) and ease of use by allowing inexplicit communication. Piccolo exposes a global table interface which is available to all parts of the computation simultaneously, allowing users to specify programs in an intuitive manner similar to that if writing programs for a single machine. Synchronization is achieved by allowing users to defer synchronization logic to the system. Instead of explicitly locking tables in order to perform updates, users can attach accumulation functions to a table; these are used automatically by the framework to correctly combine concurrent updates to a table entry.

5 RDDs - Spark

People soon started to demand more and more from the underlying model in terms of usability, complexity, and execution time. Things like multi-pass applications (machine learning and graph algorithms), interactive adhoc queries, and more real-time stream processing. The Spark programming model tries to disengage the replication of data from the fault-tolerance property of distributed file systems. Instead of allowing one to arbitrary make updates to shared state, which is hard to maintain consistent, a restricted form of distributed shared memory is proposed. Additionally an efficient fault recovery system using `lineage` is employed. These novel memory abstractions are called Resilient Distributed Datasets (RDDs). They are immutable and can only be built through coarse-grained deterministic operations. Fault tolerance is achieved using lineage; operations used to build the data are stored. This minimizes the overhead since only the operation is logged not the actual data. In the event of failures only the lost partitions are recomputed and there is no cost if nothing fails.

It is demonstrated that despite their restrictions, RDDs are quite general and can express suprisingly many parallel algorithms; that is to apply the same operation to many items. To demonstrate their use, they have implemented RDDs in a sytem called Spark; A programming interface that allows the creation, control and operation of such datasets.

RDDs can be created from stable storage or other RDDs through *transformations*. They need not be materialized at all times. Users can control their persistency or partitioning allowing for optimizations. RDDs are used through *actions*, operations that return a value. Spark computes RDDs lazily the first time they are used in an action by pipelining transformations (thus eliminating the need for creating, using, and removing temporary files).

RDDs are compared against distributed shared memory (DSM) and is shown that they perform better on applications which perform bulk writes. RDDs offer more efficient fault tolerance and can additionally exploit data locality, by means of partitioning and scheduling, to improve performance.

To use Spark, developers write a driver program that connects to a cluster of workers. Spark code on the driver tracks RDDs' lineage.

6 Pregel

Pregel is a framework, developed at Google, to query distributed, directed graphs.It became known as the MapReduce for graphs; it has the same Map and Reduce steps but it also has several iterations of those steps. It aims at operating all connected machines at full capacity at all times and at reducing the overall network traffic. It is good on performing calculations touching all vertices but performs poor when calculations involve only a small fraction of the graph's vertices.

6.1 Motivation

Classical graph algorithms include Pattern Match (identify similar components - clustering), Traversals (iteratively explore the graph), and Global Measurements (count all vertices). These operations require to touch all vertices and their neighborhoods, keep a history of visited edges, and/or a global view on the graph. Because of these requirements, theses classical algorithms are hard to implement on distributed a graph, a graph so large that does not fit in one computer. Thus, Pregel was developed to alleviate this limitations.

6.2 Architecture

(more on Pregel . . .)

7 History - MPI

Message Passing Interface [1] In the early 1990s, high-performance computing was in the process of converting from the vector machines that had dominated scientific computing in the 1980s to massively parallel processors (MPPs). In addition, people were beginning to use networks of desktop workstations as parallel computers. Both the MPPs and the workstation networks shared the message-passing model of parallel computation, but programs were not portable. There was no unified, common syntax that would enable a program to run in all the parallel environments that were suitable for it from the hardware point of view. The MPI Forum developed a standard for the strict message-passing model, in which all data transfer is a cooperative operation among participating processes. It was assumed that the number of processes was fixed and that processes were started by some (unspecified) mechanism external to MPI. I/O was ignored, and language bindings were limited to C and Fortran 77.

MPI, the Message-Passing Interface, is an application programmer interface (API) for programming parallel computers. It was first released in 1992 and transformed scientific parallel computing. Today, MPI is widely using on everything from laptops (where it makes it easy to develop and debug) to the world's largest and fastest computers. Among the reasons for the the success of MPI is its focus on performance, scalability, and support for building tools and libraries that extend the power of MPI.

Two different approaches:

Explicit multithreading, which is the use of an API that manipulates threads within a single address space. This approach may be sufficient on systems that can devote a large number of CPUs to servicing a single process, but interprocess communication will still need to be used on scalable systems. The MPI API has been designed to be thread safe. However, not all implementations are thread safe. An MPI-2 feature is to allow applications to request and MPI implementations to report their level of thread safety. In some cases the compiler generates the thread parallelism. In such cases the application or library uses only the MPI API, and additional parallelism is uncovered by the compiler and expressed in the code it generates. Some compilers do this unaided; others respond to directives in the form of specific comments in the code. *OpenMP* is a proposed standard for compiler directives for expressing parallelism, with particular emphasis on loop-level parallelism. Both C and Fortran versions exist.

8 Summary

8.1 Data Partition and Locality

Reducing disk and network access. -¿ Use of locality OR use in-memory computations. Locality via partitioning.

8.2 Scheduling

Scheduler

8.3 Fault tolerance

Fault tolerance = Replication or Logging. RDDs -¿ lineage.

9 Discussion

Describe where each paradigm is strong (and why) and where it is weak Future design objectives/thoughts... MapReduce on Hadoop2. Piccolo's development has stopped. Microsoft discontinued active deployment on Dryad, shifting focus to other frameworks (Apache Hadoop)-¿ wikipedia for reference. Spark.

Notes

[1] from https://bibliotecas.cio.mx/ebooks/e0206.pdf