CME 323: Distributed Algorithms and Optimization, Spring 2020
`http://stanford.edu/~rezab/dao`.
Instructor: Reza Zadeh, Matroid and Stanford.

Lecture 11, 5/5/2020. Scribed by Andreas Santucci, Edited by Robin Brown.

11 Introduction to Distributed Computing

In the first half of this course, we introduced parallel computation. Now, we will shift our focus to distributed computation. Generally, parallel computing refers to systems where multiple processors are located in close vicinity of each other (often in the same machine), and thus work in tight synchrony. Distributed computation can refer to a wide variety of system, ranging from commodity cluster computing to computing in a data network; these systems all share the additional challenge of inter-processor communication and coordination.

Typically, parallel computing systems are built for the sole purpose of exploiting processor cooperation to solve large, complex problems. On the other hand, many distributed computation systems arise from scenarios where processors may carry out other private tasks in addition to the computational tasks of the network. For example, a data network exists to service some data communication needs, while the distributed computation in the network is only a side activity supporting the main activity. Another example is a wireless sensor network that is deployed to monitor a vast geographic area where observations are used downstream to estimate a parameter describing the environment. When the measurement data is large enough, it may make more sense to perform a distributed estimation of the parameter (i.e., solve for the maximum likelihood estimate) at the sensors rather than aggregating all data at a central fusion center. In these examples, the system architecture is dictated by the primary activities of the system and so it cannot be designed to facilitate easy computation. Later we still specifically address computing in a commodity cluster; while commodity clusters are built expressly to perform large, complex computations efficiently, they share many of the same challenges of coordination and communication with other distributed computing systems.

11.1 Issues unique to Distributed Computing

Communication One of the key challenges in distributed computing is that communication does not come for free. Moreover, there are limits to the quantities of information we can send and the speed at which we can send it. In many scenarios, the cost of transmitting information dominates other costs of the computation, such as processing time and storage capacity. While in the PRAM model, processors do not explicitly pass messages to each other, the shared memory can be read from and written to by all processors (with the exact mechanisms depending on the variant of the model) and thus can be used for communication. As a result, communication is not typically the bottleneck of the computation, and thus is not the focus of algorithm design.

Just how large is the difference between RAM and hardisk?

- Random access lookup
  - Main memory on a piece of RAM can take 100 nanoseconds to pick up a piece of data
  - A hard disk[1] can take 10 million nanoseconds to fetch the same data, i.e. 100k× slower.
- Sequential access lookup
  - Reading one megabyte from RAM: 1/4 million nanoseconds
  - Reading one megabyte from disk: 30 million nanoseconds, i.e. 100x slower

The more data we are reading, the difference between sequential RAM and disk reads shrinks, but the order of magnitude difference is still there. We need to start dealing with this now, even for our algorithms on a single machine. This dramatic difference gives rise to a model of computation called streaming or online algorithms. The basic idea is that data comes at you in a stream, and you are allowed to maintain some amount of state (much smaller than the amount of data you consume) in RAM, and at termination, state outputs results.

Incomplete Knowledge Another key challenge in distributed computing is that each processor only has a limited knowledge of the systems and the computations carried out by other processors. To illustrate extent of this challenge, consider uniform random sampling from the set of inputs to a problem (for example, if you wanted to run stochastic gradient descent). In the parallel or centralized settings, the entire input to the problem is store in memory, so sampling from this data is as simple as uniformly generating an integer from 1 through n (where n is the size of the input) and selecting the data entry associated with that integer. In distributed computing, the data is often sharded between machines. This means that each processor may not know the size of the input shared at other processors and the size of the entire input is unknown. Consequently, the simple centralized approach cannot be used in the distributed case. (we will discuss simple random sampling in detail later in the course).

Depending on the application, each processor may not even have full knowledge of the other processors in the network. For example, a fleet of mobile robots might need to jointly optimize their actions while each robot does not have knowledge of the presence of other robots outside of its sensing range. We will not focus on this setting in this course—this example is primarily to illustrate the knowledge we can no longer take for granted in distributed computing.

Fault Tolerance We have not previously discussed the potential of failure in the PRAM model (and nor is it typically a focus in classes on centralized algorithms). The possibility of failure certainly exists in these settings, however, protection from failure tends to be more straightforward. Should a bug in the software, or a hardware fault cause an algorithm to terminate unexpectedly in these settings, the computations that have already been carried out are often unsalvageable—there really is not much more that can be done besides executing the program again.

[1] not a solid state disk

In the distributed setting, there can be hundreds or even thousands of processors carrying out a portion of the computation. This makes the probability that one of the processors fails very likely. To illustrate this idea, consider your laptop; if it fails on average once a year, then if we have 365 laptops, on average, one will fail each day. While we typically ignore fault tolerance in the design of centralized, and parallel algorithms, this issue needs to be addressed when we are dealing with distributed algorithms. In the distributed setting, the failure of one component does not necessarily cause other processes to fail. Consequently, we might hope to be able to automatically detect and recover from faults during the execution of algorithm with minimal re-execution of computations.

When we introduce Resilient Distributed Datasets (RDDs) and Spark, we will see how they achieve efficient fault tolerance efficiently by tracking the the transformations used to build datasets. This ultimately allows Spark to reconstruct portions of the RDD in the case of node-failure.

References

[1] R. Zadeh. *Introduction to Distributed Optimization.*

[2] R. Zadeh. *Distributed Computing with Spark and MapReduce.*