CME 323: Distributed Algorithms and Optimization, Spring 2015
http://stanford.edu/~rezab/dao.
Instructor: Reza Zadeh, Matroid and Stanford.

Lecture 9, 4/25/2016. Scribed by Max Bodoia, Erik Burton, and Nikhil Parthasarathy.

9 Lecture 9

9.1 Scheduling

9.1.1 Problem Definition

An important problem in any parallel or distributed computing setting is figuring out how to schedule jobs optimally. What this means is that a scheduler must be able to assign sequential computation to processors or machines in order to minimize the total time necessary to process all the jobs. We assume that the processors are identical (i.e. each job takes the same amount of time to run on any of the machines). More formally, we are given p processors and an unordered set of n jobs with processing times $J_1, \ldots, J_n \in \mathbb{R}$. Say that the final schedule for processor i is defined by a set of indices of jobs assigned to processor i. We call this set S_i. The load for processor i is therefore, $L_i = \sum_{k \in S_i} J_k$. The goal is to minimize the *makespan* defined as $L_{max} = \max_{i \in \{1,\ldots,p\}} L_i$.

9.1.2 The Greedy Algorithm

The intuition behind the greedy algorithm discussed here is simple: in order to minimize the makespan we don't want to give a job to a machine that already has a large load. Therefore, we consider the following algorithm. Take the jobs one by one and assign each job to the processor that has the least load at that time. This algorithm is simple and is *online* which is necessary for the scheduling task.

9.1.3 Optimality of the Greedy Approach

We now claim that this seemingly naive greedy algorithm actually has a *competitive ratio* of 2. In other words, the algorithm is in the worst-case 2 times worse than the optimal. For this analysis, we define the optimal makespan (minimal makespan possible) to be OPT and try to compare the output of the greedy algorithm to this. We also define L_{max} as above to be the makespan.
Claim: Greedy Algorithm has competitive ratio of 2.
Proof: We first want to get a handle (lower bound) on OPT. We first know that the optimal makespan must be at least the sum of the processing times for the jobs divided amongst the p processors. In other words,

$$OPT \geq \frac{1}{p} \sum_{i=1}^{n} J_i \tag{1}$$

A second lower bound on OPT is that OPT is at least as large as the time of the longest job in the sequence:

$$OPT \geq \max_i J_i \tag{2}$$

Now consider running the greedy algorithm and identifying the processor responsible for the makespan of the greedy algorithm (i.e. k = argmax$_i L_i$). Let J_t be the load of the last job placed on this processor.

Before the last job was placed on this processor, the load of this processor was thus L_{max} - J_t. By the definition of the greedy algorithm, this processor must have also had the least load before the last job was assigned. Therefore, all other processors at this time must have had load at least L_{max} - J_t. Therefore, we clearly have that

$$p(L_{max} - J_t) \leq \sum_{i=1}^{p} L_i = \sum_{i=1}^{n} J_i \tag{3}$$

The last equality comes from the fact that the sum of the loads must be equal to the sum of the processing times for the jobs. Rearranging the terms in this expression let's us express this as:

$$L_{max} \leq \frac{1}{p} \sum_{i=1}^{n} J_i + J_t \tag{4}$$

Using equations 1 and 2 along with the fact that $J_t \leq \max_i J_i$, we get that by using the greedy algorithm:

$$L_{max} \leq OPT + OPT = 2 * OPT \tag{5}$$

This shows that the greedy algorithm provides us with a scheduling time that is not more than 2 times more than the optimal. ■

For more information on the general makespan problem and some of its variants, see this nice explanation.

9.2 Intro to Distributed Computing

9.2.1 Recursion not so cheap

Suppose we want to compute the sum of n items via recursion. In a distributed cluster, we cannot just arbitrarily pair up the n items and apply the binary-tree recursion we learned earlier. This is because two numbers in a pair may lie in different machines, and as network speeds are (relatively) slow the cost of moving data to other machines is very high. The better way to do it is to have each machine sum all of the numbers it contains, and then sum these intermediate sums, as this minimizes the amount of data that needs to be moved between machines.

When machines talk to each other, there are two costs:

Latency: how long does it take to send a bit of information (units = seconds, on the order of 10 milliseconds)

Bandwidth: how much information can we send over per second (units = bits per second, on the order of 1-10 Gigabit/sec)

When summing the integers on their respective machines separately we still need the final sum as we now only have a partial sum on each machine. One way to sum a bunch of integers: have a master machine, all m machines send their sum to this machine, and the master machine computes the final sum. The problem is that this creates a bandwidth bottleneck at the master machine.

A better way: apply the binary-tree parallel computation arrangement across the m machines. Each machine computes it's sum and sends it to the next machine up the tree. The root node then reports the final sum. This way no machine has more than two incoming data-packets and we avoid bottlenecking.

9.2.2 Memory issues

Within each machine there is also a large difference between accessing memory and accessing data on disk:
Main memory seek: 100 nanoseconds
Hard disk seek: 10,000,000 nanoseconds (100k times slower!)

However, the difference decreases significantly when reading a sequence of data:
Read 1Mb from main memory: 250,000 nanoseconds
Read 1Mb from hard disk: 30,000,000 nanoseconds (only 100 times slower)

This gives rise to streaming algorithms, which assume you have limited random-access memory and infinite streaming memory. Some people call these algorithms big-data because hard disks are quite large these days (say 4 Tb). However, they aren't truly big-data because they are inherently bound to the size of hard drives, whereas distributed computing/parallel computing can scale indefinitely.

9.3 Random Stream Sample

Given a stream of items of unknown length, we want to design an algorithm that waits for the stream to finish and then returns one item selected uniformly at random, while only using a sub-linear amount of memory. In fact, we can create an algorithm which maintains a running uniform sample of all the items seen so far as it reads the stream.

9.3.1 Idea

As we read in the stream, we keep a single item at any given time as our running random sample. When the n^{th} item is streamed in, we choose to either keep the old running sample (with probability $(n-1)/n$) or replace it with the latest item in the stream (with probability $1/n$). This ensures that at any point in time, our running sample has a uniform probability of being any of the previous items.

9.3.2 Algorithm

The algorithm works as follows:

1. Initialize return value r as empty, and stream size counter k as 1.
2. Read an item s from the stream, and flip a coin with probability $1/k$. If the coin comes up heads, set $r \leftarrow s$.
3. Increment k.
4. If items remain in the stream, go back to step 2. Otherwise, return r.

Observe that since we only store the values of r and k, our algorithm uses a logarithmic amount memory (note it is not technically a constant amount of memory because k stores the length n of the stream and therefore has at least $\log(n)$ bits). It remains to show that this algorithm correctly maintains a uniform random sample from the stream.

9.3.3 Proof

We claim that after n items of the stream have been read (for any value of n), the value stored in r has equal probability $1/n$ of being any of these n items. We prove this claim by induction.

For the base case, consider the state of the algorithm when $n = 1$ (i.e. after a single item has been read from the stream). In this case, our stream size counter has value $k = 1$, and so r will be set to the value of the first item in the stream with probability $1/k = 1/1 = 1$. Therefore r correctly represents a uniform sample from the singleton set containing just the first item from the stream.

For the inductive step, assume our claim is correct for all sequences up to length $n-1$. We wish to show that it is also correct for sequences of length n. Consider the state of the algorithm after the n^{th} item is read from the stream and processed. Our stream size counter has value $k = n$, and so we will set r to be the newest element of the stream with probability $1/n$ and leave r unchanged with probability $(n-1)/n$.

We wish to show that r has equal probability of being any of the stream items seen so far, i.e. that $\Pr(r = s_i) = 1/n$ for $i = 1, \ldots, n$ (where s_i is the i^{th} item of the stream). We know that r will have the value s_n if and only if it was replaced on the most recent step of the algorithm. Since this replacement occurs with probability $1/n$, it follows that $\Pr(r = s_n) = 1/n$.

Furthermore, we know that if r was not replaced on the most recent step, then it has the same value as it had after $n-1$ steps. By the inductive hypothesis, this value of r represents a uniform random sample of the first $n-1$ stream items (i.e. $\Pr(r = s_i) = 1/(n-1)$ for $i = 1, \ldots, n-1$). Since r is not replaced with probability $(n-1)/n$, we have

$$\Pr(r = s_i) = \frac{n-1}{n} \cdot \frac{1}{n-1} = \frac{1}{n} \qquad \text{for } i = 1, \ldots, n-1$$

This means that after n items have been read from the stream, we have $\Pr(r = s_i) = 1/n$ for every i from 1 to n, which completes the inductive step. Therefore at each stage of the algorithm, r represents a uniform random sample of all the items seen so far in the stream.

9.4 Advantages of Spark

The advantage of Spark is that it automatically handles a lot of the main issues surrounding distributed computing. It may not be the best at any one thing, and the class is not endorsing Spark as the optimal solution, but it is very popular.

First problem solved: On each machine the transfer from disk to memory can be a major bottleneck. Spark offers an automatic solution, which is not always the best, but works reasonably well.

Second problem solved: Cluster commodity machines fail frequently, and software needs to be able to handle these failures. In particular the data must remain available even in the case of hard-drive failure. Spark takes care of this automatically.

Third problem solved: Sending information between machines is expensive, so sending large amounts of data is out of the question. It is much better to send code to machines holding data. Spark handles this for you.

Fourth problem solved: Code for workers, drivers, and schedulers is already written. Although you could spend your time doing this, it just takes time away from what you actually want to work on.

Spark's tradeoff is that it takes away full expressivity of coding and forces you to use specific primitive functions (resembling SQL operations) and data structures (Resilient Distributed Datasets (RDD)).

As a result normal for-loops and arrays are not available. Instead the basic tool is the RDD with accompanying parallel operations.