Algorithms for Massive Data Science	Lecture date: March 18, 2019
Instructor: Ben Moseley	Scribe: Thomas Lavastida

1 Course Motivation

The size of data sets and the rate at which we can collect data has increased greatly in recent years. Contrasting this, it is less understood how to process this data efficiently to improve decision making and analytics in practice. This is not due to a lack of algorithms to solve the corresponding problems, but due to a lack of algorithms that can efficiently process large data sets. The primary bottlenecks are large running times and memory that is too small to store the entirety of a large data set. These bottlenecks have created a need for new approaches to how we model computation and design algorithms. We will focus on models of computation that address these bottlenecks by implicitly enforcing small run time and small memory footprints.

2 Overview of Approaches to Big Data

Broadly speaking, approaches to efficiently processing big data fall into two groups, sequential algorithms and parallel/distributed algorithms.

2.1 Sequential Algorithms

In the sequential world we think of data being stored in some external memory and it is processed by a single processor. An algorithm in this world should try to be efficient with respect to how it accesses the external memory. In other words, we think of algorithms that need to access the external memory often as being inefficient.

In the second half of the course we will study the streaming model, which will be similar to this notion of I/O efficient algorithms. The Streaming model will be formally defined later in the lecture, but for now we can think of a Streaming algorithm as one that only makes one pass through the input and it only needs to store a small part of the input in order to construct a solution to the problem of interest.

2.2 Parallel and Distributed Algorithms

In the parallel/distributed world there are usually many processors that can be used to process the data simulataneously. How memory is handled usually depends on the model. For example, in the PRAM model there are a collection of processors each with access to a large shared memory. The input to the problem is stored on the shared memory, and then the processors work together to solve the problem of interest. To contrast this in message passing models such as CONGEST, the machines have local memory and are able to communicate by sending messages over a communication network. Graphics Processing Units (GPU's) have also been used to speed up data processing tasks.

Our focus on parallel/distributed algorithms will be through the Massively Parallel Model of Computation (MPC), formerly known as the MapReduce (MR) model.

3 Defining the Models

Before formally defining the Streaming and MPC models, we need to recall the classical RAM model of computation.

3.1 The RAM Model

The Random Access Machine, or RAM, model consists of a single processor and pool of memory. The memory is an array of words with each word being addressable by a natural number. For technical reasons, we assume that words are made up of either $O(1)$ or $O(\log n)$ bits, where n is the input size of a problem. The processor is able to read and write to any memory location in $O(1)$ time. It also has a number of basic operations such as $+, -, \times, \div$, testing conditions, branching, etc. Each of these basic operations are assumed to take $O(1)$ time. Thus the goal in designing algorithms for the RAM model is to minimize the total number of operations used by the algorithm.

The motivation for this model is that it captures a simplified view of how a standard computer system works. However, there are many aspects of real systems that are not captured by this model. Typically, a real system has a memory hierarchy with multiple levels of cache, DRAM, and disk memory that all have to communicate with each other. This typically induces bandwith and latency costs that are not captured by the RAM model. Modern CPUs also have many cores that can work on different tasks in parallel. Despite this difference between theory and practice, the RAM model has been effective in finding the core techniques for designing efficient algorithms.

3.2 Motivation for Streaming

In the Streaming model we imagine that there is a huge amount of data arriving one piece at a time, and there is limited space. As each piece of data arrives, the algorithm will process and then discard it. For correctness, the algorithm must maintain some function $f(A_i)$ where A_i is the first i pieces of data.

In practice, this scenario may be realized by a large data set that is stored on a number of storage disks. The processor may have a small amount of cache memory where it can store a very small "sketch" of the data in order to maintain the function $f(A_i)$ at each step. Accessing data from the disk incurs a high time cost, so it is desirable to minimize the number of passes made over the data.

3.3 The Vanilla Streaming Model

In the Streaming model there is a single processor with a very limited amount of memory. The processor makes one pass over the input data, receiving each piece of the input in discrete time steps. The input is an ordered set $A = \{a_1, a_2, \ldots, a_m\}$ and we let $A_i = \{a_1, \ldots, a_i\}$ be the first i elements. Each item a_i has size $O(\log n)$ and in step i the algorithm receives item a_i. The goal is either to compute some function $f(A)$, that is the solution to a problem of interest, or be able to return $f(A_i)$ at step i. Rather than storing all elements of A_i and naively computing $f(A_i)$, the algorithm must use a limited amount of space while still being able to return $f(A_i)$. In general, the algorithm must use space $o(\min\{m, n\})$, with an ideal space of requirement of $O(\log m + \log n)$. Relaxed versions of this model allow for more than one pass over the data.

Streaming is a very well understood area of research, with many standard techniques and algorithms. This contrasts with the growing area of Massively Parallel algorithms, which are not as well understood.

3.4 Motivation for MPC

The MPC model is motivated by practical distributed computing frameworks such as MapReduce and Spark. These frameworks act as a somewhat restricted programming language that allow the user to easily deploy code on 10's to 1000's of machines to process large data sets in parallel. These languages are restricted in a way in order to ensure efficient machine utilization as well as easy parallelization.

Initially, these frameworks were used for very simple data processing tasks such as computing word frequency in a document. In order to solve more interesting problems using these frameworks, we can model the essential aspects of these frameworks. The goal of these models is to capture the key constraints in writing efficient code within these frameworks. This has opened up the possibility of being able to solve big data problems we cannot currently handle. There are many open problems with potential for high impact in this area.

3.4.1 How do these Frameworks Operate?

Typically, there is a set of m machines, connected by a communication network that is a complete graph. That is each machine can send and receive messages to/from any other machine. Initially, the data is equally distributed across all the machines in either an arbitrary or random manner. The computation proceeds in synchronous *rounds*. During a round, each machine can process its local data by running a sequential algorithm. The round ends when all machines complete their local computation and syncrhonize. In between rounds, the machines communicate with each other to exchange data. The communication between rounds is one of the highest bottlenecks in performance for these frameworks.

There are several key features that we need to model. First, since rounds are synchronous, the length of a round is bounded by the longest computation run by a machine. Thus we need local computations to be efficient. Next, since communication is significant bottleneck we need to minimize the total amount of communication. We use the number of rounds as a simple surrogate for the total amount of communication. To avoid trivial solutions, we require that the local memory of each machine is small compared to the overall size of the data. We also require that a machine can communicate no more data than the size of its local memory each round.

There is a brief history for these models starting with the paper of Karloff, Suri, and Vassilvitskii in SODA 2010 [1]. The model led to clean algorithmic ideas that were implementable in practice. Over time the model has been refined, so we will focus on a more recent version of the model.

3.5 The MPC Model

In the MPC model, there are m machines and an input of total size n. Each machine is a RAM with a bounded amount of memory. We require the space/memory per machine S to be $S = \tilde{O}(n/m)$, where $\tilde{O}(\cdot)$ hides factors that are polylogarithmic in n or m. Thus the total amount of memory across all machines is $\tilde{O}(n)$. Generally, we will have $m = n^{\epsilon}$ for some constant $\epsilon \in (0, 1)$. In some cases we can get algorithms that work for any contant $\epsilon \in (0, 1)$, but this is not necessary. This

restriction enforces that the number of machines and the memory per machine is sublinear in n. This is an important restriction since we think of the data set as being very large.

Ideally, the number of rounds an algorithm uses would be constant, but we generally allow for polylogarithmic in n number of rounds. Moreover, the local computation on each machine in each round should be a polynomial time algorithm in the RAM model using at most S space. Theoretically, we allow for abitrary polynomial time local computations, but in practice it is important to keep local computation as efficient as possible.

Communication occurs between rounds and can occur between any pair of machines. Each machine can communicate at most its local memory amount of data in a round.

3.5.1 The Extra Memory Setting

Sometimes it is useful that we have slightly more total memory when trying to generate algorithmic ideas. After finding a good algorithm for this setting, it can then be refined to fit into the more restrictive case above. The extra memory setting is as follows. The total memory is $\tilde{O}(n^{1+\epsilon/2})$ for some $\epsilon \in (0, 1)$. Taking $m = \Theta(n^{\epsilon})$ ensures that the number of machines is sublinear in n and that the memory per machine is $\tilde{O}(n^{1+\epsilon/2}/n^{\epsilon}) = \tilde{O}(n^{1-\epsilon/2})$ and thus also sublinear in n.

3.5.2 Key Constraints

Data is either randomly or arbitrarily distributed among the machines. Data can only be processed locally, and no communication between machines occurs during a round, only between rounds. How data is communicated between machines must be decided locally, i.e. there is no coordinator that makes communication decisions for machines. Each machine locally decides which other machines to send messages to and what to send.

Unlike the Streaming model, no single machine sees the entire data set. This is a result of requiring sublinear memory per machine and at most polylogarithmic in n rounds of computation. The maximum amount of data a single machine can see under these restrictions is $\tilde{O}(n^{1-\epsilon}) \cdot \text{polylog}(n) = o(n)$. This requires algorithms to really break up a problem and use parallelism to find solutions.

3.5.3 Pros and Cons of the Model

A strong pro of the model is that it is simple and elegant to describe. For the most part, it captures the major constraints in effectively using frameworks such as Spark and MapReduce, while hiding many of the ugly details. This allows us to identify the key algorithmic challenges for finding scalable solutions to a problem. This model is also different from past models such as PRAM, CONGEST, and Congested Clique, so in principle we can find very different solutions to problems. In general we do worst case analysis, which yields robust solutions.

A con of this model is that in some cases it may be too simplified. Complicated algorithms designed for this model with strong bounds on the number of rounds may not translate well to efficient code in Spark or MapReduce. Some problems may not fit into this model at all. Moreover, it is hard to find lower bounds due to a connection to results in circuit complexity [2]. Practical inputs may not be worst case, so the worst case results we show may be very pessimistic in practice.

4 Word Frequency Count in 2 Rounds

Now that we have seen the MPC model lets see a problem and give a simple 2 round algorithm to solve it. In this problem, the input is a collection of documents. Each document is a tuple (ID, W) consisting of a numeric label ID and a list of words W, where words are possibly duplicated. To solve the problem we must output for each distinct word the number of times it appears across all documents. For example if the input is $(0, [\text{'big'}, \text{'data'}]), (1, [\text{'big'}])$, the output is $(\text{'big'}, 2), (\text{'data'}, 1)$.

The total input size N is the total number of words across all documents. Since many words are repeated often in documents (such as "the"), we assume that the number of distinct words is $O(\sqrt{N})$. We also assume that each document has at most $O(\sqrt{N})$ words and that words are constant size. We give a 2 round algorithm using $m = \Theta(\sqrt{N})$ machines. Initially the documents are distributed across the m machines.

Round 1

In the first round, for each distinct word w on a machines local memory, the machine computes the frequency count of w on its local documents. Let $c_{w,i}$ be the count of word w on machine i. Now each word w picks a unique machine. Each machine i communicates $c_{w,i}$ to the machine designated for word w.

Round 2

The machine for each word w computes $\sum_{i=1}^{m} c_{w,i}$ and outputs the resulting count.

Analysis

The algorithm runs in two rounds and it clearly computes the correct result. Let's check that it falls into the memory restrictions. Initially, the data is partitioned equally across the machines so the memory per machine is $O(N/m) = O(\sqrt{N})$ when $m = \Theta(\sqrt{N})$. Thus we satisfy the memory requirements with $\epsilon = 1/2$. Since there are $O(\sqrt{N})$ distinct words, each can be assigned a unique machine for this choice of m, moreover each machine sends and receives at most $O(\sqrt{N})$ pieces of information between rounds 1 and 2, satisfying the communication requirements.

Note that we made several assumptions for this analysis. The algorithm can be made more robust to relax some of these assumptions.

References

[1] Howard J. Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for mapreduce. In Moses Charikar, editor, *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010, Austin, Texas, USA, January 17-19, 2010*, pages 938–948. SIAM, 2010.

[2] Tim Roughgarden, Sergei Vassilvitskii, and Joshua R. Wang. Shuffles and circuits (on lower bounds for modern parallel computation). *J. ACM*, 65(6):41:1–41:24, 2018.